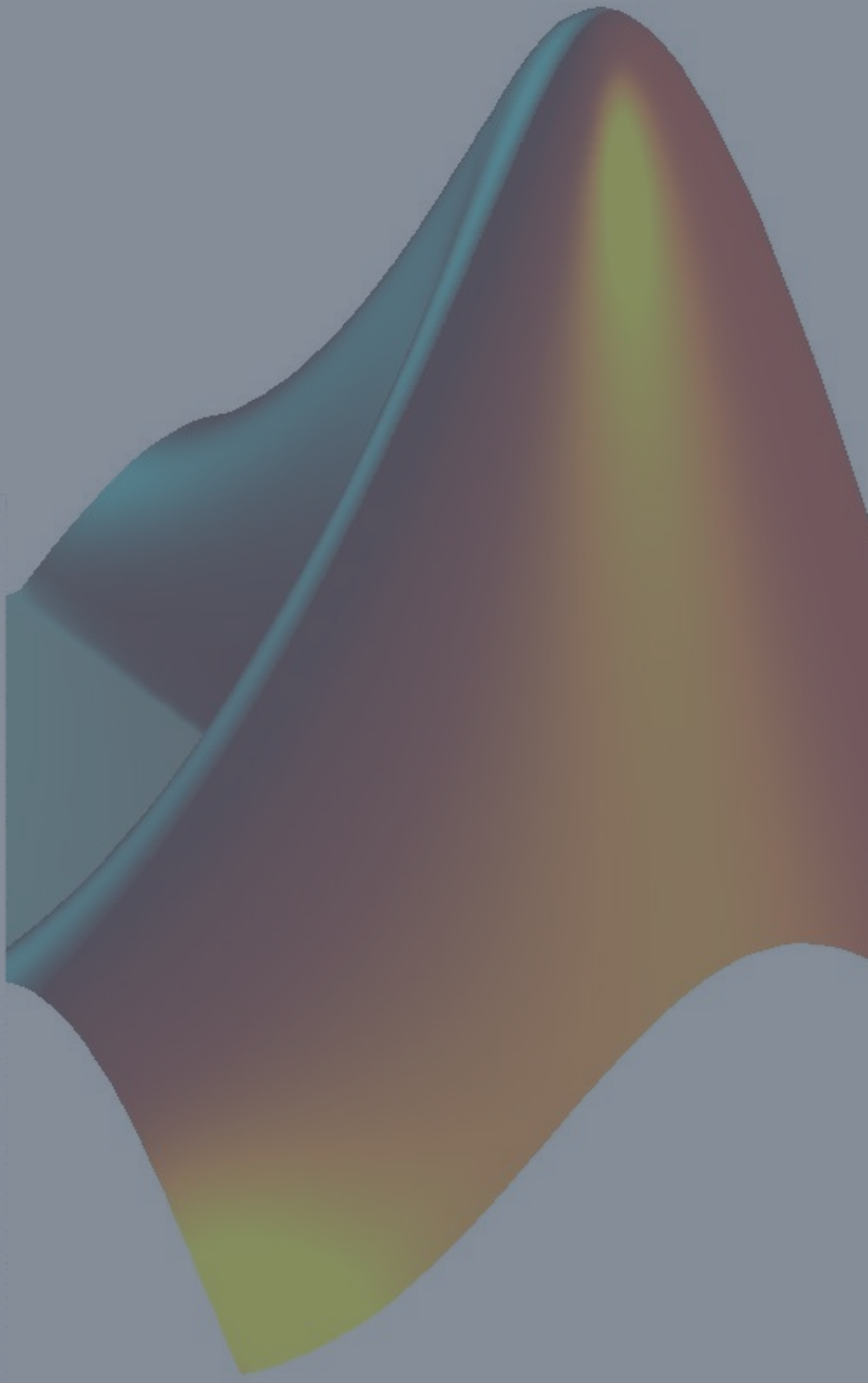
Queen Mary
University of London

QMUL2015
GRADFEST



MATLAB TRAINING I: DATA PROCESSING

Contents

List of Figures

June, 2015

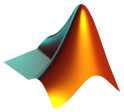
This material was created for the GradFest 2015, MATLAB® Data Processing session.

