# Land surface process modelling with PCRaster Python

***Release 2.5***

**Derek Karssenberg, Judith Verstegen, Oliver Schmitz**

**Jan 30, 2026**

# CONTENTS


1 Map Algebra 3

2 Dynamic Modelling 21

3 Stochastic Modelling 65

4 Data Pre-Processing with GDAL 117

5 Programming with Python 127

6 Calibration 161

Download this website as pdf.

Download this website as epub (for e-readers).

To request the data set for the exercises email d.karssenberg@uu.nl

CHAPTER

ONE

# MAP ALGEBRA

Download this website as pdf.

Download this website as epub (for e-readers).

To subscribe to our courses visit http://www.pcraster.eu

## 1.1 PCRaster maps, data types and display

### 1.1.1 Introduction

The aim of these exercises is to learn how Geographical Information Systems can be used for the analysis and synthesis of spatial data in physical geography, soil science, hydrology and environmental science. The emphasis is on the understanding of the methods of analysis rather than on data structures, methods of data storage or producing high quality graphic display.

The exercises use the PCRaster Python software.

### 1.1.2 Reading maps from disk and visualisation

To run the software, open a Python command prompt and type at the prompt:

```
import pcraster as pcr <Enter>
```

This loads the PCRaster module.

To use the PCRaster maps in the exercise data set, you need to be sure that the path in your Python command prompt is directing to the folder containing the exercise data set. In many Python environments you can set this in advance, using an option in the environment. But you can also set it on the fly.

To check your path, type (still at the Python command prompt):

```
import os <Enter>
print(os.getcwd())
```

If it prints the path to the folder with your data set, proceed below at 'To use a map..'! If not, you can choose to set the path at the prompt, using *os.chdir(stringWithPath)*, where *stringWithPath* defines the path, for instance:

```
os.chdir(r"C:\Exercises\MapAlgebra") <Enter>
```

Check your path again by typing:

```
print(os.getcwd())
```

If all is fine, continue below.

To use a map, you first need to read it from disk:

```
waterMap = pcr.readmap("water.map") <Enter>
```

which reads the file `water.map` and assigns it to the variable `waterMap`, which is stored in memory.

Now you can visualise the map by typing:

```
pcr.aguila(waterMap) <Enter>
```

The maps provided are: `buildg.map`, `firestat.map`, `iswater.map`, `phreatic.map`, `rainstor.map`, `roads.map`, `topo.map`, `water.map`, `dump.map`, `isroad.map`, `logging.map`, `points.map`, `rainyear.map`, `soils.map`, `trees.map`, `wells.map`.

Now read the `isroad.map` from disk using the same command and use a variable name `isroadMap` and display it. You can display as many maps as you like. If you want to have the same cursor position, however, on all maps, it is better to open them at once, by using multiple maps as input of the aguila command, for instance:

```
pcr.aguila(isroadMap, waterMap)
```

Click on the maps to find some cells where you would expect a bridge, the cross indicates the cursor position. You can select 'Show cursor and values' from the Aguila menu to retrieve cell values.

Read from disk and display `topo.map`. It is the digital elevation model of the area. Right-click on the legend and select 'Edit draw properties..' to change settings. For instance, change the display to contours.

Below, you find the meaning of the codes used on the maps.

```
buildg.map
0 no buildings
1 house
2 public building
3 tip
4 industry
5 mine

dump.map
0 no dump
1 dump

firestat.map
0 no station
1 fire-station

isroad.map
0 no road
1 road

iswater.map
0 no water
1 water

logging.map
0 no felling
```

(continues on next page)

(continued from previous page)

```
1 felling

roads.map
0 no roads
1 dirt road
2 metalled road

soils.map
1 stream
2 swamp
3 lake
4 clay
5 silty clay
6 peat
7 sandy clay
8 gravel
9 sand
10 boulder clay

trees.map
0 open
1 pine
2 deciduous
3 mixed woods

water.map
0 dry land
1 streams
2 swamp
3 lakes

wells.map
0 no well
1 well
```

### 1.1.3 Data types

PCRaster uses data typing of the data in the database: each map has one of the six data types used attached to it. These data types help you and PCRaster to structure the data. See the table below.

| data type | description of attributes | domain | example |
|---|---|---|---|
| boolean | boolean | 0 (FALSE), 1 (TRUE) | suitable/unsuitable, visible/non visible |
| nominal | classified, no order | whole values | soil classes |
| ordinal | classified, order | whole values | succession stages, income groups |
| scalar | continuous, linear | real values | temperature, concentration |
| directional | continuous, directional | 0 to 360 degrees | aspect |
| ldd | direction to neighbour cell | codes of directions | drainage networks |

---

**Question 1:** What is the data type of `buildg.map`?

a) boolean

b) nominal

c) ordinal

d) scalar

e) directional

f) ldd

Correct answer: b

Feedback: Buildings is a classified data type without order, so it is a nominal data type.

