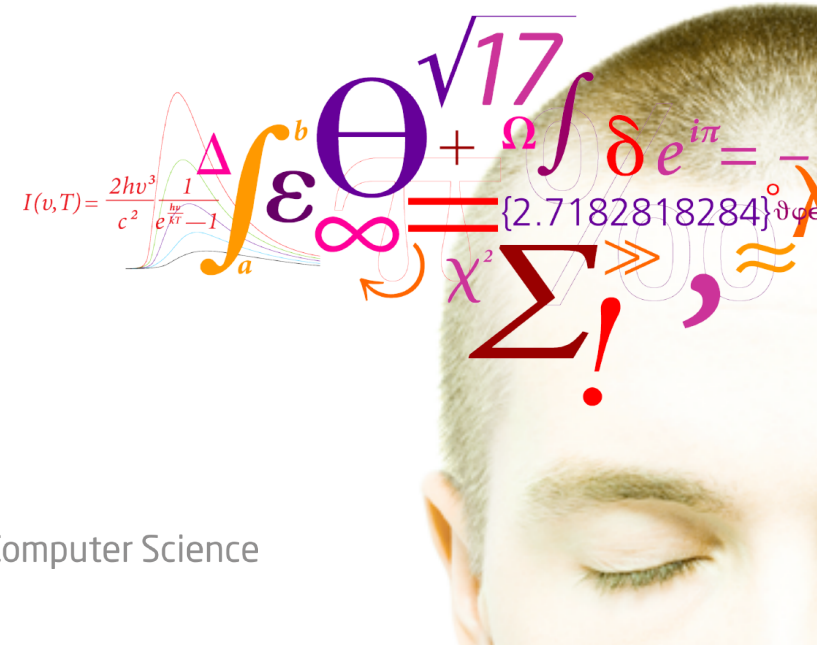
Introduction to programming and data processing

Project Catalogue for DTU Course 02631–34, 02691–94

Mikkel N. Schmidt, Vedrana Andersen, Martin S. Andersen,
Jens Starke, Nikolay K. Dimitrov.

2017



DTU Compute
Department of Applied Mathematics and Computer Science

Contents

Requirements for the project work

Introduction

In this project work you must develop, test, and document a program. The overall idea behind the projects is to test your understanding of the programming tools and methods you have learned throughout the course, and to do so in a fun and rewarding way. It is your responsibility to demonstrate that you have met the learning objectives (see the DTU Coursebase). Your code will be evaluated according to the criteria described below, and you must hand-in according to the submission requirements no later than the deadline given on CampusNet (Assignments). Late submissions will not be accepted.

Program specifications

In the project descriptions, the specifications for the program will be covered, i.e. the required functions and the main script. Note that the specifications are not exhaustive, and some challenges have been left for you to identify, solve, implement, and document. This is done on purpose; we want to see you make your own decisions and see how you choose to handle the problems you encounter. It is perfectly acceptable for you to make your own choices when something is not covered by the specifications. Do note that your choices must be documented and your arguments for making those choices will be evaluated.

General requirements

- Each function must be implemented according to the given interface and may not be implemented as a script. It is allowed to use sub-functions and built-in functions within a function. You may also develop "helper" functions that are called from within in the functions, if you believe this will result in a more elegant or practical solution.
- The main script must be implemented according to the given specifications. It is not allowed to implement the main script as a function.
- It is not allowed to invoke (call) scripts within another script or within a function. You may only implement one script, namely the the main script.
- You are allowed to use built-in functions, and you are encouraged to explore if there exists built-in functions that you can use in your own code.
- All functions must contain appropriate help text describing the purpose of the function, the input and output arguments, how the function is used, as well as other information that is necessary to understand and use the function. It is recommended to use help text structure used in the built-in functions as inspiration.
- Excessive error handling within a function may lead to a decrease in computational performance, especially if the function is called often. For this reason, it is often best practice to assume that input arguments for a function are valid (and state in the help text what requirements there are to the arguments.) Note that this does not cover interactive user input. Error check interactive user input within the scope where it is passed to the program.

Your own extensions

- You are encouraged to extend the functionality of the specified functions, if you manage to complete the specified requirements for functions and script well before the deadline. This way you get more programming practice, and you can further demonstrate the extend of your programming knowledge. Note: Should you choose to extend the functions, you may extend the specified function interfaces, as long as it is possible to call the functions as specified in the descriptions.

Evaluation

The following criteria are used by us when evaluating your work.

- Fulfillment of the learning objectives, as specified in the Coursebase (http://www.kurser.dtu.dk) coupled to the curriculum.
- The completeness and the quality of your implementation evaluated according to project requirements, learning objectives and best practice programming as described in the learning material.

Specifically we will examine the following (non-exhaustive) list of criteria:

A) Specifications

- Is all the specified functionality implemented?
- Do the required functions follow the specified format?
- Are all other specifications followed?

B) Correctness

- Is data read in correctly?
- Do all functions do what they are supposed to?
- Are computations etc. done correctly?

C) Structure and choice of solutions

- Is there too long and confusing code?
- Are there over-complicated solutions that should be simplified?
- Are loops and vectorized computation used sensibly?
- Are suitable functions from existing program libraries used?

D) Visualization and graphical plots

- Do the plots show what they are supposed to?
- Are there reasonable axis labels, legens, titles etc.?
- Are the plots clear and easy to read?

E) Comments and documentation

- Is the interface to all functions documented sufficiently?
- Is there a suitable amount of comments in the code?
- Are the comments good and useful for understanding the code?

F) Userfriendlyness and error handling

- Is the program easy and logical for a user to use?
- Is the user informed about how to use the program?
- Is erroneous user input handled in a reasonable way?

Submission details

Before the deadline (see CampusNet Assignments), you must submit all implemented code files (main script and functions) as well as other dependencies, separately or in one compressed zip-file as a "group hand in". Make sure to test that script and functions run without error before you hand-in, as part of our evaluation of your work is execution and testing of the program.