For enquires regarding its content, contact **Michail Palaiokostas** at **m.palaiokostas (at) gmail.com** .

School of Engineering and Materials Science
Queen Mary, University of London
Mile End Road
E1 4NS
London
UK

1 What is MATLAB®?

MATLAB® is a numerical computing commercial software, that can be used either interactively through the graphical user interface (GUI) or automatically by executing special script files. The MATLAB® platform, consists of many toolboxes that provide the user with a plethora of tools for different applications. Among the scientific community, MATLAB® is a commonly used software, therefore, many toolboxes have been created to cover almost any discipline.



MATLAB® logo

> ***Alternatives:*** *Although MATLAB® is one of the most commonly used tools for data processing, is not the only one. In fact, there are many free, open-source alternatives, with the similar capabilities and large supporting communities. GNU Octave*[1] *and Scilab*[2] *are the open-source "siblings" of MATLAB®, while Python*[3] *is an easy-to-learn all-purpose programming language, that is often used by scientists for data analysis. Finally, R*[4] *is one of the best programming languages for statistics and data analysis.*

2 What is the purpose of this tutorial?

This tutorial is aimed at all scientists and engineers, that want to see how MATLAB® works, overcome the initial fear and eventually implement it in the research process. In particular, this tutorial will cover the first steps of a basic statistical analysis of research data. The data acquisition method is irrelevant, as long as, it is in an accessible file format (.xlsx, .ods, .dat etc.) and arranged in a consistent manner (rows/columns, separated by tabs/spaces/-commas etc.).

While working interactively is sometimes useful, the true power of scripting languages comes from the ability to "record" the methodology and use it regardless of the specific data. Therefore, this tutorial consists of two parts and eight scripting exercises, of progressive difficulty, that introduce several parts of a typical data processing workflow. The first part (seven exercises) will go through the analysis of a single data file, while in the second part (last exercise), will demonstrate the automatic analysis of ten data files. In particular, the exercises show how to:

1. create the first script
2. import data from files
3. plot
4. obtain basic statistics
5. filter data
6. fit a function to data
7. save analysis to file
8. automate for a big number of files

[1] `http://www.gnu.org/software/octave/`
[2] `http://www.scilab.org/`
[3] `https://www.python.org/`
[4] `http://cran.r-project.org/`

3 General script structure

In order to automate the tasks, the user writes the commands in special script files (ending in .m) and then MATLAB® interprets and executes them from top to bottom, line by line. Scripts usually follow the following structure:

```
Initialising script
Setting parameters and constants
Reading input
Core analysis part
Saving analysis results
Closing script
```

4 Preparation

The rest of the tutorial is based on pre-built datasets and a specific folder structure. To be able to continue with the tutorial, its recommended to do the following:

1. Download the file **GradFest2015-MATLAB-1.zip** from: `http://tinyurl.com/GF2015-MATLAB`
2. Unzip the downloaded file and copy and paste the folder to the Desktop.
3. Do the exercises of the tutorial inside this folder.

Alternatively, execute the code in the appendix, to create new datasets.

5 Single file analysis

5.1 Exercise: Creating your first script

Before starting the analysis, we have to create the first script. As a first task, we will create a script that plots user's name.

- Open MATLAB® and click the button New > Script (This will open a new window, called "Editor")
- In Editor, write the following:

```
%% INITIALISING SCRIPT
  clc        % Clears the command window
  clear      % Clears the workspace (vectors etc.)
  close all  % Closes all active figures

%% SETTING PARAMETERS
  disp('DATA ANALYSIS SCRIPT')
  username='Iwannabeadoctor'; % Set your name between the quotes
  output=['Researcher: ', username]; % Creates a string vector
  disp(output)                    % Prints the name of the user
  disp('Analysis starting...')
```

The symbol % initiates a comment, therefore MATLAB® ignores it.

