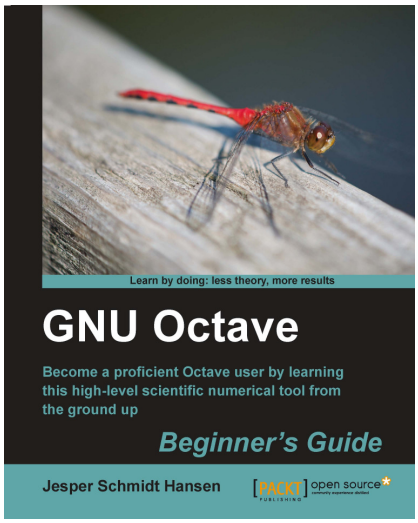
[PACKT] open source
community experience distilled
PUBLISHING

GNU Octave Beginner's Guide

Jesper Schmidt Hansen



Chapter No. 7
"More Examples: Data Analysis"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.7 "More Examples: Data Analysis"

A synopsis of the book's content

Information on where to buy this book

About the Author

Jesper Schmidt Hansen holds a Ph.D. in soft material science and is currently doing research in the field of nanofluidics and dynamics at Roskilde University, Denmark. He has been using GNU Octave on a daily basis for many years, both as a student and later as a researcher. The applications have varied from solving partial and ordinary differential equations, simple data plotting, data generation for other applications, dynamical system investigations, and advanced data analysis.

Firstly, I wish to thank the reviewers. They have been a great help and their many (at times overwhelmingly many) comments and hints have improved the manuscript considerably.

I have received encouragement and good ideas from everyone at the Department of Science, Systems and Models, Roskilde University. Especially, I want to thank Professor Jeppe Dyre from the Danish National Research Foundation centre "Glass and Time" for giving me the opportunity to finish the book in the last phase of the writing.

Also, I have found Octave's official mailing list very useful. Unlike many other user groups, there is a very constructive and helpful atmosphere here. I thank everyone who has submitted questions and all those that have replied.

I now realize that having a one year old child, a full time job, as well as writing a book is not really an ideal cocktail. I must thank Signe Anthon for her tremendous support and patience during the writing of this book. When I signed the contract with Packt Publishing, I was happy finally to be able to make a contribution to the open source community—Signe's contribution is just as big as mine!

GNU Octave Beginner's Guide

Using a range of very different examples, this beginner's guide will take you through the most important aspects of GNU Octave. The book starts by introducing how you work with mathematical objects like vectors and matrices, demonstrating how to perform simple arithmetic operations on these objects and explaining how to use some of the simple functionality that comes with GNU Octave, including plotting. It then goes on to show you how to extend and implement new functionality into GNU Octave, how to make a toolbox package to solve your specific problem, and how to use GNU Octave for complicated data analysis. Finally, it demonstrates how to optimize your code and link GNU Octave with C++ code enabling you to solve even the most computational demanding tasks. After reading GNU Octave Beginner's Guide, you will be able to use and tailor GNU Octave to solve most numerical problems and perform complicated data analysis with ease.

What This Book Covers

Chapter 1, Introducing GNU Octave briefly introduces you to GNU Octave. It explains how you can install GNU Octave and test your installation. This first chapter also discusses how to customize the appearance and the behavior of GNU Octave as well as how you install additional packages.

Chapter 2, Interacting with Octave: Variables and Operators shows you how to interact with GNU Octave through the interactive environment. Learn to instantiate objects of different types, control their values, and perform simple operations on and between them.

Chapter 3, Working with Octave: Functions and Plotting explains GNU Octave functions and shows several examples of the very useful functionalities that come with GNU Octave. In this chapter, you will see how you can perform two- and three-dimensional plotting, control the graph appearance, how to have multiple plots in the same figure window, and much more.

Chapter 4, Rationalizing: Octave Scripts looks at how you can rationalize your work using scripts. It will teach you how to control the programming flow in your script and how to perform loops using different statements. At the end of the chapter, you are shown how you can save your work and load it back into GNU Octave's workspace.

Chapter 5, Extensions: Write Your Own Octave Functions takes a closer look at functions and teaches how you can write your own GNU Octave functions. You will learn how to control and validate user input to the function. The important concept of vectorization is discussed and an example of this is given in the last part of the chapter.

Chapter 6, Making Your Own Package: A Poisson Equation Solver teaches you how to make your own GNU Octave package from a collection of related functions. The package will be able to solve one- and two-dimensional Poisson equations and is therefore relevant for many problems encountered in science and engineering. In this chapter, you will also learn how to work with sparse matrices in GNU Octave.

Chapter 7, More Examples: Data Analysis shows you examples of how GNU Octave can be used for data analysis. These examples range from simple statistics, through data fitting, to Fourier analysis and data smoothing.

Chapter 8, Need for Speed: Optimization and Dynamically Linked Functions discusses how you can optimize your code. This includes vectorization, partial looping, pre-instantiation of variables, and dynamically linked functions. The main part of the chapter shows how to use GNU Octave's C++ library and how to link this to the GNU Octave interactive environment. Special attention is paid to explaining when and when not to consider using dynamically linked functions.

7
More Examples: Data Analysis

Octave is an ideal tool to perform many different types of data analysis. The data can be generated by other programs or be collected from a database and then loaded into Octave's workspace. The data analysis tools in Octave are based on a truly impressive arsenal of different functions. We will only discuss a few of them here, namely, how to perform the simplest statistical analysis, function fitting, and Fourier (or spectral) analysis using the fast Fourier transform.

In brief terms, upon reading this chapter, you will learn:

- More about the ASCII file formats that can be loaded into Octave's workspace.
- How you can use Octave to perform simple descriptive statistics.
- About fitting different functions to data.
- How to use Octave to perform Fourier analysis.

Loading data files

When performing a statistical analysis of a particular problem, you often have some data stored in a file. In *Chapter 4*, it was shown how you can save your variables (or the entire workspace) using different file formats and then load them back in again. Octave can, of course, also load data from files generated by other programs. There are certain restrictions when you do this which we will discuss here. In the following matter, we will only consider ASCII files, that is, readable text files.

When you load data from an ASCII file using the `load` command (see *Chapter 4*), the data is treated as a two-dimensional array. We can then think of the data as a matrix where lines represent the matrix rows and columns the matrix columns. For this matrix to be well defined, the data must be organized such that all the rows have the same number of columns (and therefore the columns the same number of rows). For example, the content of a file called `series.dat` can be:

```
1    232.1    334

2    245.2    334

3    456.23   342

4    555.6    321
```

In *Chapter 4*, we learned how to load this into Octave's workspace:

```
octave:1> load -ascii series.dat;
```

whereby the data is stored in the variable named `series`. In fact, Octave is capable of loading the data even if you do not specify the ASCII format. The number of rows and columns are then:

```
octave:2> size(series)

ans =
  4     3
```

I prefer the file extension `.dat`, but again this is optional and can be anything you wish, say `.txt`, `.ascii`, `.data`, or nothing at all.