Project 1

Exercise project

Bacteria Data Analysis

Introduction

In this project you will develop a computer program for handling data related to growthrates of different bacteria at different temperatures. You must implement the functions and main script described in the following according to the specifications.

Data load function

Interface

```
def dataLoad(filename):
    # Insert your code here
    return data
```

Input arguments

`filename`: A string containing the filename of a data file.

Output arguments

`data`: An $N \times 3$ matrix.

User input

No.

Screen output

Yes (error message, see specifications below.)

Description

The function must read the data in the data file `filename`. Each line in the data file consists of the following fields:

`Temperature`, `Growth rate`, and `Bacteria`.

The fields contain numeric values and are separated by a space character. The following illustrates an example excerpt of such a data file:

```
25  0.109  1
20  0.096  2
15  0.517  3
35  1.086  4
40  0.934  2
35  0.109  1
 ⋮
```

The `Bacteria` field is a numeric code that correspond to one of the following bacteria names:

`Bacteria`	Name
1	Salmonella enterica
2	Bacillus cereus
3	Listeria
4	Brochothrix thermosphacta

The data in the file must be stored in an $N \times 3$ matrix called `data`, where N is the number of valid rows in the data file.

Handling data errors

There might be one or more erroneous lines in the data file which the function must handle. If the data load function detects an erroneous line in the data file it must skip this line, output an error message to the screen, and continue with the next line. The function must only return the data from the the valid rows. The error message must explain in which line the error occurred and what the error was. The function should check for the following:

- The `Temperature` must be a number between 10 and 60.
- The `Growth rate` must be a positive number.
- The `Bacteria` must be one of the four mentioned in the table above.

Data statistics function

Interface

```
def dataStatistics(data, statistic):
    # Insert your code here
    return result
```

Input arguments

`data`: An $N \times 3$ matrix with columns Temperature, Growth rate, and Bacteria.
`statistic`: A string specifying the statistic that should be calculated.

Output arguments

`result`: A scalar containing the calculated statistic.

User input

No.

Screen output

No.

Description

This function must calculate one of several possible statistics based on the data. A "statistic" here denotes a single number which describes an aspect of the data, as for example a mean (average) value. The statistic that must be calculated depends on the value of the string `statistic`. The following table shows the differeninterfacet possible values of `statistic` and a description of how to calculate the corresponding statistic.

`statistic`	Description
`'Mean Temperature'`	Mean (average) Temperature.
`'Mean Growth rate'`	Mean (average) Growth rate.
`'Std Temperature'`	Standard deviation of Temperature.
`'Std Growth rate'`	Standard deviation of Growth rate.
`'Rows'`	The total number of rows in the data.
`'Mean Cold Growth rate'`	Mean (average) Growth rate when Temperature is less than 20 degrees.
`'Mean Hot Growth rate'`	Mean (average) Growth rate when Temperature is greater than 50 degrees.

You are encouraged to use suitable built-in functions to compute the statistics where possible.

Data plot function

Interface

```
def dataPlot(data):
    # Insert your code here
```

Input arguments

`data`: An $N \times 3$ matrix with columns Temperature, Growth rate, and Bacteria.

Output arguments

No.

User input

No.

Screen output

Yes (plots, see specifications below.)

Description

This function must display two plots:

1. "Number of bacteria": A bar plot of the number of each of the different types of Bacteria in the data.
2. "Growth rate by temperature": A plot with the Temperature on the x-axis and the Growth rate on the y-axis. The x-axis must go from 10 to 60 degrees, and the y-axis must start from 0. The plot should contain a single axis with four graphs, one for each type of Bacteria. The different graphs must be distinguished using e.g. different colors, markers, or line styles.

The plots should include a suitable title, descriptive axis labels, and a data legend where appropriate. You are allowed to present the plots in separate figure windows or as sub-plots in a single figure window.

Main script

Interface Must be implemented as a script.

Input arguments No.

Output arguments No.

User input Yes (see specifications below.)

Screen output Yes (see specifications below.)

Description The user of the data analysis program interacts with the program through the main script. When the user runs the main script he/she must have at least the following options:

1. Load data.
2. Filter data.
3. Display statistics.
4. Generate plots.
5. Quit.

The user must be allowed to perform these actions (see specifications below) in any order as long as he/she chooses, until he/she decides to quit the program. The details of how the main script is designed and implemented is for you to determine. It is a requirement that the program is interactive, and you should strive to make it user friendly by providing sensible dialogue options. Consider what you would expect if you were to use such a script, and what you think would be fun. Play around and make a cool script.

Error handling You must test that all input provided by the user are valid. If the user gives invalid input, you must provide informative feedback to the user and allow the user to provide the correct input.

It must not be possible to display statistics or generate plots before any data has been loaded. If the user attempts to do this, he/she should be given appropriate feedback stating that this is not possible.

1. Load data If the user chooses to load data, you must ask the user to input the filename of a data file. Remember to check if the file name is valid. Load in the data using the `dataLoad` function which will display information about any erroneous lines in the data file.

2. Filter data If the user chooses to filter data, he/she must be able to specify one or more conditions which must be satisfied in order for the data rows to be included in the calculation of statistics and generation of plots. As a minimum requirement the user must be able to specify two types of filters:

1. A filter for the Bacteria type, for example `Bacteria` $=$ *Listeria*.
2. A range filter for the Growth rate, for example $0.5 \leq$ `Growth rate` ≤ 1.

Having specified such a filter, statistics and plots should be generated only for the subset of data rows where the condition is met. The user should also be able to disable the filter conditions, and when he/she does this, subsequently the statistics and plots should again be generated for the whole data. If a filter is active, information about the filter should at all times be visible in the user interface.

3. Display statistics If the user chooses to display statistics, you must ask the user which statistic to display. Use the `dataStatistics` function to compute the desired statistic and display it on the screen along with a description of the statistic. If a filter is active, the statistics must be computed only for data rows that satisfy the filter conditions.