- Save the file by clicking the button Save and selecting the main folder of the downloaded file and a filename (e.g.single.m).
- To execute the script, click the button Run. In the Command Window, you should see the name.
- Well done for completing your first MATLAB® script!

5.2 Exercise: Importing data to your script

Most of the times, the outcome of a simulation or an experiment, is a large number of data, stored in files. These files can be in any format, but as long as they are in a tabular format, their importing in MATLAB® is very easy. In this exercise, we will import 1250 lines of data from a file containing a noisy sinusoidal signal that was recorded over a few hundred milliseconds.

- Continue on the same file, created before.
- In the Editor, add the lines, save and execute:

```
%% READING INPUT
  cd('datasets')            % Change to dataset directory
  filename='tutorial-1.dat'; % Name of the file to import
  data=dlmread(filename);   % Save the contents to the matrix "data"
  cd('..')                  % Return to previous directory
```

- In this case, we imported data from a text delimited file. If the data was stored in an spreadsheet, we could use the command:

  ```
  xlsread()
  ```

- That's it! Your data are now loaded and ready to be processed!

> *Sometimes, data files are too big to be processed "all-at-once". This is an issue, when the size of the file is equal or larger than the size of the free RAM memory in the machine. In order to circumvent the size limitations, there are alternative ways of reading data, however they are out of the scope of this introductory tutorial. However, keeping in mind these limitations, is very important during the designing of any data processing script.*

5.3 Exercise: Plotting

One of the most common tasks for a researcher, is the plotting of data. MATLAB®, gives to researchers the opportunity to change a plethora of parameters in their plots, by setting them once in a script file and then just applying them automatically for the rest of the files. In this exercise, we will plot the unprocessed data that we imported in the previous exercise and save it in a high quality figure.

- Continue on the same file, created before.
- In the Editor, add the following lines, save and execute:

```
%% PLOTTING ... (see the help page of "plot" for details!)

  x=data(:,1);   % Setting x-axis data
  y=data(:,2);   % Setting y-axis data

  % ... a simple figure
  figure(1)
  plot(x,y)
```

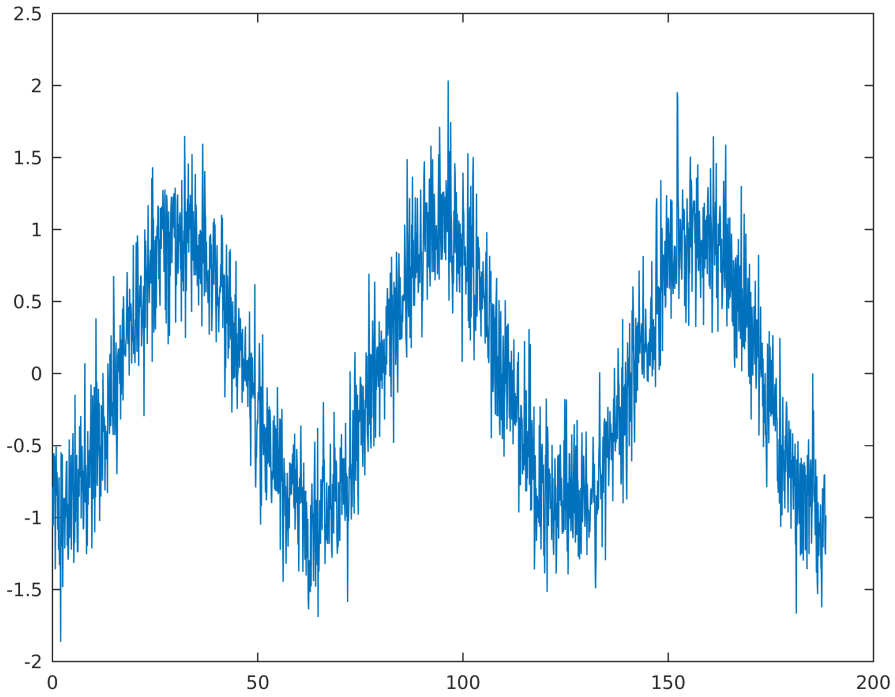


Figure 1: A simple figure.