In the data files you can have:

- Octave comments
- Data blocks separated by blank lines (or equivalent empty rows)
- Tabs or single and multi-space for number separation

Thus, the following data file will successfully load into Octave:

```
# First block

1    232    334

2    245    334

3    456    342

4    555    321
```

```
# Second block
1            231          334
2            244          334
3            450          341
4            557          327
```

The resulting variable is a matrix with 8 rows and 3 columns. If you know the number of blocks or the block sizes, you can then separate the blocked-data.

Now, the following data stored in the file `bad.dat` will not load into Octave's workspace:

```
1    232.1    334
2    245.2
3    456.23
4    555.6
```

because line 1 has three columns whereas lines 2-4 have two columns. If you try to load this file, Octave will complain:

```
octave:3>  load -ascii bad.dat
error: load: bad.dat: inconsisitent number of columns near line 2
error:load: unable to extract matrix size from file 'bad.dat'
```

Simple descriptive statistics

In *Chapter 5*, we implemented an Octave function `mcintgr` and its vectorized version `mcintgrv`. This function can evaluate the integral for a mathematical function f in some interval $[a; b]$ where the function is positive. The Octave function is based on the Monte Carlo method and the return value, that is, the integral, is therefore a stochastic variable. When we calculate a given integral, we should as a minimum present the result as a mean or another appropriate measure of a central value together with an associated statistical uncertainty. This is true for any other stochastic variable, whether it is the height of the pupils in class, length of a plant's leaves, and so on.

In this section, we will use Octave for the most simple statistical description of stochastic variables.

Histogram and moments

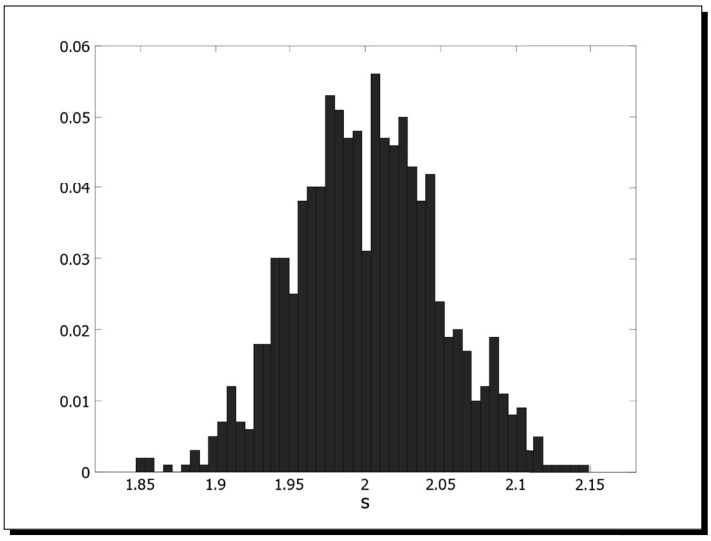
Let us calculate the integral given in Equation (5.9) one thousand times using the vectorized version of the Monte Carlo integrator:

```
octave:4> for i=1:1000

> s(i) = mcintgrv("sin", 0, pi, 1000);

> endfor
```

The array `s` now contains a sequence of numbers which we know are approximately 2. Before we make any quantitative statistical description, it is always a good idea to first plot a histogram of the data as this gives an approximation to the true underlying probability distribution of the variable `s`. The easiest way to do this is by using Octave's `hist` function which can be called using:

```
octave:5> hist(s, 30, 1)
```

The first argument, `s`, to `hist` is the stochastic variable, the second is the number of bins that `s` should be grouped into (here we have used 30), and the third argument gives the sum of the heights of the histogram (here we set it to 1). The histogram is shown in the figure below. If `hist` is called via the command `hist(s)`, `s` is grouped into ten bins and the sum of the heights of the histogram is equal to `sum(s)`.



From the figure, we see that `mcintgrv` produces a sequence of random numbers that appear to be normal (or Gaussian) distributed with a mean of 2. This is what we expected. It then makes good sense to describe the variable via the sample mean defined as:

$$\bar{s} = \frac{1}{N}\sum_{i=i}^{N} s_i \,, \tag{7.1}$$

where N is the number of samples (here 1000) and s_i the i'th data point, as well as the sample variance given by:

$$\text{var}\,(s) = \frac{1}{N-1}\sum_{i=i}^{N} (s_i - \bar{s})^2. \tag{7.2}$$

The variance is a measure of the distribution width and therefore an estimate of the statistical uncertainty of the mean value. Sometimes, one uses the standard deviation instead of the variance. The standard deviation is simply the square root of the variance $\sigma = \sqrt{\text{var}\,(s)}$.

To calculate the sample mean, sample variance, and the standard deviation in Octave, you use:

```
octave:6> mean(s)
ans = 1.9999
octave:7> var(s)
ans = 0.002028
octave:8> std(s)
ans = 0.044976
```

In the statistical description of the data, we can also include the skewness which measures the symmetry of the underlying distribution around the mean. If it is positive, it is an indication that the distribution has a long tail stretching towards positive values with respect to the mean. If it is negative, it has a long negative tail. The skewness is often defined as:

$$\text{skew}\,(s) = \frac{1}{N}\sum_{i=1}^{N} \left(\frac{s_i - \bar{s}}{\sigma}\right)^3. \tag{7.3}$$

We can calculate this in Octave via:

```
octave:9> skewness(s)

ans = -0.15495
```

This result is a bit surprising because we would assume from the histogram that the data set represents numbers picked from a normal distribution which is symmetric around the mean and therefore has zero skewness. It illustrates an important point—be careful to use the skewness as a direct measure of the distributions symmetry—you need a very large data set to get a good estimate.[1]

[1] *Also see discussion in "Numerical Recipes: The Art of Scientific Computing, 3rd Edition", Press et al., Cambridge University Press (2007).*

You can also calculate the kurtosis which measures the flatness of the sample distribution compared to a normal distribution. Negative kurtosis indicates a relative flatter distribution around the mean and a positive kurtosis that the sample distribution has a sharp peak around the mean. The kurtosis is defined by the following:

$$\text{kurt}\,(s) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{s_i - \bar{s}}{\sigma}\right)^4 - 3. \tag{7.4}$$

It can be calculated by the `kurtosis` function.

```
octave:10> kurtosis(s)

ans = -0.02310
```

The kurtosis has the same problem as the skewness—you need a very large sample size to obtain a good estimate.