---

## 1.2 Point operations

### 1.2.1 Introduction

Before we continue, type the following:

```
exec(open("openmaps.py").read())
```

It reads all maps and assigns them to corresponding variable names. For instance, the file `phreatic.map` is assigned to the variable `phreaticMap`, or `topo.map` becomes `topoMap`. If you close Python and restart it again, you need to retype this command to get all your maps loaded!!

New maps can be derived from existing maps with PCRaster functions and operators. These provide a very powerfull calculator for maps. It offers you a wide range of operators (for instance plus, minus, slope, windowaverage) that act upon input maps. If needed, the operators can be combined, in the same way as it is done in algebraic calculations. For instance the statement:

```
resultMap = pcr.slope(topoMap) * (1 + phreaticMap)  <Enter>
```

Is perfectly valid. Try it! And display the `resultMap` with Aguila.

The software contains point operations, area operations, neighbourhood operations and map operations. This section covers point operations. These operate only on the values of the map layers relating to each cell; in other words, for each cell the operation is independent of the cell value(s) of neighbouring cells.

### 1.2.2 Quantitative computation with scalar maps

Quantitative calculation includes several mathematical operations, such as * (multiply), `sin`, `ln`, +. The input maps of a quantitative computation must always be maps of scalar data type.

To create an overlay which contains the depth of the unsaturated zone, you can subtract for every raster cell the phreatic level (i.e., the groundwater level, `phreaticMap`) from the surface level (`topoMap`). Both maps give the level in metres above sea level. Type:

```
unsatMap = topoMap - phreaticMap <Enter>
```

Display the result.

Calculate a map that gives the depth of the unsaturated zone in mm. Use the operator * and call the result map `unsatmmMap`. Display the map.

Somebody wants to determine a map with the infiltration rate on basis of the soil type (`soilsMap`) and the unsaturated zone (`unsatMap`), by typing the following operation:

```
infilMap = soilsMap * unsatMap <Enter>
```

Try the operation and note the error message (particularly the last line).

---

**Question 2:** Why is it not possible to multiply `soilsMap` with `unsatMap`?

a) The `unsatMap` contains negative cell values, which does not make sense.

b) The data type of `soilsMap` is nominal, and nominal data types do not allow multiplication.

c) The data type of `unsatMap` is nominal, and nominal data types do not allow multiplication.

d) The `unsatMap` contains very high cell values, which does not make sense.

Correct answer: b

Feedback: The data type of the soils map is nominal, because it contains classes. A class cannot be input to a multiplication, for instance one cannot multiply a peat soil by twenty. This is why the error message is generated.

---

### 1.2.3 Boolean algebra operations

Boolean algebra is a combination technique that assumes that cells contain only two different values. Operations for boolean algebra can only be applied on maps with the related Boolean data type and the result will be a map of Boolean data type. On such a map, cells are said to be TRUE (a cell value of 1) if a certain attribute is located in that cell, and they are said to be FALSE (a cell value of 0) if the attribute is not located in that cell. Maps containing these conditions can be combined with boolean algebra operations.

You have maps containing the location of the roads (`isroadMap`) and the location of the surface water (`iswaterMap`). Display `isroadMap` and `iswaterMap`.

If we are interested in the location of bridges, we can assume that bridges occur where water coincides with roads. This is calculated with the & (and) operator as shown in the table below.

| | | `isroadMap` | |
|---|---|---|---|
| | **&** | False | True |
| `iswaterMap` | False | **False** | **False** |
| | True | **False** | **True** |

**Table:** Boolean values in bold provide result of the **&** operator with two Boolean inputs, `isroadMap` and `iswaterMap`, for all combinations of True and False inputs on these maps.

The operator `and` can be used to evaluate this boolean relation. Type:

```
isbridgeMap = isroadMap & iswaterMap <Enter>
```

Display the inputs and results in one statement and check the location of the bridges:

```
pcr.aguila(isroadMap, iswaterMap, isbridgeMap) <Enter>
```

Other Boolean operators are ~ (not), | (or) and ^ (xor). Type:

```
x1Map = isroadMap & iswaterMap <Enter>
x2Map = isroadMap | iswaterMap <Enter>
x3Map = isroadMap ^ iswaterMap <Enter>
x4Map = isroadMap & ~ iswaterMap <Enter>
```

And display the resulting maps.