4. Generate plots

If the user chooses to generate plots, call the `dataPlot` function to display the plots. If a filter is active, the plots must be generated based only on data rows that satisfy the filter conditions.

5. Quit

If the user chooses to quit the program, the main script should stop.

Data file

You are encouraged to make your own test data file in order to validate that your program functions correctly. It is important that you also ensure and document that your program works correctly in case of errors in the data file as described in section .

Project 2

Exam project

Program for grading students

Introduction

In this project you must develop, test, and document a program for processing grades for students. You must implement the functions and main script described in the following according to the specifications.

Grade rounding function

Interface

```
def roundGrade(grades):
    # Insert your code here
    return gradesRounded
```

Input arguments

`grades`: A vector (each element is a number between -3 and 12).

Output arguments

`gradesRounded`: A vector (each element is a number on the 7-step-scale).

User input

No.

Screen output

No.

Description

The function must round off each element in the vector `grades` and return the nearest grade on the 7-step-scale:

7-step-scale: Grades	12	10	7	4	02	00	-3

For example, if the function gets the vector [8.2, -0.5] as input, it must return the rounded grades [7, 0] which are the closest numbers on the grading scale.

Final grade function

Interface

```
def computeFinalGrades(grades):
    # Insert your code here
    return gradesFinal
```

Input arguments

`grades`: An $N \times M$ matrix containing grades on the 7-step-scale given to N students on M different assignments.

Output arguments

`gradesFinal`: A vector of length n containing the final grade for each of the N students.

User input

No.

Screen output

No.

Description

For each student, the final grade must be computed in the following way:

1. If there is only one assignment ($M = 1$) the final grade is equal to the grade of that assignment.
2. If there are two or more assignments ($M > 1$) the lowest grade is discarded. The final grade is computed as the mean of $M - 1$ highest grades rounded to the nearest grade on the scale (using the function `roundGrade`).
3. Irrespective of the above, if a student has received the grade -3 in one or more assignments, the final grade must always be -3.

Grades plot function

Interface

```
def gradesPlot(grades):
    # Insert your code here
```

Input arguments

`grades`: An $N \times M$ matrix containing grades on the 7-step-scale given to N students on M different assignments.

Output arguments

No.

User input

No.

Screen output

Yes (plots, see specifications below.)

Description

This function must display two plots:

1. "Final grades": A bar plot of the number of students who have received each of possible final grades on the 7-step-scale (computed using the function `computeFinalGrades`).
2. "Grades per assignment": A plot with the assignments on the x-axis and the grades on the y-axis. The x-axis must show all assignments from 1 to M, and the y-axis must show all grade -3 to 12. The plot must contain:
    1. Each of the given grades marked by a dot. You must add a small random number (between -0.1 and 0.1) to the x- and y-coordinates of each dot, to be able tell apart the different dots which otherwise would be on top of each other when more than one student has received the same grade in the same assignment.
    2. The average grade of each of the assignments plotted as a line

The plots should include a suitable title, descriptive axis labels, and a data legend where appropriate. You are allowed to present the plots in separate figure windows or as sub-plots in a single figure window.

Main script

Interface

Must be implemented as a script.

Input arguments

No.

Output arguments

No.

User input

Yes (see specifications below.)

Screen output

Yes (see specifications below.)

Description

When the user runs the main script he/she must first be asked to enter the name of a comma-separated-values (CSV) file containing grades given to a number of students for a number of assignments (see description of file format below). Remember to check if the file name is valid. After reading in the data, you must display some information about the loaded data, including at least the number of students and the number of assignments.

Next, the user must have at least the following options:

1. Load new data.
2. Check for data errors.
3. Generate plots.
4. Display list of grades.
5. Quit.

The user must be allowed to perform these actions (see specifications below) in any order as long as he/she chooses, until the user decides to quit the program. The details of how the main script is designed and implemented is for you to determine. It is a requirement that the program is interactive, and you should strive to make it user friendly by providing sensible dialogue options. Consider what you would expect if you were to use such a script, and what you think would be fun. Play around and make a cool script.

File format

The first row in the CSV file will contain the column headings. Each of the following rows will contain a student id, a name, and a number of grades for a student. An example of a data file with four students and three assignments is given below:

```
StudentID,Name,Assignment1,Assignment2,Assignment3
s123456,Michael Andersen,7,7,4
s123789,Bettina Petersen,12,10,10
s123468,Thomas Nielsen,-3,7,2
s123579,Marie Hansen,10,12,12
```

Remember that your program should work for any number of students and any number of assignments. You are encouraged to make your own test data file in order to validate that your program functions correctly.

Error handling

You must test that all input provided by the user are valid. If the user gives invalid input, you must provide informative feedback to the user and allow the user to provide the correct input.

1. Load new data

If the user chooses to load new data, the user must be asked to input a valid filename of a data file, and data must be loaded in the same way as in the beginning of the script.

2. Check for data errors

If the user chooses to check for data errors, you must display a report of errors (if any) in the loaded data file. Your program must at least detect and display information about the following possible errors:

1. If two students in the data have the same student id.
2. If a grade in the data set is not one of the possible grades on the 7-step-scale.

3. Generate plots

If the user chooses to generate plots, call the `gradesPlot` function to display the plots.

4. Display list of grades

If the user chooses to display the list of grades, you must display the grades for each assignment as well as the final grade for all of the students in alphabetical order by their name. The displayed list must be formatted in a way so that it is easy to read.

5. Quit

If the user chooses to quit the program, the main script should stop.

Beam deflection

Introduction

In this project you must develop, test, and document a program for analysis of a physical system in which a beam is deflected by a load. You must implement the functions and main script described in the following according to the specifications.

Beam deflection function

Interface

```
def beamDeflection(positions, beamLength, loadPosition, loadForce,
    beamSupport):
    # Insert your code here
    return deflection
```

Input arguments

`positions`: Positions [m] to compute the beam deflection (vector), below denoted x.
`beamLength`: Length of beam [m] (scalar), below denoted ℓ.
`loadPosition`: Position of the load [m] (scalar), below denoted a.
`loadForce`: Force of the load [N] (scalar), below denoted W.
`beamSupport`: Support of beam (string), equal to `both` or `cantilever`.