Sample moments

As you may know, the sample mean, variance, skewness, and kurtosis are examples of sample moments. The mean is related to the first moment, the variance the second moment, and so forth. Now, the moments are not uniquely defined. One can, for example, define the k' th absolute sample moment p_a^k and k' th central sample moment p_c^k as:

$$p_a^k = \frac{1}{N}\sum_{i=1}^{N} s_i^k \quad \text{and} \quad p_a^k = \frac{1}{N}\sum_{i=1}^{N} (s_i - \bar{s})^k. \tag{7.5}$$

Notice that the first absolute moment is simply the sample mean, but the first central sample moment is zero. In Octave, you can easily retrieve the sample moments using the `moment` function, for example, to calculate the second central sample moment you use:

```
octave:11> moment(s, 2, 'c')

ans = 0.002022
```

Here the first input argument is the sample data, the second defines the order of the moment, and the third argument specifies whether we want the central moment `'c'` or absolute moment `'a'` which is the default. Compare the output with the output from Command 7—why is it not the same?

Comparing data sets

Above, it was shown how you can use Octave to perform the very simplest statistical description of a single data set. In this section, we shall see how to statistically compare two data sets. What do we exactly mean by a statistical comparison? For example, we could test if two data sets statistically have the same means (this is known as the student's t-test), or if they have the same underlying probability distribution (the χ^2 – test).

In Octave, you can perform almost all known statistical tests: the student's t-test, z-test, the Kolmogorov-Smirnov test, and many more. Here I will only show you how to perform one variance of the t-test and how to compute the Pearson correlation coefficient.

The correlation coefficient

The following table shows the height and weight of boys from age 2 to 15:

Age (Years)	Weight (Kilograms)	Height (Centi-metres)	Age (Years)	Weight (Kilograms)	Height (Centi-metres)
2	12.5	85.5	9	24.9	129.0
3	13.2	93.2	10	27.3	134.6
4	15.2	102.3	11	31.3	139.8
5	17.8	102.4	12	35.0	146.3
6	19.7	113.9	13	39.6	152.1
7	20.9	119.8	14	43.8	158.1
8	22.5	123.7	15	53.0	162.8

One can easily see that both height and weight are increasing with respect to age. To see if the two data sets are actually correlated, we need to be a bit more careful. Usually the correlation between two data sets is quantified by using the Pearson's correlation coefficient which is given by:

$$r_p = \frac{\sum_{i=1}^{N} (x_i - \bar{x})(y_i - \bar{y})}{(N-1)\,\sigma_x \sigma_y}\,, \tag{7.6}$$

where σ_x is the standard deviation of the data set x_i and σ_y is the standard deviation of y_i. Values of r_p around one indicates good correlation between the two data sets. The Peason correlation coefficient is easily calculated in Octave using `cor` (or `corrcoef`). This function has the syntax:

```
r = cor(x, y)
```

No need to explain, I guess.

Assume that we have stored the data in an ASCII file called `boys.dat` like this:

```
# Age    weight   Height
  2     12.5      85.5
  3     13.2      93.2
```

```
...

octave:12> load -ascii boys.dat;
```

We then need to find the correlation between the second and the third column:

```
octave:13> cor(boys(:,2), boys(:,3))

ans = 0.97066
```

That is, the two data sets are indeed correlated, which we would expect.

The student t-test

The following sequence of numbers shows the height of the pupils in a class of 21 children (in centimetres):

156.92 140.00 163.20 167.24 149.84 149.21 166.86 152.01 147.53 157.56 154.48 170.33 155.82 162.24 161.43 174.94 146.30 151.08 150.82 154.49 165.98

The mean is 157.07 centimetres. The national average height is 161.11 centimetres. Under the assumption that the height of the pupils is normal distributed around the mean, can we show that the mean is statistically the same as the national average? Octave's `t_test` can help. A simple version of the syntax is:

```
pvalue = t_test(x, m)
```

Here `pvalue` is the probability that the null hypothesis (that the two means are the same) is true, `x` is the data, and `m` is the mean that we test against.

Suppose we have the heights stored in an array called `heights`. To perform the test, we use:

```
octave:14> t_test(heights, 161.11)

ans = 0.0508369
```

which means that we cannot definitely conclude that the mean height in the class is the same as the national average height assuming no variance in the latter. Usually one accepts the null hypothesis for `pvalue > 0.05`, so here we have a border-line situation.

The table below lists some of the most common statistical test functions:

Function name	Description
`t_test(x, m, opt)`	Tests the null-hypothesis that the normal distributed sample data `x` has mean `m`.
`t_test2(x, y, opt)`	Tests the null-hypothesis that the normal distributed sample data `x` and `y` has same mean.

Function name	Description
`kolmogorov_smirnov_test(x, dist)`	Tests the null-hypothesis that the sample data `x` comes from the continuous distribution `dist`.
`kolmogorov_smirnov_test_2(x, y, opt)`	Tests the null-hypothesis that the sample data `x` and `y` come from the same continuous distribution.
`var_test (x, y, opt)`	Tests the null-hypothesis that the normal distributed sample data `x` and `y` have same variance (F-test).
`chisquare_test_homogeneity(x,y,c)`	Tests the null-hypothesis that `x` and `y` come from the same probability distribution using the bins given via `c`.

Function fitting

In many areas of science, you want to fit a function to data. This function can represent either an empirical or a theoretical model. There are many reasons to do this, for example, if the theoretical model agrees with the observed data values, the theory is likely to be right and hopefully you have gained new insight into the phenomenon you are studying.

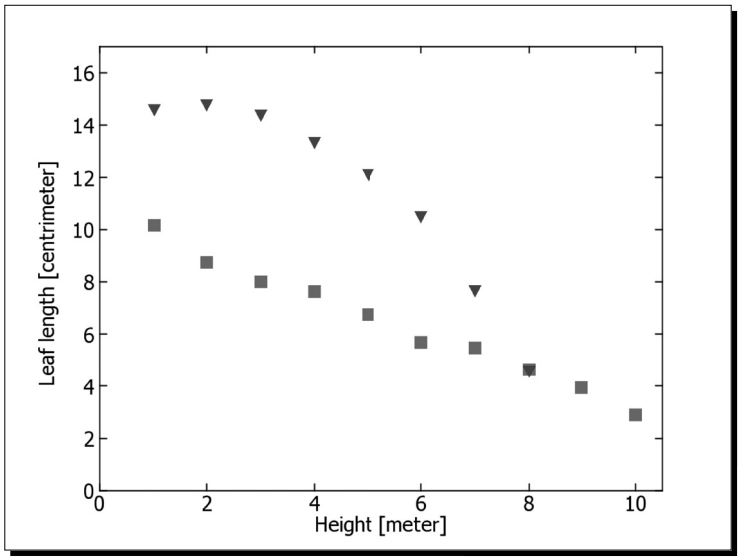
In this section, we will discuss some of Octave's fitting functionality. I will not go into details with the algorithms that are behind the fitting functions—this will simply take up too much space and not be of much relevance for the points.