- The figure on the screen must be similar to Figure 1, which shows the data, but is not yet "publishable". In order to improve it, we should add the following lines to our code:

```
% ... a proper figure
figure(2)
plot(x,y,'Color','r','Marker','o','Linestyle','--')
hold on                          % Add the following settings
title('My first MATLAB plot!') % Set the title
xlabel('Time [ms]')              % Set x-axis label
ylabel('Signal values')          % Set y-axis label
xlim([min(x),max(x)])            % Manually set x-axis limits
legend('File 1')                 % Set the legend
hold off                         % Stop adding settings
```

- The figure on the screen must be similar to Figure 2.
- In order to complete the exercise, we should save the produced figure in a high quality format.To achieve this, we should add the following lines to our code:

```
% Save figure 2 as a high quality .png
figurename='plot1';
resolution='-r300';       % Resolution in dots per inch
cd('figures')
print('-f2',figurename,'-dpng',resolution)
cd('..')
```

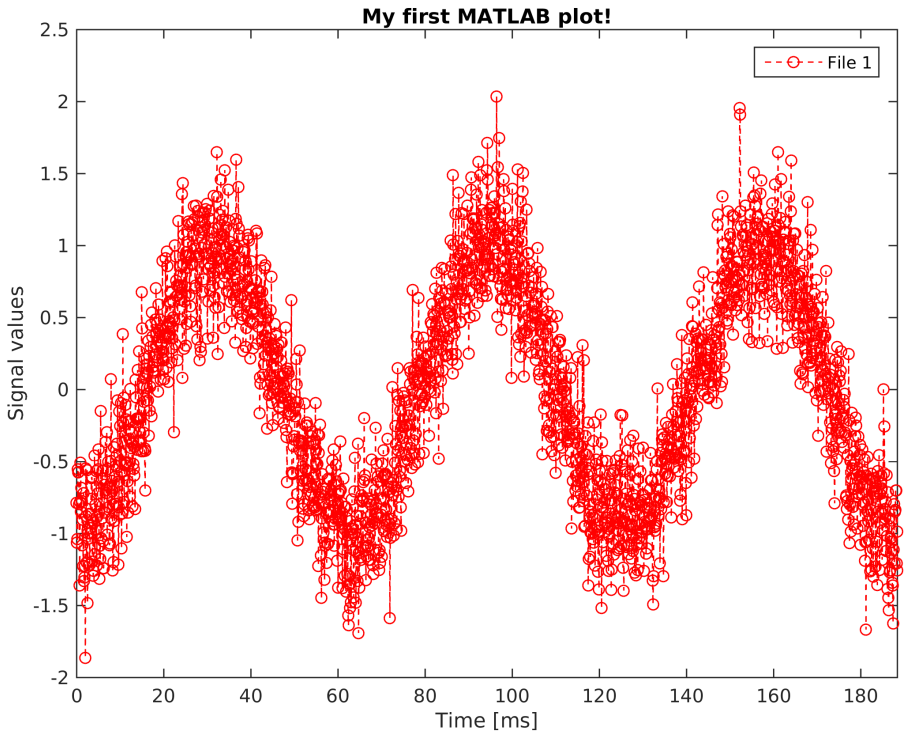


Figure 2: A better figure.

- The working directory should have a new .png file called "plot1.png"
- We have now successfully produced our first plots with MATLAB®!