Listed below are four cross tables.

| Table A | | isroadMap | |
|---|---|---|---|
| | | False | True |
| iswaterMap | False | **False** | **False** |
| | True | **False** | **True** |

---

| Table B | | isroadMap | |
|---|---|---|---|
| | | False | True |
| iswaterMap | False | **False** | **True** |
| | True | **False** | **False** |

---

| Table C | | isroadMap | |
|---|---|---|---|
| | | False | True |
| iswaterMap | False | **False** | **True** |
| | True | **True** | **True** |

---

| Table D | | isroadMap | |
|---|---|---|---|
| | | False | True |
| iswaterMap | False | **False** | **True** |
| | True | **True** | **False** |

---

**Question 3:** Which of the cross tables belongs to `x1Map = isroadMap and iswaterMap`?

a) Table A

b) Table B

c) Table C

d) Table D

Correct answer: a

Feedback: None

---

---

**Question 4:** Which of the cross tables belongs to `x2Map = isroadMap | iswaterMap`?

a) Table A

b) Table B

c) Table C

d) Table D

Correct answer: c

Feedback: None

---

---

**Question 5:** Which of the cross tables belongs to `x3Map = isroadMap ^ iswaterMap`?

a) Table A

b) Table B

c) Table C

d) Table D

Correct answer: d

Feedback: None

---

---

**Question 6:** Which of the cross tables belongs to `x4Map = isroadMap & ~ iswaterMap`?

a) Table A

b) Table B

c) Table C

d) Table D

Correct answer: b

Feedback: None

---

### 1.2.4 Comparison operators resulting in boolean maps

For each cell, the comparison operators define a relation between the cell value on a first input map and a second input map. If this relation holds, a boolean TRUE is assigned to the cell. If it does not hold, a Boolean FALSE is assigned to the cell. The resulting map is a Boolean map. The two input maps may be real PCRaster maps or a constant value representing a map that is totally filled with cells of that value.

Try:

highMap = topoMap > 40 <Enter>

Display `topoMap` and `highMap`.

The syntax of a PCRaster operator defines how the operator is used (for instance the number of input maps, the order in which the input maps must be typed in, the use of brackets). For instance, the syntax of the > (greater than) operator is: `Result = expression1 > expression2`. Result is the output map that is generated, expression1 and expression2 are the two input "maps". The name expression is used here instead of a map because it does not always need to be a map name. The expressions may be:

- names of variables (e.g. `topoMap`)
- constant values (like 40 in the example given above), or;
- operations resulting in maps.

An example of using an operation resulting in a map is:

```
testMap = topoMap > (0.234 * 19)
```

In this case `Result` is `testMap`, `expression1` is `topoMap` and `expression2` is an operation resulting in a map: (0.234 * 19), 0.234 multiplied with 19.

In addition to the > (greater than) operator, you can use == (equal), >= (greater than or equal), <= (less than or equal), < (less than) and != (not equal). These operators have a syntax that corresponds with the > operator.

Display `buildgMap` and click on the map to find the cell value used to represent the mine (if you don't see the legend, it is code 5 in the legend). Make a boolean map, call it `isMineMap`, that is TRUE at cells with the mine and FALSE at cells without the mine. Use the `buildgMap` and the == operator. Display `isMineMap` together with the `buildgMap`.

Give the command you typed to get `isMineMap`.

---

**Question 7:** What command did you use to get `isMineMap`.

a) `isMineMap = buildgMap = 5`

b) `isMineMap = buildgMap == mine`

c) `isMineMap = buildgMap == 5`

d) `isMineMap = buildgMap = mine`

Correct answer: c

Feedback: Note that in PCRaster you cannot use names for legend entries in operations. This is why the constructs with `mine` do not work. You have to refer to the cell codes associated with classes, always. Mine has a code 5 on the map, and to select mine, you need to select cells with code 5.

---

### 1.2.5 Conditional operators with a boolean map

The conditional operators use an input map of Boolean data type to determine whether a first expression, a missing value or a second expression must be assigned to a particular cell.