Polynomial fitting

Suppose we want to investigate the length of the leaves in two different families of trees at different heights. Normally the leaves are largest near the ground, in order to increase the photosynthesis. The figure below shows fabricated data of the leaf length as a function of height from the ground for two imaginary families of trees called tree A (red squares) and tree B (blue triangles). For some reason, we have the idea that the leaf length for tree A, we denote this by y^A, is a linear function with respect to height x, but for tree B, the leaf length y^B follows a third order polynomial with respect to height. That is, we should test if the models:

$$y^A(x) = c_1 x + c_0 \quad \text{and} \quad y^B(x) = c_3 x^3 + c_2 x^2 + c_0 \tag{7.7}$$

can fit the data well if we use the polynomial coefficients as fitting parameters.



In Octave this is straightforward using `polyfit`. This function can be called via the following syntax:

```
[cfit s] = polyfit(x, y, n)
```

where `x` is the independent/free variable (in this case the height), `y` is the measured data (leaf length), and `n` is the degree of the polynomial. The first output is an array with the polynomial coefficients and therefore has length `n+1`, and the second is a structure that contains information about the fit. We shall study some the important structure fields below.

Time for action – using polyfit

1. Assume that we have loaded the relevant data into Octave and stored the leaf lengths of tree A and B in variables `yA` and `yB`. The corresponding heights are stored in `xA` and `xB`.

2. To fit the linear model to the data in `yA`, we do the following:

```
octave:15> [cA sA] = polyfit(xA, yA, 1);
```

3. We must check if the fit made any sense at all. First, we can plot the resulting linear fit together with the data:

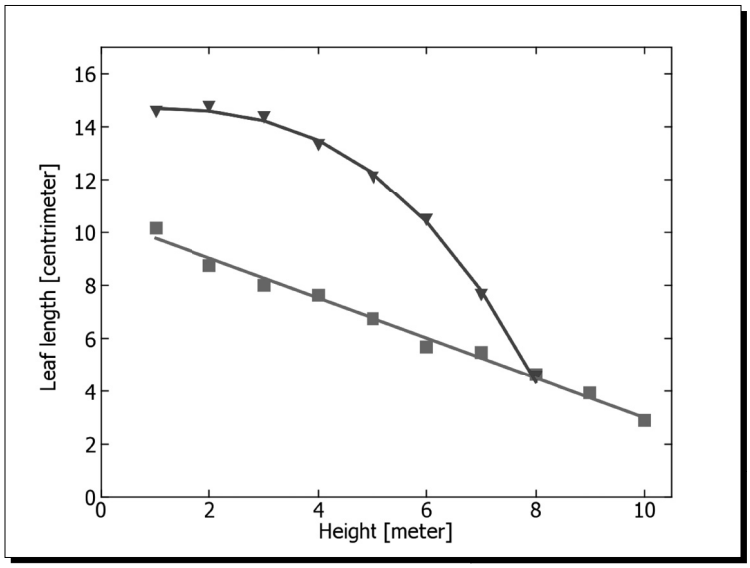
```
octave:16> plot(xA, yA, 'rs', xA, sA.yf, 'r');
```

and is shown in the figure below with red squares and a red line.

4. The fit of y^B to the third order polynomial follows the same procedure:

```
octave:17> [cB  sB] = polyfit(xB, yB, 3);

octave:18> hold on; plot(xB, yB, 'bv', xB, sB.yf, 'b');
```

The plot is shown in the figure below:



Notice that I have used the plotting style format `'rs'` and `'bv'`. Depending on your plotting backend, this may not be supported. You can change it to, for example, `'ro'` and `'b+'`

What just happened?

`polyfit` finds the polynomial coefficients c_n, c_{n-1},...,c_1, c_0 such that the difference between the polynomial (our statistical model) $y = y\,(x_i, c_n, c_{n-1},...,c_1, c_0)$ and the data points y_i is minimized by some measure. This measure is the sum of the residuals, $r_i = y_i - y$, that is:

$$\text{find } c_n, c_{n-1}, \ldots, c_1, c_0 \text{ to minimize } \sum_{i=1}^{N} r_i^2 = \sum_{i=1}^{N} (y_i - y)^2,$$

where N is the number of fitting points. This fitting procedure is called a least squares fit.

As mentioned above, `polyfit` returns a structure that stores information about the fit. In Command 16, we use one of the structure fields `yf` that contains the fitted values $y\ (x_i, c_n, c_{n-1},...,c_1,c_0)$ to plot the resulting fit. We could alternatively have used the polynomial coefficients returned in `cA`:

```
octave:19> cA
cA =
    -0.75172     10.52164
```

Using `polyval` (from *Chapter 3*):

```
octave:20> plot(xA, yA, 'rs', xA, polyval(cA, xA), 'r');
```

Goodness of the fit

From the figure above, it looks as if the fits represent the data quite well, that is, the polynomials seem to be good models of the leaf length variation. A visual verification of the fits and models in general is not really enough. Scientists often prefer some objective quantity to indicate whether the fit is satisfactory or not.

`polyfit` stores a quantity in the structure field `normr`, namely the 2-norm of the residuals. This is given by:

$$\text{norm}(r_i, 2) = \sqrt{\sum_{i=1}^{N} r_i^2}. \tag{7.8}$$

This is however not of much help here because this quantity depends on the absolute values of the residuals. One can instead use the correlation coefficient:

$$r_{\text{cor}} = 1 - \frac{\text{norm}(r_i, 2)^2}{(N-1)\sigma^2}, \quad 0 \leq r_{\text{cor}} \leq 1. \tag{7.9}$$

You can see that for small residuals (and possibly a good fit), the correlation coefficient is close to 1; if the fit is poor, it is close to 0. Unfortunately, `polyfit` won't calculate the quantity for you, but you can easily do it yourself.

Time for action – calculating the correlation coefficient

Let us try to calculate the correlation coefficient for the fit of the leaf length for tree A. We just need to follow Equation (7.9):

```
octave:21> denom = (length(yA) -  1)*var(yA);

octave:22> rcor = 1 - sA.normr^2/denom
rcor =  0.96801
```

This gives an indication that the fit is good as we expected.

What just happened?

In Command 21, we calculated the denominator in Equation (7.9). Notice that instead of calculating the square of the standard deviation, we simply use the variance found with `var`. From Equation (7.9), we see that the 2-norm of the residuals enters the nominator. This is already calculated in `polyfit` and stored in the structure field `normr`, so we use this in the evaluation of the correlation coefficient.

Residual plot

If there is a systematic deviation between the fit and the data, the model may have to be rejected. These deviations can be sufficiently small and are therefore not captured by the correlation coefficient. They can, however, be seen via residual plots. In Octave, it is straightforward to do, and I leave this to you to do as an exercise.

Non-polynomial fits

Of course, not everything can be described via simple polynomial models. Phenomena related to growth are for example often described via an exponential function or a power law. These two examples are trivial to solve with the polynomial fitting technique discussed above, but you may have particular fitting models that need a more general fitting procedure. Octave can help you with this as well.