Output arguments

`deflection`: Deflection [m] (vector of same length as input `positions`), below denoted y.

User input

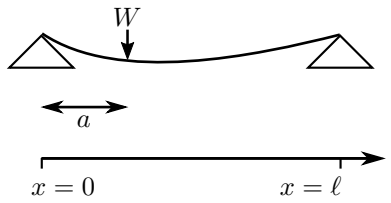
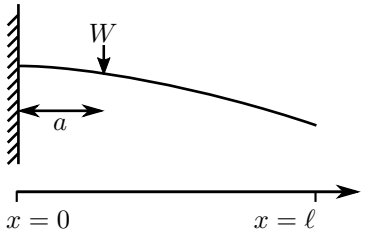
No.

Screen output

No

Description

The function must compute the deflection of a beam of the given length with a single load of the given force placed at the given position, according to the formulas below. The beam can either be supported at both ends or supported only at one end (cantilever).

`beamSupport`	Sketch	Formula for deflection
`both`		$y = \begin{cases} \frac{W(\ell-a)x}{6EI\ell}(\ell^2 - x^2 - (\ell-a)^2), & x < a \\ \frac{Wa(\ell-x)}{6EI\ell}(\ell^2 - (\ell-x)^2 - a^2), & x \geq a \end{cases}$
`cantilever`		$y = \begin{cases} \frac{Wx^2}{6EI}(3a - x), & x < a \\ \frac{Wa^2}{6EI}(3x - a), & x \geq a \end{cases}$

$E = 200 \cdot 10^9$ [N/m^2], $I = 0.001$ [m^4].

Superposition function

Interface

```
def beamSuperposition(positions, beamLength, loadPositions, loadForces
    , beamSupport):
    # Insert your code here
    return deflection
```

Input arguments

`positions`: Positions [m] to compute the beam deflection (vector).
`beamLength`: Length of beam [m] (scalar).
`loadPositions`: Positions of the loads [m] (vector).
`loadForces`: Forces of the loads [N] (vector).
`beamSupport`: Support of beam (string), equal to `both` or `cantilever`.

Output arguments

`deflection`: Deflection [m] (vector of same length as input `positions`).

User input

No.

Screen output

No

Description

The function must compute the deflection of a beam of the given length and support with *multiple loads* of the given forces placed at the given positions. To compute the deflection under multiple loads, the superposition principle states that you must compute the deflection for each of the loads separately and sum up the deflections.

If there are no loads (if the vectors `loadPositions` and `loadForces` are empty) the function should return a zero vector of the same length as `positions`.

Beam plot function

Interface

```
def beamPlot(beamLength, loadPositions, loadForces, beamSupport):
    # Insert your code here
```

Input arguments

`beamLength`: Length of beam [m] (scalar).
`loadPositions`: Positions of the loads [m] (vector).
`loadForces`: Forces of the loads [N] (vector).
`beamSupport`: Support of beam (string), equal to `both` or `cantilever`.

Output arguments

No.

User input

No.

Screen output

Yes (plots, see specifications below.)

Description

This function must display a plot titled: "Beam deflection"

The plot must show the beam deflection as a function of the x-coordinate for the whole beam (i.e. a plot similar to the sketches in section but with multiple loads). From the plot it must be easy to see or read off the length of the beam, the positions and forces of the loads, the maximum deflection, and the type of support of the beam.

The plot should include a title, descriptive axis labels, and a data legend if appropriate.

Main script

Interface

Must be implemented as a script.

Input arguments

No.

Output arguments

No.

User input

Yes (see specifications below.)

Screen output

Yes (see specifications below.)

Description

This program allows the user to configure a beam with multiple loads, save/load a configured beam, and display a plot of the beam deflection. The user of the program interacts with the program through the main script. When the user runs the main script he/she must have at least the following options:

1. Configure beam.

2. Configure loads.
3. Save beam and loads.
4. Load beam and loads.
5. Generate plot.
6. Quit.

The user must be allowed to perform these actions (see specifications below) in any order as long as he/she chooses, until he/she decides to quit the program. The details of how the main script is designed and implemented is for you to determine. It is a requirement that the program is interactive, and you should strive to make it user friendly by providing sensible dialogue options. Consider what you would expect if you were to use such a script, and what you think would be fun. Play around and make a cool script.

Error handling

You must test that all input provided by the user are valid. If the user gives invalid input, you must provide informative feedback to the user and allow the user to provide the correct input. It must not be possible for loads to be placed outside the beam or to have negative beam length or negative load forces.

Initialization

When the program starts the beam length must be set to 10 [m], the support must be set to `both`, and there must be no loads.

1. Configure beam

If the user chooses to configure a beam, he/she must be asked to input the beam length and the type of support.

2. Configure loads

If the user chooses to configure loads, he/she must have the opportunity to see the current loads (if any), add a load (position and force), or remove a load.

3. Save beam and loads

If the user chooses to save, he/she must be asked to input a file name. The current beam and loads (length of beam, type of support, positions and forces of loads) must be saved to the file. You must decide on a file format to use.

4. Load beam and loads

If the user chooses to load a beam and loads, he/she must be asked to input a filename, and the beam and loads must be read from the file. The program must be able to read the files saved in the menu option above.

4. Generate plot

If the user chooses to generate the plot, call the `beamPlot` function to display the plots.

5. Quit

If the user chooses to quit the program, the main script should stop.

Lindenmayer systems

Introduction

In this project you must develop, test, and document a program for computing and plotting Lindenmayer systems. You must implement the functions and main script described in the following according to the specifications.