5.4 Exercise: Obtaining basic statistics

Obtaining the statistics of the data, is usually the main reason of a data processing code. While there is a broad range of needed statistics, depending on the application and discipline, there is a basic set of properties, that gives a satisfactory overview of the examined data. In this exercise, we will see how to obtain the mean and standard deviation of our data, as well as, some more useful numbers.

- Continue on the same file, created before.
- In the Editor, add the following lines, save and execute:

```
%% STATISTICAL ANALYSIS...

  % ...calculate the mean
  average=mean(data(:,2));
  output=['For ',filename,' the mean is: ',num2str(average)];
  disp(output)

  % ...calculate the standard deviation
  stdev=std(data(:,2));
  output=['For ',filename,' the standard deviation is:
      ',num2str(stdev)];
  disp(output)
```

*The command **num2str** converts a number to a character so that it can be printed together with text.*

- In these lines, we do not manually calculate the mean and standard deviation (std). Instead, we use a MATLAB® build-in function that does the work for us.
- We will use similar build-in functions to calculate the minimum and maximum values of the signal. The result will be printed on the screen. Write the following into your code:

```
% ... calculate the minimum and maximum
minValue=min(data(:,2));
output=['For ',filename,' the minimum value is:
    ',num2str(minValue)];
disp(output)
maxValue=max(data(:,2));
output=['For ',filename,' the maximum value is:
    ',num2str(maxValue)];
disp(output)
```

- Finally, we will compute the percentage of data, that is smaller and larger than the average. Write the following into your code:

```
% ... percentage of data that are larger or smaller than the
    average
dataRows=size(data,1);    % Returns the number of measurements
counterSmaller=0;
counterLarger=0;
for i = 1:dataRows
  if data(i,2)<=average
      counterSmaller=counterSmaller+1;
  else
      counterLarger=counterLarger+1;
  end
end
percSmaller=(counterSmaller/dataRows)*100;
percLarger=(counterLarger/dataRows)*100;
output=['For ',filename,', ',num2str(percLarger),'% is higher than
    the average and ',num2str(percSmaller),'% is lower.'];
disp(output)
```

- In the last code snippet, we used two variables as counters and a for-loop to scan all the data values. In every iteration of the loop, we compared the current value, with the value of the average that we found before. Based on the comparison, we added a +1 to the respective counter. In the end, we computed the final percentages by dividing the counters, with the total number of values (*dataRows*).
- Congratulations on completing the introduction to basic statistics with MATLAB®!

5.5 Exercise: Filtering data

Many times it is important to filter experimental and simulation data, according to a specific criterion. In this exercise, we will see how to isolate all the values that are less than or equal to the average and keep the rest.

- Continue on the same file, created before.
- In the Editor, add the following lines, save and execute:

```
%% FILTERING DATA...

% ... create a new dataset with only the larger than average values
larger=zeros(counterLarger,2); % Create an empty matrix to store
    the filtered values
counterSide=1;
for i=1:dataRows
    if data(i,2)>average
        larger(counterSide,1)=data(i,1);
        larger(counterSide,2)=data(i,2);
        counterSide=counterSide+1;
    end
end
```

- In the above code, we essentially created a new matrix, to store the filtered data.
- Also, we initialized this matrix (*larger*), by filling it with zeros and by setting its size, to be equal to the *counterLarger*, that we found before.
- Since we "transferred" data between matrices of different size, we used a second counter (*counterSide*) in order to save the values in the correct position of matrix *larger*. For example, if the for-loop encounters for the first time a larger value in the 3rd position of matrix *data*, then this should be saved in the 1st position of matrix *larger*:

  ```
  1) If for i=3, the if-condition becomes True, so:
  2) data[3] -- > larger[1]
  3) Increase the counter of larger by 1 to be ready for the next step
  ```

- To demonstrate what exactly happened above, write the following into your code and execute:

```
% Plot only the filtered data
figure(3)
x=larger(:,1);
y=larger(:,2);
plot(x,y,'Marker','*','Color','black')
hold on
title('Filtered signal')
xlabel('Time [ms]')
ylabel('Signal values')
legend('File 1')
hold off
```

- Figure 3 shows the final result of the filtering process (the new dataset with only the larger than average values).
- Well done for completing the filtering data exercise!