Transforms

Before we discuss how to fit more general functions to data, it is worthwhile to see if we can transform the fitting function into a different form that allows us to use the polynomial fitting procedure above.

A very simple example of such a transform is if the data set follows a power law function, say:

$$y(x) = bx^a. \tag{7.10}$$

We can transform this into a linear equation by taking the logarithm on both sides:

$$\ln(y) = \ln(b) + a\ln(x). \tag{7.11}$$

In this way, the new variable $y' = \ln(y)$, is a linear function of $x' = \ln(x)$, that is, we can write $y' = ax' + b'$ with $b' = \ln(b)$. We then fit Equation (7.11) to the transformed data using the `polyfit` function. Remember to transform the fitted parameter b back using the inverse transform, $b = e^{b'}$.

In the example above, the transform is trivial. You could consider other possible ways of transforming your model. For example, if it is possible to Fourier transform the data and the model it could perhaps be a better choice to perform the fit in Fourier space. We will discuss the Fourier transform later in this chapter.

General least squares fitting

In case you cannot transform the model into a proper form that allows you to use `polyfit`, you can use `leasqr`. This function comes with the optimization package **optim** which can be downloaded from the Octave-Forge web page. See *Chapter 1* on how to install and load packages. `leasqr` is a really powerful function that allows you to perform even the most complicated fits—if I must choose my favorite Octave function, I think this would be it.

The syntax is as follows:

```
[yfit pfit cvg iter ...] = leasqr(x, y, p, fun, opt)
```

Describing all the inputs and outputs in sufficient detail would take a great deal of effort, so we shall only discuss a limited number of features, but these will take us quite far. If you are interested in knowing more about the function simply type: `help leasqr`.

The inputs are:

`x`: the independent variable

`y`: the measured/observed data

`p`: an initial parameter guess

`fun`: the model (fitting function)

`opt`: up to six optional arguments specifying the weights of each data point, maximum number of iterations the algorithm can perform, and so on

`leasqr` can return 10 outputs, the first four are:

`yfit`: the fitted function values

`pfit`: the fitted parameter values

`cvg`: is 1 if the fit converged (likely to be successful), 0 if not

`iter`: number of iterations done

The crucial point when using `leasqr` is that you must provide the model in the form of an Octave function that can be fitted to the data set. This function must be dependent on at least one parameter and follow the syntax:

```
y = fun(x,p)
```

Again, `x` is the free variable, and `p` is the parameter list.

Let us illustrate `leasqr` by fitting the following two parameter model:

$$y(x) = \frac{1}{1 + \alpha x^{\beta}} \tag{7.12}$$

to some data that we simply generate ourselves. Note, α and β are the fitting parameters.

Time for action – using leasqr

1. Let us generate data with normally distributed random noise:

```
octave:23> x=linspace(0, 5); y = 1./(1 + 1.2*x.^1.8) + \
    > randn(1,100)*0.03;
```

2. Then we specify the model using the following function definition:

```
octave:24>  function y = ffun(x, p)

> y = 1./(1+p(1)*x.^p(2));

> endfunction
```

3. Give an initial guess of the parameters α and β:

```
octave:25> p = [0.5 0.0];
```

4. We can now fit the model to data:

```
octave:26> [yfit pfit cvg iter] = leasqr(x, y,  p, "ffun");
```

   Easy!

5. We can check whether the fitting algorithm converged or not, and how many iterations it used:

```
octave:27> cvg, iter

cvg = 1
iter = 6
```

6. The values of the fitted parameters are of course important:

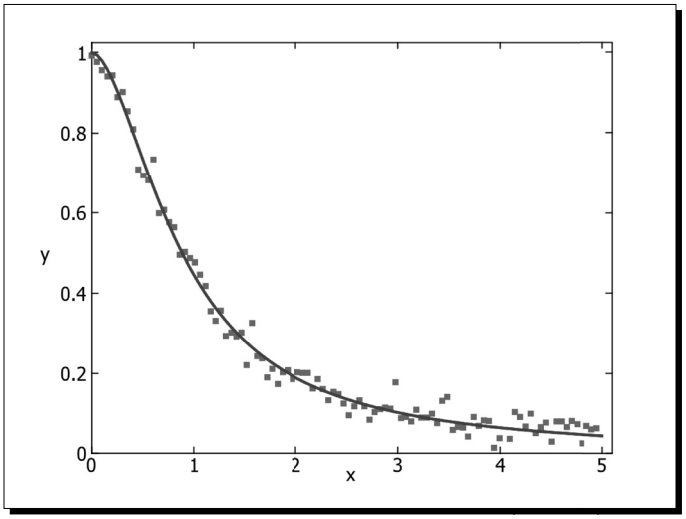
```
octave:28> pfit

p =
  1.1962
  1.7955
```

   This is very close to the values that we would expect from Command 24. The fit is plotted together with the data in the figure below.

What just happened?

In Command 23, we instantiated the free variable `x`, and set it to be a vector with element values between 0 and 5 and consisting of one hundred elements. The variable `y` then plays the role of the dependent data set. We added a little noise with low amplitude to make it more realistic, hence the call to `randn`. In Command 24, we defined the model through an Octave function and we then set our initial guess in Command 25. This guess needs to be reasonable—how far from the true parameter value it can be depends on the model and the data. Sometimes the algorithm may not converge, and you must then use another (a better) initial guess. It should be clear what Commands 26 to 28 do.

Always have a valid reason to use a specific functional form as a model. A fit of a model with too many or too few free parameters will fail to converge to the data set even though the function has the correct form.



Note that if you follow the commands above, your fitted parameters may be a bit different and the number of iterations may be six plus or minus one depending on what random numbers your machine generated.

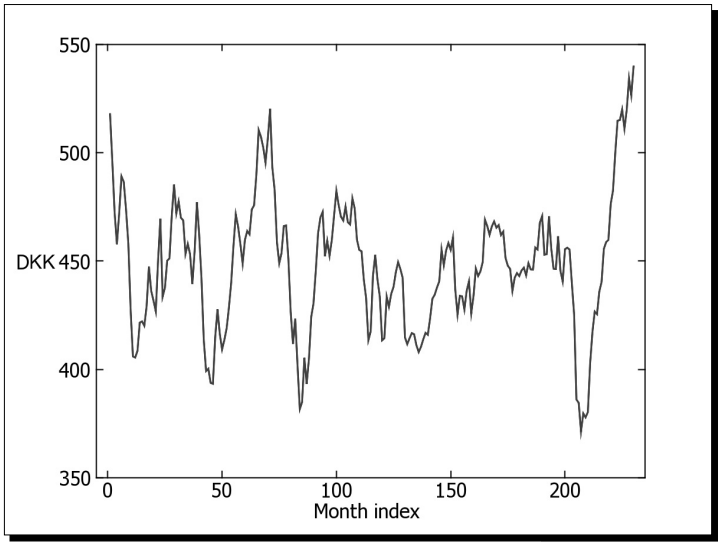
Have a go hero – calculating the deviation of the Monte Carlo integrator

In Command 4, we calculated the standard deviation of the Monte Carlo integration scheme for the integral $\int_0^{\pi} \sin(x)\,dx$. Here we used 1000 Monte Carlo loops. Calculate the standard deviation when the number of loops is, say 50, 70, … , 1990, 2010. You can still use 1000 samples as the sample size. Determine the functional form of the standard deviation as a function of Monte Carlo loops.