Lindenmayer systems A Lindenmayer system is defined iteratively, and it consists of: a) an alphabet of symbols which can be used to create strings, b) an initial string used to begin the iterative construction and c) replacement rules that specify how to replace selected symbols of the string by strings of symbols (from the same alphabet). Originally, these Lindenmayer systems were used to describe the behaviour of plant cells and to model the growth processes of plant development. In this exercise you will work with two systems called the Koch curve and the Sierpinski triangle.

Koch curve The Koch curve can be generated using a) an alphabet consisting of the symbols S, L and R, b) the initial string 'S' and c) the replacement rules

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{SLSRSLS} \\ \texttt{L} &\rightarrow \texttt{L} \\ \texttt{R} &\rightarrow \texttt{R} \end{aligned}$$

The initial string is `S`. After the first iteration one obtains the string `SLSRSLS`. After the second iteration one obtains the string `SLSRSLSLSLSRSLSRSLSRSLSLSLSRSLS`. The first three iterations are visualized using turtle graphics (see below) in Fig. 2D.1.

Sierpinski triangle The Sierpinski triangle can be generated using a) an alphabet consisting of the symbols A, B, L and R, b) the initial string `A` and c) the replacement rules for each step of the iteration.

$$\begin{aligned} \texttt{A} &\rightarrow \texttt{BRARB} \\ \texttt{B} &\rightarrow \texttt{ALBLA} \\ \texttt{L} &\rightarrow \texttt{L} \\ \texttt{R} &\rightarrow \texttt{R} \end{aligned}$$

The initial string is `A`. After the first iteration one obtains the string `BRARB`. After the second iteration one obtains the string `ALBLARBRARBRALBLA`. The inititalization and the result after iteration 4 and 8 is visualized using turtle graphics (see below) in Fig. 2D.2.

Visualization using turtle graphices The sequence of symbols in the string will be visualized graphically by translating each symbol in a command for a so-called turtle graphics: `S`, `A`, and `B` is interpreted as the line segment from an initial location (we will use the origin of the planar coordinate system for this) drawn along an initial direction — we use for this the canonical basis vector $(1, 0)^T$. `L` is interpreted as turning left with the angle $\frac{1}{3}\pi$ and `R` as turning right with the angle $-\frac{2}{3}\pi$ (Koch) or $-\frac{1}{3}\pi$ (Sierpinski). The length of the line segment is scaled by a factor $\frac{1}{3}$ (Koch) or $\frac{1}{2}$ (Sierpinski) after each iteration step.

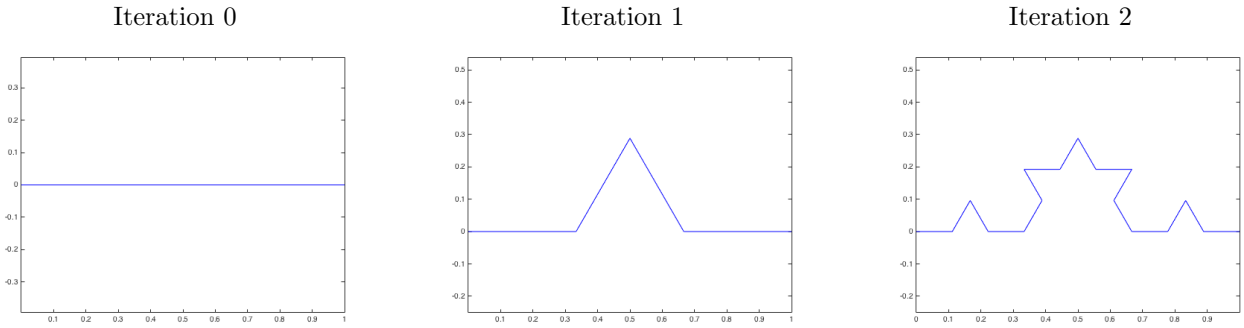


Figure 2D.1: Graphical representation of the first iterations of the Koch curve.

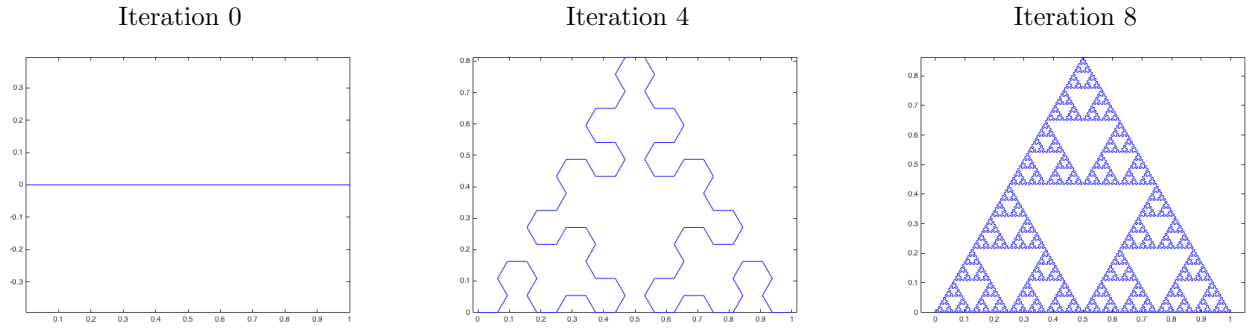


Figure 2D.2: Graphical representation of the initial, fourth and eights iteration of the Sierpinski triangle.

Lindenmayer Iteration

Interface

```
def LindIter(System, N):
    # Insert your code here
    return LindenmayerString
```

Input arguments

`System`: A string containing the name of the Lindenmayer system currently under scrutiny. The input can take the values `Koch` or `Sierpinski`.
`N`: The number of iterations that should be calculated.

Output arguments

`LindenmayerString`: A string containing the output after `N` iterations of the chosen Lindenmayer system.

User input

No.

Screen output

No.

Description

The function must calculate N iterations of the system specified by `System` according to the replacement rules for the Koch curve and Sierpinski triangle respectively.

The output of the iteration function should be stored in the string `LindenmayerString`.