There are two conditional operators. The first conditional operator is `ifthen`, with the following syntax: `Result = pcr.ifthen(condition, expression)` where `Result` is the output map, `condition` is a Boolean map or expression and `expression` is the expression that is assigned to Result in those cells where condition has a TRUE value (cell value 1). For those cells where condition is FALSE (cell value 0) the value of Result is not defined and a missing value is assigned.

The elevation at the mine can be determined with `isMineMap` and `topoMap`. The `isMineMap` was created during the previous exercise. Type:

```
topatminMap = pcr.ifthen(isMineMap, topoMap) <Enter>
```

Display `topoMap`, `isMineMap` and `topatminMap`. The map `isMineMap` is TRUE (cell value 1) at the mine and the resulting map `topatminMap` has the value of `topoMap` for that area. The map `isMineMap` is FALSE (cell value 0) for all the remaining cells and `topatminMap` is assigned a missing value for these cells.

The second conditional operator is the if then else operator, with the following syntax: `Result = ifthenelse(condition,expression1,expression2)` where `Result` is the output map, `condition` is a boolean expression, `expression1` is the expression that is assigned to `Result` in those cells where condition has a TRUE value (cell value 1) and `expression2` is the expression that is assigned to `Result` in those cells where `condition` has a FALSE value (cell value 0)

In the future, the mine on `isMineMap` will be used for open-cast mining (Dutch: 'dagbouw'): 20 metres of ground will be removed at the mine. As a result the surface level at the mine will decrease with 20 metres compared to the current surface level (given on `topoMap`).

Calculate the elevation map of the whole study area (no missing values) that will result after digging down 20 metres of ground at the mine. Call this new elevation map `toponewMap`. Use the `ifthenelse operator`.

To answer the following question, calculate the lowest elevation value on `toponewMap`:

```
lowestPointOnTopoNew = pcr.mapminimum(toponewMap)
```

---

**Question 8:** What is the lowest elevation value on `toponewMap`?

a) 1.0

b) -1.0

c) 3.0

d) 12.0

Correct answer: a

Feedback: None

```
toponewMap = pcr.ifthenelse(isMineMap, topoMap - 20, topoMap)
lowestPointOnTopoNew = pcr.mapminimum(toponewMap)
```

---

## 1.3 Area operations for descriptive statistics

### 1.3.1 Introduction

Area operations compute a new value for each cell as a function of existing cell values of cells associated with an area containing that cell. The area operations are like point operations to the extent that they compute new cell values on basis of one or more map layers. Unlike point operations, however, each cell value is determined on the basis of the several cell values in the zone containing the cell under consideration. In most cases the operation represents a descriptive statistics calculation, for instance the mean value over each area.

### 1.3.2 Calculation of statistics of an area

For calculation of a map that contains for each soil class the average elevation in the soil class, the `areaaverage` operator is used. Try:

```
soiltopoMap = pcr.areaaverage(topoMap, soilsMap) <Enter>
```

Display `soiltopoMap`, `topoMap` and `soilsMap`.

---

**Question 9:** What is calcualated by the `areaaverage` operation in this case?

a) The operation assigns to each cell the average elevation value of cells in a map that belong to the same area (class on `soilsMap`) as the cell itself.

b) The operation assigns the area of the area (class on `soilsMap`) to which the cell belongs to the cell itself.

c) It gives an average idea of the distribution of the elevation in the soil classes.

Correct answer: a

Feedback: None

---

For assigning the area of the soil class to which each cell belongs, the `areaarea` operator can be applied. Try:

```
soilareaMap = pcr.areaarea(soilsMap) <Enter>
```

Display `soilareaMap` and the `soilsMap`.

---

**Question 10:** What is name or cell code of the soil class that covers the largest area in the study area?

a) Gravel, with cell code 8.

b) Sand, with cell code 9.

c) Boulder clay, with cell code 10.

Correct answer: c

Feedback: None

---

### 1.3.3 Finding contiguous areas; a study problem

The pine forests in this area are partly used for commercial logging. One of the constraints to evaluate whether a certain patch of pine trees will produce enough yield to create some profit when logged, is the size of the patch. Logging of pine trees is economically feasible only for patches (contiguous areas) with pines that are larger than 4 hectares. The goal of this question is to create a boolean map that has TRUE cell values for patches with pine that are feasible for logging and a boolean FALSE cell values for the remaining area.

Display `treesMap`.

If you cannot see the codes, it has the following cell codes:

- open, code 0
- pine, code 1

- deciduous, code 2
- mixed wood, code 3

Create a boolean map (call it `pineMap`) that contains a 1 (TRUE) for cells with pines and a 0 (FALSE) for cells without pines. Use the `treesMap` in an operation with ==. Display the map you just created, `pineMap`.