Fourier analysis

The figure below shows the monthly average exchange rate between the Australian dollar and Danish kroner starting from January 1, 1991, and ending on March 2010. The y axis is the price in kroners for 100 dollars and the x axis is the month index after Janary 1, 1991. It is clearly seen that there exists an underlying oscillatory behavior. This could motivate us to buy dollars paying with kroners whenever the exchange rate is low, and then make an exchange back to kroners whenever the rate is high, thereby making a net earning. The exchange rate data are a bit hard to read because of the superimposed noise and by the fact that the periodicity is not strictly a constant. It would therefore be desirable to have a way to extract the main periods or frequencies before we make large investments. A Fourier (or spectral) analysis can help you with this.

Before we analyse the exchange rate data, however, we should try to get an understanding of how the Fourier transform works in Octave using a simple example.



The Fourier transform

The fundamental principle behind Fourier analysis is the Fourier transform. If the function f is dependent on time t and is integrable, the Fourier transform of f is defined as:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\, e^{i\omega t} dt. \tag{7.13}$$

Note that the function F is a function of the angular frequency, ω, and that it is in general a complex valued function. In general, f is also a complex valued function, but we shall not consider this case here.

Now, suppose we have a set of N discrete data points $f_1, f_2, \ldots, f_N$ that are sampled at constant sampling interval Δt at times $t_1, t_2, \ldots, t_N$. In this situation, the integral form of the Fourier transform given above can be approximated by the discrete Fourier transform (DFT):

$$F_n = \sum_{k=1}^{N} f_k e^{i\omega_n t_k} \Delta t. \tag{7.14}$$

I will explain the index n in the following matter. According to the sampling theorem (or Nyquist-Shannon theorem), we cannot choose the frequencies ω_n at random in order to perform a correct Fourier analysis, but they are given by:

$$\omega_n = \frac{2\pi n}{N\Delta t}, \quad n = -\frac{N}{2}, \ldots, \frac{N}{2}. \tag{7.15}$$

This means that to retrieve large frequency modes, we must have sufficient frequent sampling which makes good sense. Substituting Equation (7.15) into Equation (7.14) and assuming $t_k = (k-1)\,\Delta t$ such that $t_1 = 0$, we arrive at:

$$F_n = \Delta t \sum_{k=1}^{N} f_k e^{i\pi n\,(k-1)/N}. \tag{7.16}$$

In principle we could just code Equation (7.16) directly. Octave even supports complex numbers, so it could be done in a few lines. As it stands, however, this will lead to what is called an order N^2 algorithm which means that the number of operations we need to perform is proportional to N^2 and thus the execution time increases dramatically for large sample sizes[2]. For a large data set, it would therefore not be a clever approach. Out of pure curiosity, we will try it out later in an exercise.

Time for action – using the fft function

1. Let us try to Fourier transform the function:

$$f(t) = \sin(2t) + 2\sin(5t), \tag{7.17}$$

where $t \in [0;\, 2\pi]$ using 150 data points. This function is characterized by two different frequencies (or modes) that will show in the Fourier transformation as two distinct peaks.

[2] *This is also referred to as an O(N^2) algorithm. O is pronounced 'big-O'.*

2. To generate the data we use:

```
octave:29> N=150; t = linspace(0,2*pi, N);
octave:30> f = sin(2*t) + 2*sin(5*t);
```

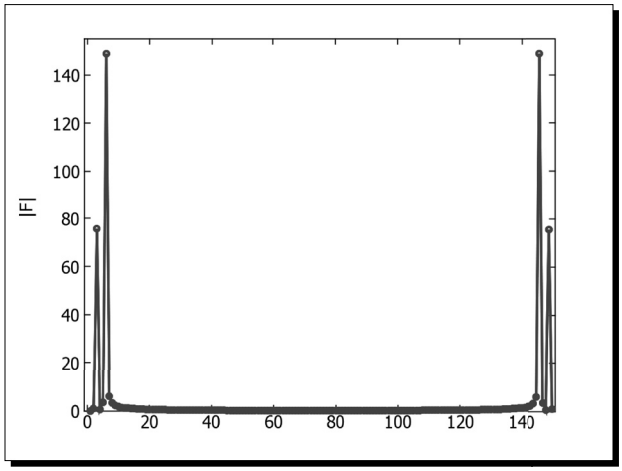
3. Then we simply transform those data via:

```
octave:31> F = fft(f);
```

4. The complex vector `F` is not really of much information itself, so we often display the absolute value (or magnitude) of the elements in the array:

```
octave:32> plot(abs(F), "o-")
```

which produces the plot seen in the figure below:



What just happened?

In Commands 29 and 30, we generated the data set. I strongly recommend that you try to plot `f` versus `t` to see the function that we are transforming. We then call `fft` and plot the absolute value of the transformed data.

Things look a bit odd though. We expected two peaks representing the two different modes in Equation (7.17), but the figure above shows four peaks that seem like mirror images of each other. This actually makes sense. From Equation (7.15), we see that the frequencies go from:

$$-\frac{\pi}{\Delta t} \leq \omega_n \leq \frac{\pi}{\Delta t}. \tag{7.18}$$

This explains that there is a mirroring taking place because $|F(-\omega)| = |F(\omega)|$. Note, the maximum frequency $\pi / \Delta t$ is denoted the Nyquist frequency ω_N.