Translation to turtle graphics commands

Interface

```
def turtleGraph(LindenmayerString):
    # Insert your code here
    return turtleCommands
```

Input arguments

`LindenmayerString`: A string of symbols representing the state of the system after the Lindemayer iteration.

Output arguments

`turtleCommands`: A row vector containing the turtle graphics commands consisting of alternating length and angle specifications $[l_1, \phi_1, l_2, \phi_2, \dots]$.

User input

No.

Screen output

No.

Description

This function translates the string of symbols in `LindenmayerString` into a sequence of turtle graphics commands. The output consists of a row vector of numbers, which alternate between a number specifying the length of a line to be drawn and a number specifying the angle of the new line segment with respect to the drawing direction of the last line segment.

Your function function should use the replacement rules consistent with the Lindenmayer system you choose to investigate. Which system this is can either be inferred

from the input `LindenmayerString` directly, or passed to the function by augmenting it by further input or output variables in the functions you program. You are free to choose how to implement this and creativity is appreciated.

Turtle graphics plot function

Interface

```
def turtlePlot(turtleCommands):
    # Insert your code here
```

Input arguments

`turtleCommands`: A row vector consisting of alternating length and angle specifications $[l_1, \phi_1, l_2, \phi_2, \dots]$.

Output arguments

No.

User input

No.

Screen output

Yes (plot, see specifications below.)

Description

The plot function should turn the input vector `turtleCommands` into a graphical visualisation. This can be done by specifying the coordinates of the corners $\vec{x} = (x, y)$ of the straight line segments with a plot command. To find these coordinates, your function should follow the input vector, starting at the origin of the planar coordinate system and adding a new pair of coordinates after every line segment. The line segment has the length l_i, and must be drawn at an angle ϕ_i with respect to the previous line (as specified in the input vector of `turtleCommands`). The initial point is $\vec{x}_0 = (0, 0)^T$ and the initial direction is along a unit vector $\vec{d}_0 = (1, 0)^T$. Here a positive angle corresponds to turning left, and a negative one to turning right. The computation of the vector pointing to the point $\vec{x}_{i+1}$ by using the previous point $\vec{x}_i$ and previous drawing direction $\vec{d}_i$ with

$$\vec{d}_{i+1} = \begin{pmatrix} \cos(\phi_{i+1}) & -\sin(\phi_{i+1}) \\ \sin(\phi_{i+1}) & \cos(\phi_{i+1}) \end{pmatrix} \cdot \vec{d}_i \tag{2D.1}$$

$$\vec{x}_{i+1} = \vec{x}_i + l_{i+1} \cdot \vec{d}_{i+1} \tag{2D.2}$$

The results can be checked by comparing them with Fig. 2D.1 and 2D.2. The plots should include a suitable title, descriptive axis labels, and a data legend where appropriate. You are allowed to present the plots in separate figure windows or as sub-plots in a single figure window.

Main script

Interface

Must be implemented as a script.

Input arguments

No.

Output arguments

No.

User input

Yes (see specifications below.)

Screen output

Yes (see specifications below.)

Description

The user interacts with the program through the main script. When the user runs the main script he/she must have at least the following options:

1. Choose the type of Lindenmayer system and the number of iterations.
2. Generate plots.
3. Quit.

The user must be allowed to perform any reasonable number of Lindenmayer iterations for the chosen Lindenmayer system and subsequent graphical presentation of the obtained strings. The details of how the main script is designed and implemented is for you to determine. It is a requirement that the program is interactive, and you should strive to make it user friendly by providing sensible dialogue options. Consider what you would expect if you were to use such a script, and what you think would be fun. Play around and make a cool script.

Error handling You must test that all input provided by the user are valid. If the user gives invalid input, you must provide informative feedback to the user and allow the user to provide the correct input. This includes also unreasonable choices with respect to the number of iterations (e.g. because of computational time).

1. Choose Lindenmayer system When the user chooses a Lindenmayer system, you must ask the user to input one of the options listed in the following table.

`Input Option`	Name
1	Koch curve
2	Sierpinski triangle

2. Generate plots If the user chooses to generate plots, call the `turtlePlot` function to display the plots corresponding to the current Lindenmayer system.

3. Quit If the user chooses to quit the program, the main script should stop.

Possible extension As a possible extension (not required) you could allow a user to define a new Lindenmayer system and to compute iterations of that and visualize this graphically. Further possibilities are to compute certain properties of the Lindenmayer systems like the curve length depending on the number of iterations.

Wind data analysis

Introduction

In this project you will develop a computer program for handling data related to wind energy. You must implement the functions and main script described in the following according to the specifications.

Data load function

Interface

```
def dataLoad(filename, Nx, Ny, Nz):
    # Insert your code here
    return data
```

Input arguments

`filename`: A string containing the filename of a data file.
`Nx`, `Ny`, `Nz`: Size of the first, second, and third dimension (x-, y-, and z-coordinate) of the 3-dimensional output array.

Output arguments

`data`: A 3-dimensional array with size $N_x \times N_y \times N_z$.

User input

No.

Screen output

Yes (error message, see specifications below.)

Description

The function must read the data in the binary data file `filename`. Note that the file is a binary file and not a text file. The data file consists of $N_z \times N_y \times N_x$ numbers (floating point single precision). The sequence of the numbers corresponds to indexes z, y and x which all increase sequentially from 1 to N_z, N_y and N_x respectively. The fastest-varying index is z, followed by y, and the slowest varying index is x. That is, the first N_z numbers from the sequence correspond to indexes z going from 1 to N_z, $y = 1$, and $x = 1$. Based on this ordering rule, the function must convert the data to a three-dimensional array with dimensions $N_z \times N_y \times N_x$.

Handling data errors

There might be a disagreement between the number of data points in the input file, and the specified array dimensions. If the number of data points does not match the expected number, it must print an suitable informative error message.