> To be able to determine the size of each of the individual patches of pine-trees a separation in contiguous areas on `pineMap` has to be made. This is done with the clump operator. It groups in a boolean, nominal or ordinal map all those cells that have the same class-value and neighbour each other. Have a look at the PCRaster manual page for `clump` to get more information.

Every group ('clump') of cells satisfying these conditions is assigned a new class-value in the resulting map. Type:

```
pineclumMap = pcr.clump(pineMap) <Enter>
```

And display `pineMap` and `pineclumMap`.

Now, calculate a map (call it `pineareaMap`) that contains for each clump on `pineclumMap` the area of the clump under consideration. Use the `areaarea` operator. Display the `pineareaMap` together with `pineMap` by typing:

```
pcr.aguila(pineareaMap, pineMap) <Enter>
```

Note that the maximum value in the legend of `pineareaMap` is the large area without pine trees (which is also calculated by `areaarea`). As a result, some patches are hidden by this somewhat unpractical scale of the legend of `pineareaMap`. To get the areas of the patches of pine trees, click with the mouse on `pineMap` and read the values on `pineareaMap` from the data view.

You can change the cell values in the area without pine trees to 0 with the `ifthenelse operator`. Type:

```
pineare2Map = pcr.ifthenelse(pineMap, pineareaMap, 0) <Enter>
```

Display `pineMap` and `pineare2Map` and use the mouse.

---

**Question 11:** What is the data type of `pineare2Map`?

a) Nominal

b) Ordinal

c) Boolean

d) Scalar

Correct answer: d

Feedback: It gives the area, which is a continuous (floating point) value, so it has a scalar data type.

---

Now you can find the answer to the study problem. Use the `pineare2Map` to find contiguous areas with pines that are larger than 4 hectares. Use the > operator in an operation with `pinare2Map`. Note that each cell is 50 x 50 m2.

---

**Question 12:** How many contiguous patches (clumps) exist greater than 4 hectares?

a) 1

b) 2

c) 8

d) 7

Correct answer: b

Feedback: None

```
pineMap = treesMap == 1
pineclumMap = pcr.clump(pineMap)
pineareaMap = pcr.areaarea(pineclumMap)
pineare2Map = pcr.ifthenelse(pineMap, pineareaMap, 0)
contMap = pineare2Map > (4 * 100 * 100)
```

---

## 1.4 Neighbourhood operations: windows, frictions paths, visibility analysis

### 1.4.1 Introduction

Neighbourhood operations relate a cell to its neighbours. The value of each cell is changed on basis of a relation with neighbouring cells or flow of material (for instance water) from neighbouring cells. A rich suite of neighbourhood operators is available in PCRaster. This section introduces you to the window operators, operators for calculating distances over a map (spreading) and operators for visibility analysis. The next section covers neighbourhood operators for catchment analysis.

### 1.4.2 Direct neighbourhood operations: window operations

The window operators calculate statistical values (for instance average value, maximum value) of cells within a window that moves over the map. For each cell a new value is calculated on basis of the cell values within a square window around the cell. A wide range of window operators is available, but in the exercises only the windowaverage and windowdiversity will be used.

The map of the unsaturated zone (`unsatMap`) is rather 'noisy' and there are relatively many spikes and holes. This may be partly due to the interpolation routines used to create the surface elevation map and the phreatic level map. To create a smoother map from the `unsatMap` the `windowaverage` operator can be used.

The syntax of the windowaverage operator is: `Result = pcr.windowaverage(expression, windowlength)` where `expression` is the input map of scalar data type. The `windowlength` defines the size of the window used, it is the length of the square window in the distance units used on the maps.

The next `windowaverage` operation requires the `unsatMap`, created in a previous section. If you have not yet created it in your current session, recreate it by typing:

```
unsatMap = topoMap - phreaticMap <Enter>
```

Then, type:

```
unsat150Map  = pcr.windowaverage(unsatMap, 150) <Enter>
unsat250Map  = pcr.windowaverage(unsatMap, 250) <Enter>
```

---

**Question 13:** What is the size of the window, counted in number of cells, used for the calculation of `unsat150Map`? Keep in mind that the size of one cell is equal to 50 by 50 metres!

a) 1 cell by 1 cell

b) 2 cells by 2 cells

c) 3 cells by 3 cells

d) 4 cells by 4 cells

Correct answer: c

Feedback: The length of one cell is 50, so using a windowlength map of 150, the operation will use a 3 x 3 cell window.