`fft` organizes the output in two halves. The first half corresponds to the frequencies from 0 to $\pi/\Delta t$, the second half from $-\pi/\Delta t$ to 0. This is easily fixed with `fftshift` such that the values of `F` correspond to frequencies in the interval $-\pi/\Delta t \leq \omega_n \leq \pi/\Delta t$.

```
octave:35> F = fftshift(F);
```

Before we plot the magnitude of `F` again, we should also generate the frequencies we plot against. First, the sampling interval:

```
octave:34> dt = t(2);
```

then we define n in Equation (7.15) via:

```
octave:35> n=[-N/2:1:N/2];
```

But hold on, this array has length $N+1$! To fix this problem, we can simply remove the last and highest frequency point:

```
octave:36> n(N+1)=[];
```

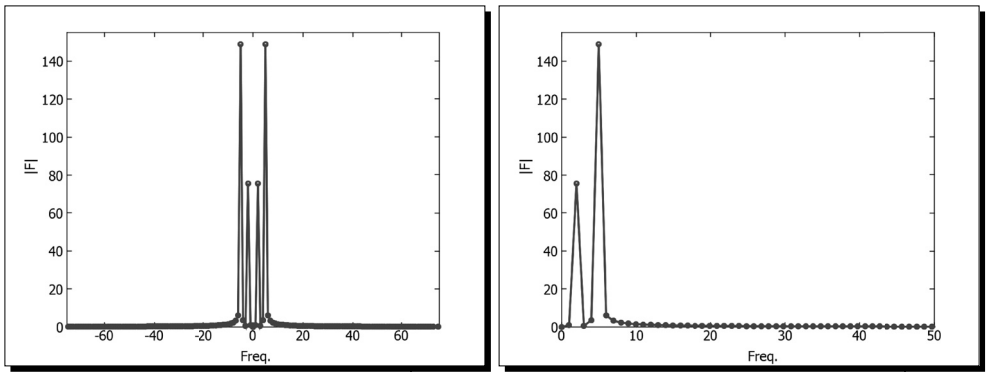
According to Equation (7.15), the frequency is calculated as:

```
octave:37> freq = 2*pi*n./(N*dt);
```

We are now ready to plot the magnitude of `F` as a function of frequency—this is sometimes referred to as the magnitude spectrum:

```
octave:38> plot(freq, abs(F), 'o-')
```

The result from Command 38 is shown in the left-hand side figure below. In the right-hand side figure, only positive frequencies are shown. We now see that the two frequencies appear where we expect. The fact that the higher frequency component also has larger amplitude is also captured in the magnitude spectrum.



In the example above, we transformed a data set composed of 150 points. If you change this number, you will see that the magnitude of `F` will change too: the more the points, the larger the magnitude. The reason for this is that `fft` excludes the multiplication with Δt, see Equation (7.16). Thus to get a spectrum that is independent of the number of data points, we can simply multiply the output from `fft` with Δt.

Changing the number of data points also means changing the Nyquist frequency, ω_N, according to Equation (7.18). It is very important to remember when you perform fast Fourier transforms that the frequency is not a freely variable parameter, but is determined by the sample size and sampling interval. Also, the Nyquist frequency must be sufficiently large for the Fourier transform to capture all the frequencies embedded in the data. In fact, if ω_N is not large enough (that is, if the sampling interval is too large), the high-end frequencies will not show up correctly in the spectrum. This is known as aliasing. To be sure that there is no aliasing problem, the spectrum for the largest frequencies should be zero. We shall soon see that it is not always possible to have the necessary sample rate.

Fourier analysis of currency exchange rate

Let us return to the problem of the currency investment. We now know how to perform a Fourier analysis of data using Octave's `fft` function, so should we just dive into it? Well, we are not quite there yet. If we simply made a Fourier transform of the data, there will be a large peak at very low frequencies. This peak comes from the fact that the average exchange rate is not zero and that there exists an overall trend of increasing price. If we want to make shorter term investments, we are mainly interested in knowing about the higher frequency behavior and we should therefore remove these trends.

Time for action – analysing the exchange rate

1. Assume that we have loaded the currency exchange rate data into a variable `curr` and that we have 230 data points. The month index is given by:

```
octave:39> m_index = [0:229];
```

2. The increasing trend is given by the end-points in the data:

```
octave:40> a = (curr(230)-curr(1))/229; b = curr(1);
```

3. Subtracting the trend from the data:

```
octave:41> curr1 = curr - a*m_index - b;
```

4. To ensure a data set with zero-mean, we use:

```
octave:42> curr2 = curr1-mean(curr1);
```

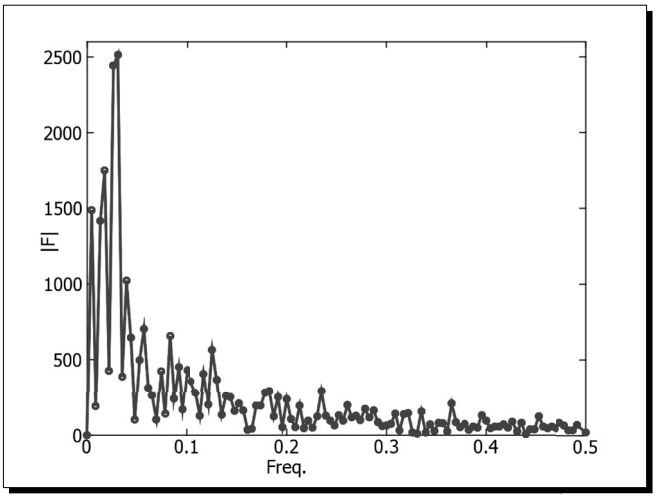
5. We can then perform a Fourier analysis on the "trend-free" data using the ordinary frequency $f = \omega / 2\pi$:

```
octave:43> N=230; n=[-N/2:N/2]; n(N+1) = []; freq = n./N;

octave:44> acurr = abs( fftshift( fft(curr2) ) );
```

6. To plot the result, we use:

```
octave:45> plot(freq, acurr)
```

The result is shown below for positive frequencies. Note that we may have an aliasing problem!



7. Now the ordinary frequency of the main mode is:

```
octave:46> i = find( acurr==max(accur) ); freq(i)

ans = 0.030434
```

What just happened?

In Command 39, we define the month index to go from zero (first month of 1991) to 229 (the second month in 2010). To remove the overall increasing trend, we simply subtract the line that goes through the end-points in the data set, here represented via the linear function $y = ax + b$, where x is the month index. a is calculated to 0.096. This will lead to a new data set where the end points are zero, but all others will be negative, which is also unwanted since this gives a non-zero mean which also shows in the Fourier spectrum. This bias is then removed in Command 42. It should be clear what Commands 43 to 45 do, but note that we use the ordinary frequency rather than the angular frequency.

The main frequency mode is computed in Command 46 by using the `find` function. This frequency corresponds to a period $T = 1/f$, that is, if we sell and reinvest every sixteen and a half months we could expect a net earning.

It is worth noting that since we have a possible aliasing problem, we cannot make any conclusions about the high frequency modes.

To sum up, the Fourier analysis using `fft` can be done in the following six steps:

1. Consider if you should remove any trends in the data set and do so if needed.
2. Fourier transform your data using `fft`.
3. Rearrange the output via `fftshift`.
4. Generate the sample frequency. Remember to remove the last high-end frequency component.
5. Plot the spectrum for positive frequencies.
6. Is the spectrum for high frequencies zero? If not, be aware of any aliasing problems.

Inverse Fourier transform and data smoothing

It would be very useful to somehow remove the "noise" that we see in the currency plot because it hides the underlying characteristics of the data and it brings no useful information. Such noise shows up in the high frequency end in the spectrum. Can we use this for anything? The answer is yes and the idea goes as follows: if we can somehow set the high frequencies to zero and then make an inverse Fourier transform, we should still see all the slowly varying modes, but no high frequency ones. The only thing we should be careful about is how we remove the unwanted frequencies—Octave takes care of the inverse Fourier transform.