Data statistics function

Interface

```
def dataStatistics(data, statistic, Yref, Zref, DeltaX):
    # Insert your code here
    return result
```

Input arguments

`data`: An $N_z \times N_y \times N_x$ array containing wind speed values.
`statistic`: A string specifying the statistic that should be calculated (`Mean`, `Variance`, or `Cross correlation`.)
`Yref`, `Zref`: The reference y- and z-coordinate for the cross-correlation. Only needed when `statistic` is `Cross correlation`. Denoted y_{ref} and z_{ref} below.
`DeltaX`: Separation in x-coordinate for which the cross-correlation has to be evaluated. Only needed when `statistic` is `Cross correlation`. Denoted Δx below.

Output arguments

`result`: A two-dimensional array with size $(N_y \times N_z)$ containing the calculated statistic for each point in the y-z plane.

User input

No.

Screen output

No.

Description

This function must calculate one of three possible statistics based on the data. A "statistic" here denotes an $N_y \times N_z$ matrix of numbers calculated from the values along the x-dimension for each of the y- and z-coordinates. The following shows the different possible statistics and how to calculate them:

`Mean` The mean value for each point in the y-z plane, calculated over the x-dimension.

$$M_{y,z} = \frac{1}{N_x} \sum_{x=1}^{N_x} D_{x,y,z} \tag{2E.1}$$

`Variance` The variance for each point in the y-z plane, calculated over the x-dimension.

$$V_{y,z} = \frac{1}{N_x} \sum_{x=1}^{N_x} \left(D_{x,y,z} - M_{y,z} \right)^2 \tag{2E.2}$$

`Cross-correlation` The cross-correlation at lag Δx betweeen each time series in the y-z plane and the reference time series at y_{ref}, z_{ref}.

$$C_{y,z} = \frac{1}{N_x - \Delta x} \sum_{x=1}^{N_x - \Delta x} D_{x,y,z} D_{x+\Delta x, y_{\text{ref}}, z_{\text{ref}}} \tag{2E.3}$$

You are encouraged to use suitable built-in functions to compute the statistics if it is possible.

Data plot function

Interface

```
def dataPlot(data, statistic):
    # Insert your code here
```

Input arguments

`data`: An $N_y \times N_z$ array containing calculated statistics.
`statistic`: A string specifying the type of statistic to be plotted (`Mean`, `Variance`, or `Cross correlation`.)

Output arguments

No.

User input

No.

Screen output

Yes (plots, see specifications below.)

Description

This function must produce a plot of the statistics calculated using the function `dataStatistics`. The function must produce a 2-dimensional plot where the values of the array for the y- and z-coordinates can be seen, e.g a contour or a surface plot.

The plot must include a suitable title, descriptive axis labels, and a data legend where appropriate.

Main script

Interface

Must be implemented as a script.

Input arguments

No.

Output arguments

No.

User input

Yes (see specifications below.)

Screen output

Yes (see specifications below.)

Description

The user of the data analysis program interacts with the program through the main script. When the user runs the main script he/she must have at least the following options:

1. Load data.
2. Display statistics.
3. Generate plots.
4. Quit.

The user must be allowed to perform these actions (see specifications below) in any order as long as he/she chooses, until he/she decides to quit the program. The details of how the main script is designed and implemented is for you to determine. It is a requirement that the program is interactive, and you should strive to make it user friendly by providing sensible dialogue options. Consider what you would expect if you were to use such a script, and what you think would be fun. Play around and make a cool script.

Error handling

You must test that all input provided by the user are valid. If the user gives invalid input, you must provide informative feedback to the user and allow the user to provide the correct input.

It must not be possible to display statistics or generate plots before any data has been loaded. If the user attempts to do this, he/she should be given appropriate feedback stating that this is not possible.

1. Load data

If the user chooses to load data, you must ask the user to input the filename of a data file. Remember to check if the file name is valid. Ask the user for the dimensions of the data, and load in the data using the `dataLoad` function.

2. Display statistics

If the user chooses to display statistics, you must ask the user which statistic to display and for which coordinates to display the statistic. Use the `dataStatistics` function to compute the desired statistic and display it on the screen along with a description of the statistic. The user should at least be able to display the mean and variance for a given y- and z-coordiate.

4. Generate plots

If the user chooses to generate plots, you must ask the user which plot to generate. Then, call the `dataPlot` function to display the plot.

5. Quit

If the user chooses to quit the program, the main script should stop.

Data file

A large data-file with real wind measurements is available for testing your program. You are encouraged to also make your own test data file in order to validate that your program functions correctly.

Analysis of household electricity consumption

Introduction

In this project you must develop, test, and document a program for analysis of household electricity consumption. You must implement the functions and main script described in the following according to the specifications.

Data load function

Interface

```
def load_measurements(filename, fmode):
    # Insert your code here
    return (tvec, data)
```

Input arguments

`filename`: A string containing the filename of a data file.
`fmode`: A string specifying how to handle corrupted measurements.

Output arguments

`tvec`: An $N \times 6$ matrix where each row is a time vector.
`data`: An $N \times 4$ matrix where each row is a set of measurements.

User input

No.

Screen output

Warnings (see description below).

Description

The function must read the data from the data file `filename`. Each line of the data file consists of the following fields:

`year`, `month`, `day`, `hour`, `minute`, `second`, `zone1`, `zone2`, `zone3`, `zone4`.

All fields contain numeric values and are comma separated. Here is an example of a line from a data file:

```
2008,1,5,15,8,0,12.0,0.0,18.0,35.4
```

This corresponds to a measurement recorded/started on January 5, 2008 at 15:08:00, and the four measurements are 12.0, 0.0, 18.0, and 35.4. The measurements are in Watt-hour and is the electricity consumption over a period of one minute in a household with four zones. Zone 1 is the kitchen (mainly a dishwasher, an oven and a microwave), zone 2 is the laundry room (a washing-machine, a tumble-drier, a refrigerator and a light), zone 3 contains an electric water-heater and an air-conditioner, and zone 4 contains all electrical equipment not measured in Zones 1-3.