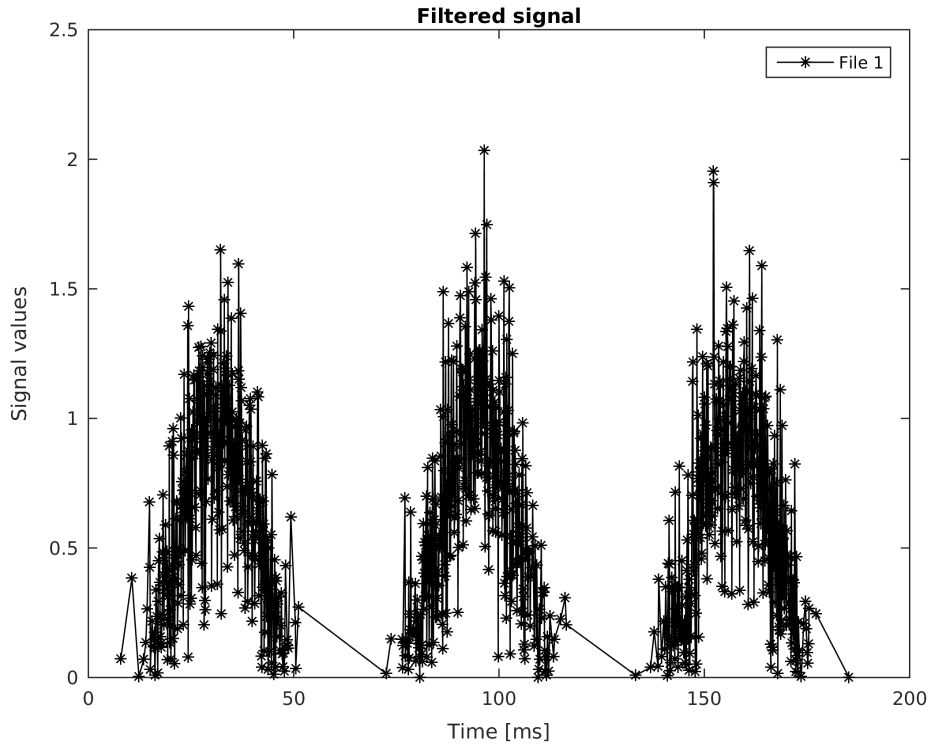


Figure 3: The result of filtering the data

5.6 Exercise: Fitting data

In this exercise we will perform another very common task of data analysis; fitting a function. MATLAB® has a very powerful toolbox to perform interactive fitting but in this exercise we will see how to do this with a script.

Sometimes it is very useful to do a visual inspection of the fit before automating its process. To initiate the fitting toolbox, type ***cftool*** *in the command window.*

- Continue on the same file, created before.
- In the Editor, add the following lines, save and execute:

```
%% FITTING A FUNCTION TO NON-LINEAR DATA

  % Do the fitting
  xfit=data(:,1);
  yfit=data(:,2);
  [fitresult, gof] = fit( xfit, yfit, 'sin1');

  % Plot the fit versus original data
  figure(4)
  x=xfit;
  y=yfit;
  plot(fitresult,xfit,yfit)
  hold on
  title('Fit vs Original data')
  xlabel('Time [ms]')
  ylabel('Signal values')
  legend('File 1', 'Fitted function');
  hold off

  % Save the plot to a .png file
  figurename='plot2';
```

```
resolution='-r300';
cd('figures')
print('-f4',figurename,'-dpng',resolution)
cd('..')
```