Analogous to Equation (7.14), the discrete inverse Fourier transform is given as:

$$f_k = \frac{1}{2\pi}\sum_{n=1}^{N} F_n e^{-i\omega_n t_k}\Delta\omega. \qquad (7.19)$$

Octave's inverse fast Fourier transform function `ifft` has the same syntax as `fft`. In its simplest form:

```
f = ifft(F)
```

Before we try it out, we should briefly discuss how to pick out the low frequencies in the spectrum.

The Butterworth filter

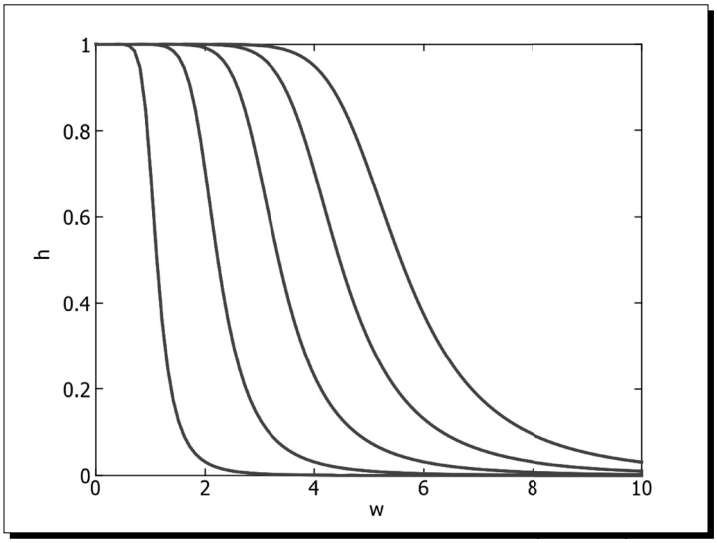
The following function is called a Butterworth function of order n:

$$h(\omega) = \sqrt{\frac{1}{1-(\omega/\omega_c)^{2n}}}, \tag{7.20}$$

where ω_c is called the critical frequency. A fifth order Butterworth function is plotted below for different critical frequencies in the range from 1 to 5: the leftmost graph with the steepest descent is for $\omega_c = 1$, and the rightmost is for $\omega_c = 5$. Note that the Butterworth function is 1 for $\omega = 0$ and goes smoothly to zero for large ω. Now, if we multiply the Butterworth function with the Fourier transform of the data, we can pick out all the low frequencies and set the high frequencies to zero. This is exactly what we want. This is called a low pass filtering.

You can apply any other type of filter, however, you should be extremely careful with the particular choice. The Butterworth filter is a smooth varying function that prevents so-called edge effects, effects that show up in step-type filters.

You can also apply the Butterworth filter for medium and high band filtering by simple function shifts.



In Octave you can also perform smoothing directly in the time domain using piecewise polynomial fits – this will usually be a low pass filter. Type `help spline` to see how.

Time for action – applying a low pass filter

1. Continuing from Commands 39-46, let us apply a tenth order Butterworth filter with $\omega_c = 0.1$ to smooth the "trend-less" data set stored in curr2, see Command 42:

```
octave:47> b_order = 10; w_c = 0.1;
```

2. The Butterworth function is simply given as:

```
octave:48> w = sqrt(1./(1 + (freq./wc).^(2*b_order)));
```

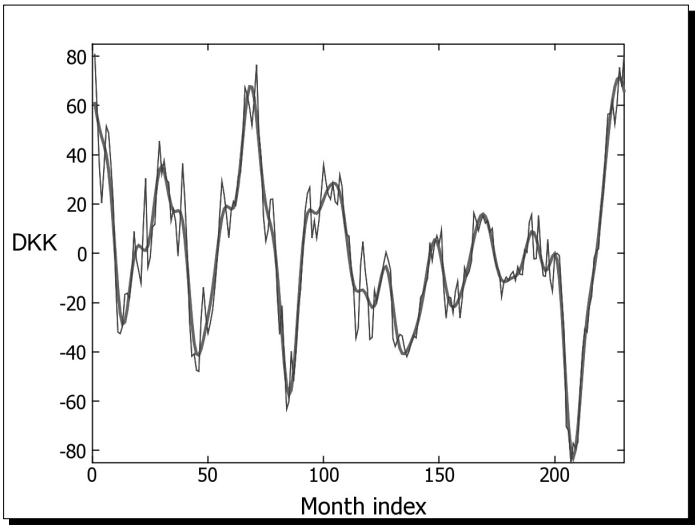
3. We can choose to rearrange either the Butterworth window function or the data before multiplying them together —we will rearrange the window function:

```
octave:49> w = fftshift(w);
```

4. In order to apply the filter to the Fourier transform of `curr2` and then perform the inverse Fourier transform, we use:

```
octave:50> plot(m_index, ifft(fft(curr2).*w), 'r', \
> m_index, curr2, 'b')
```

5. The result is given below, where the smoothed data is given by the red curve:



What just happened?

In Command 47, we specified the critical frequency and the Butterworth order parameter. There is no recipe on how to choose the particular parameters, but it is always a good idea to plot the resulting Butterworth filter to see if it actually picks out the frequencies we are after and removes the ones we wish to suppress. It is also important to be sure that the Butterworth function is arranged in the same manner as the output from `fft`. We do this in Command 49. In Command 50, we then apply the filter to the Fourier transformed data, inverse Fourier transform the filtered data back into the time domain, and plot it together with the unfiltered data.

Notice that the Butterworth filter and many other types of filters are available from the **signal** toolbox.

Have a go hero – implementing your own Fourier transform function

In this exercise, you are to code your own Fourier transform function. The syntax should be:

```
[F freq] = sft(f, dt)
```

where `F` is the Fourier transformed data, `freq` is a vector with the corresponding frequencies, `f` is the input data, and `dt` is the constant sampling interval. The function must use Equations (7.14) and (7.15) and not `fft`. `sft` should be vectorized to some degree.

Test `sft` with the function given in Equation (7.17) and compare with the corresponding figures in the section. By the way, why the name `sft`?

Summary

In this chapter we learned:

- About the file formats that Octave can load.
- How to calculate different sample moments in Octave for simple descriptive statistics.
- How to compare data sets using `cor` and `t_test`.
- How to use `polyfit` and `leasqr` to fit different functions to data.
- About the fast Fourier transform function `fft`.
- How to use the inverse Fourier transform function `ifft`.

We will now proceed with the last chapter where you will learn about optimization techniques and dynamically linked functions.

Where to buy this book

You can buy GNU Octave Beginner's Guide from the Packt Publishing website:
`http://www.packtpub.com/gnu-octave-beginners-guide/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com