The rows in the data file are in chonological order, but some rows may contain corrupted measurements, and these measurements have the value -1. For example, the following row corresponds to a corrupt measurement in zone 2:

```
2006,2,13,9,15,0,10.0,-1,20.0,32.8
```

The second input `fmode` specifies how to handle corrupted measurements:

`fmode`	Description
`'forward fill'`	replace with lastest valid measurement
`'backward fill'`	replace with next valid measurement
`'drop'`	delete corrupted measurements

If `fmode` is `'forward fill'` and the first row contrains a corrupted measurement or if `fmode` is `'backward fill'` and the last row contrains a corrupted measurement, then delete all rows with corrupted measurements and print a warning.

The data in the file must be stored in an $N \times 4$ matrix called `data` where column i corresponds the measurements in zone i and where N is either the number of rows in the data file (if `fmode` is `'forward fill'` or `'backward fill'`) or the number of valid

measurements (if `fmode` is `'drop'`). The time vectors must be stored in an $N \times 6$ matrix called `tvec` where columns 1-6 correspond to `year`, `month`, `day`, `hour`, `minute`, and `second`.

Data aggregation function

Interface

```
def aggregate_measurements(tvec, data, period):
    # Insert your code here
    return (tvec_a, data_a)
```

Input arguments

`tvec`: An $N \times 6$ matrix where each row is a time vector.
`data`: An $N \times 4$ matrix where each row is a set of measurements.
`period`: A string specifying how to aggregate measurements.

Output arguments

`tvec_a`: An $M \times 6$ matrix where each row is a time vector.
`data_a`: An $M \times 4$ matrix with aggregated measurements.

User input

No.

Screen output

No.

Description

The function takes as input a matrix of time vectors `tvec`, a matrix of measurements `data`, and a string `period` that determines how to aggregate[1] the measurements. The input `period` can take the following values:

`period`	Description
`'hour'`	hourly consumption
`'day'`	daily consumption
`'month'`	monthly consumption
`'hour of the day'`	average time-of-day consumption (hourly)

The outputs `data_a` and `tvec_a` contain the aggregated measurements and the corresponding time vectors. If `period` has the value `'hour'`/`'day'`/`'month'`, then all measurements recorded within the same hour/day/month are combined (summed), M is equal to the number of distinct hours/days/months during which measurements were recorded, and the rows of `tvec_a` identify the beginning of the M time period. If `period` has the value `'hour of the day'` then M is equal to 24 corresponding to the 24 time invervals 00:00-01:00, 01:00-02:00, ..., 23:00-00:00, the aggregated data matrix `data_a` contains the average consumption over all days in these time intervals, and the matrix `tvec_a` identifies the start of the time intervals.

Statistics function

Interface

```
def print_statistics(tvec, data):
    # Insert your code here
    return
```

Input arguments

`tvec`: An $N \times 6$ matrix where each row is a time vector.
`data`: An $N \times 4$ matrix where each row is a set of measurements.

Output arguments

No.

User input

No.

Screen output

Yes, a table with statistics (see description below).

[1] To aggregate means to "combine" or to "form or group into a class or cluster."

Description

The function prints out a table of the form

Zone	Minimum	1. quart.	2. quart.	3. quart.	Maximum
1					
2					
3					
4					
All					

with the following statistics/percentiles for each of the four zones as well as for all zones combined: minimum, 1. quartile, 2. quartile (median), 3. quartile, and maximum.

You are encouraged to use suitable built-in functions to compute the statistics.

Main script

Header

Must be implemented as a script.

User input

Yes (see specifications below.)

Screen output

Yes (see specifications below.)

Description

The user of the program interacts with the program through the main script. When the user runs the main script he/she must have at least the following options:

1. Load data.
2. Aggregate data.
3. Display statistics.
4. Visualize electricity consumption.
5. Quit.

The user must be allowed to perform these actions (see specifications below) in any order until he/she decides to quit the program. The details of how the main script is designed and implemented is for you to decide. It is a requirement that the program is interactive, and you should strive to make it user friendly by providing sensible dialogue options. Consider what you would expect if you were to use such a script, and what you think would be fun. Play around and make a cool script.

Error handling

You must test that all input provided by the user are valid. If the user gives invalid input, you must provide informative feedback to the user and allow the user to provide the correct input.

It must not be possible to aggregate data, display statistics, or generate plots before any data has been loaded. If the user attempts to do this, he/she should be given appropriate feedback stating that this is not possible.

1. Load data

If the user chooses to load data, you must ask the user to input the filename of a data file. Remember to check if the file name is valid. Ask the user how to handle corrupted measurements, and load in the data using the `load_measurements` function. As a minimum, the user must be able to choose the following methods for handling corrupted measurements:

1. Fill forward (replace corrupt measurement with latest valid measurement)
2. Fill backward (replace corrupt measurement with next valid measurement)
3. Delete corrupt measurements

2. Aggregate data

If the user chooses to aggregate data, he/she must be able to specify how to do so. As a minimum requirement, the user must have the following options:

1. Consumption per minute (no aggregation)
2. Consumption per hour
3. Consumption per day
4. Consumption per month
5. Hour-of-day consumption (hourly average)

The aggregated data must be used subsequently when computing statistics and visualizing data.

3. Display statistics If the user chooses to display statistics, then use the `print_statistics` function to compute and display a table of percentiles on the screen. The screen output should also include a description of the time scale (consumption per minute, hour, day, etc.) and the unit (e.g. Watt-hour or Kilowatt-hour).

4. Visualize If the user chooses to visualize the electricity consumption, ask the user if he/she wants to plot the consumption in each zone or the combined consumption (all zones). Display a plot with appropriate axes (time on the x-axis and consumption on the y-axis), labels, and a title. Use a bar chart if the aggregated data contains less than 25 measurements.

5. Quit Stop the main script if the user chooses to quit the program.

Data file A large data file with measurements from 2008 is available for testing, and in addition to this, you are encouraged to create your own data files with test cases in order to check that your program works correctly.