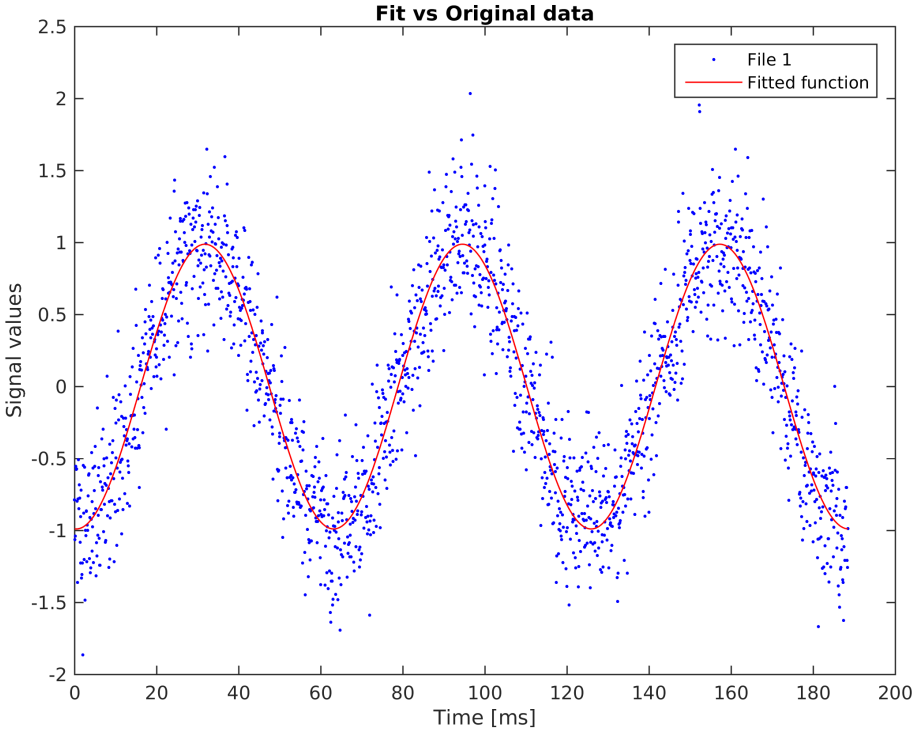


Figure 4: Fitting of a function

- The above code can be broken into three parts. In the first we performed the fitting, then plotted the fit against the original data and then saved the plot in a high quality .png file.
- To perform the fitting, we had to define the x and y sets, as well as, the function to fit. In this case, we selected a single sinusoidal because our data had an obvious sinusoidal form.
- Figure 4 shows how the fit performed against our noisy data. Pretty good!

The whole code of these exercises can be found in the folder ***solutions****, in the script* ***SolutionSingle.m***

5.7 Exercise: Saving analysis results and Closing

In this final exercise of part A, we will add a saving capability to our code and end it with a message. Obviously, after performing the previous analysis, we want to save the results in a summary file. To achieve this, we have to add three sections into the code:

1. In the beginning, opening of a summary file to save the results.
2. After the analysis, saving of the results to a cell array and to the file.
3. In the end, closing of the summary file and of the code.

- Continue on the same file, created before.

- In the Editor, after the section "%%SETTING PARAMETERS" and before "%%READING INPUT", write the following (do not execute yet!):

```
% PREPARE THE SUMMARY FILE TO SAVE THE RESULTS
finalresult=cell(1,7);      % Define the size of the file
    (rows,columns)
filename='Summary.txt';      % Define filename
fid = fopen(filename, 'wt'); % Open the file in "write" state
```

- In the above code, we defined a new data type, called cell, which allowed the saving of characters and numbers in the same array. The number of rows (1), is equal to the number of data files and the number of columns (7), is equal to the number of results to save.
- Also, we opened the file **Summary.txt** to be ready to accept input.
- To continue, in the Editor, after the section "%% FITTING A FUNCTION TO NON-LINEAR DATA", write the following (do not execute yet!):

```
%% COLLECTING ANALYSIS RESULTS
  finalresult{1,1}=filename;
  finalresult{1,2}=num2str(average);
  finalresult{1,3}=num2str(stdev);
  finalresult{1,4}=num2str(minValue);
  finalresult{1,5}=num2str(maxValue);
  finalresult{1,6}=num2str(percLarger);
  finalresult{1,7}=num2str(percSmaller);
```

- In the above code, we collected all the results to the cell array so that we can easily save it in a later stage in a file, with a specific format.
- To save the cell in the file,close it and display an ending message for the analysis, add the following in the Editor and execute:

```
%% CLOSING SCRIPT
  % Save the summary array to a file
  for i=1:size(finalresult,1)
    format='%s %s %s %s %s %s %s\r\n';
    fprintf(fid, format, finalresult{i,:});
  end
  fclose(fid);  % Close the file
  disp('Analysis completed!')
```

- That was the final exercise in the first part of the tutorial. Congratulations!

6 Multiple files analysis

Usually during research, a lot of data files are created from the same or similar experiment/simulation. In the second part of the tutorial, we will see how to automate the process of the first part, for a large number of files.

- Open the same script as before and save it as **multiple.m**.
- The analysis part will remain the same. However, in the new script we will add a for-loop to repeat the analysis for each file. The number of loops will be equal to the number of data files in the *datasets* folder.
- In the new script, before the section "%%READING INPUT", add the following (do not execute yet!):

```
%% LOOP OVER EACH DATA FILE
  for f=1:10 % Begin the loop
```

- Also, before the section "%%CLOSING SCRIPT", add the following (do not execute yet!):

```
  end % Close the loop
```

- Apart from the loop, it is necessary to change also a few more lines so that in every iteration the appropriate file is read and the plotted figures are not overwritten. The changes are shown in Table 1.

Table 1: Necessary changes, to make the script compatible with the for-loop

Section	Old	New
PREPARE	finalresult=cell(1,7);	finalresult=cell(10,7);
READING	filename='tutorial-1.dat';	filename=strcat('tutorial-',num2str(f),'.dat');
PLOTTING	figurename='plot1';	figurename=strcat('f',num2str(f),'-plot1');
FITTING	figurename='plot2';	figurename=strcat('f',num2str(f),'-plot2');
	finalresult{1,1}=filename;	finalresult{f,1}=filename;
	finalresult{1,2}=num2str(average);	finalresult{f,2}=num2str(average);
	finalresult{1,3}=num2str(stdev);	finalresult{f,3}=num2str(stdev);
SAVING	finalresult{1,4}=num2str(minValue);	finalresult{f,4}=num2str(minValue);
	finalresult{1,5}=num2str(maxValue);	finalresult{f,5}=num2str(maxValue);
	finalresult{1,6}=num2str(percLarger);	finalresult{f,6}=num2str(percLarger);
	finalresult{1,7}=num2str(percSmaller);	finalresult{f,7}=num2str(percSmaller);

- By making the changes and executing the code, 10 different sets of plots should exist in the figures folder. Also, the **Summary.txt** file should contain the results of the 10 different files.
- That was the last exercise of the tutorial. Congratulations and happy data processing!

The whole code can be found in the folder ***solutions****, in the script* ***SolutionMultiple.m***

A Appendix: Code to create the datasets

```
clc
clear
close all
for f=1:10
    x = (-3*pi:.01:3*pi);
    noise = 0.3.*randn(1,size(x,2));
    data = cos(x) + noise;
    length=size(data,2);
    dt=0;
    time=zeros(length,1);
    for i=1:length,
        time(i,1)=dt;
        dt=dt+0.1;
    end
    array(:,1)=time();
    array(:,2)=data();
    filename=strcat('tutorial-',num2str(f),'.dat');
    delete(filename)
    dlmwrite(filename,array,'\t')
end
```

The whole code can be found in the folder ***solutions****, in the script* ***AppendixData-Generation.m***