---

---

**Question 14:** How does the windowaverage operator affect the frequency distribution of the values in the maps?

a) The windowaverage operator decreases the variation in the map and thus the frequency distribution. The larger the window, the more the variation is diminished.

b) The windowaverage operator increases the diversity in the map and thus enlarges the frequency distribution. The larger the window, the more the variation is enlarged.

c) The windowaverage operator does not affect the frequency distribution at all, since it originated from the data retrieved from the input map.

Correct answer: a

Feedback: None

---

The spatial diversity of an area can be studied with the windowdiversity operator. It has a syntax that corresponds with the windowaverage operator, but the input expression must have a data type boolean, nominal or ordinal. Try:

```
soidi150Map = pcr.windowdiversity(soilsMap, 150) <Enter>
```

Also try it for different window lengths, e.g. 500.

---

**Question 15:** Explain the operation performed by the windowdiversity operator.

a) The windowdiversity operator finds the number of different cell values within a window and assigns this number to the cell for the result.

b) The windowdiversity operator determines the diversity within a window by calculating the differences between the cell values in that window.

c) The windowdiversity operator calculates the standard deviation per window and assigns that value to the cell.

Correct answer: a

Feedback: None

---

### 1.4.3 Entire neighbourhood operations: absolute distance calculation

The spread operator is used to calculate for each cell the shortest distance to non zero cell values on a boolean, nominal or ordinal map. This distance may be absolute (the real distance) or relative (taking into account frictions). Both kind of distances can be calculated with the same spread operator.

First, calculation of the absolute distance will be explained. The next section covers relative distances.

The syntax of the spread operator is: `Result = pcr.spread(pointsexpression, initialdist, friction)` where `pointsexpression` is a map of data type boolean, nominal or ordinal. The `initialdist` and `friction` are meant for calculation of relative distances (see next section), if set to 0 and 1 respectively the absolute distance will be calculated, like it will be done in this section.

The `wellsMap` is a boolean map with wells used for drink water supply in the area. Display it. Now, try:

```
welldistMap = pcr.spread(wellsMap, 0, 1)  <Enter>
```

And display inputs and outputs.

Protection zones are to be designated in the area surrounding wells used for drinking water supply. Create a boolean map (call it `wellprotMap`) that is TRUE in the area within 200 m distance (excluding a distance of 200 m itself) from wells and FALSE in the area further away from wells.

---

**Question 16:** For how many wells do their protection zones overlap (or are connected)?

a) 2

b) 11

c) 4

d) 7

Correct answer: a

Feedback: None

```
welldistMap = pcr.spread(wellsMap, 0, 1)
wellprotMap = welldistMap < 200
```

---

### 1.4.4 Entire neighbourhood operations: relative distance calculation

In the previous section, the friction map in the spread operator was set to 1 to obtain the absolute distance. By defining a different friction value or a map with friction values, friction can be included in the calculation and relative distances (for instance time) can be calculated. For instance, assume that the water is transported from the wells by a waterworks network (Dutch 'waterleiding netwerk'). On average, it takes 3 seconds for the water to be transported one metre. A map with the total travel time for the water from the nearest well to each cell is calculated as follows:

```
welltimeMap = pcr.spread(wellsMap, 0, 3) <Enter>
```

Display `welltimeMap`. It gives for each cell the time needed for the water to reach the cell under consideration (from the nearest well). The spread operator has multiplied the distance (in metres) with the time needed to move one metre.

## 1.5 Neighbourhood operations: DEM and catchment analysis

### 1.5.1 Introduction

A neighbourhood operation relates the cell to its neighbours. The value of each cell is changed on basis of some kind of relation with neighbouring cells or flow of material (for instance water) from neighbouring cells. This chapter will cover neighbourhood operators for catchment analysis.

### 1.5.2 Creating slope and aspect maps

The slope operator generates a slope map on basis of a digital elevation model (a fraction: increase in height per distance in horizontal direction, in a 3 x 3 cells window). Try:

```
slopeMap = pcr.slope(topoMap) <Enter>
```

Display `slopeMap` and `topoMap`.

With the atan operator, the slope given as an angle is calculated. Type:

```
slopedegMap = pcr.atan(slopeMap) <Enter>
```

Apply the operator `slope` to `slopeMap`. Type:

```
slope2Map = pcr.slope(slopeMap) <Enter>
```

The `slope2Map` gives the change in slope (second derivative).

The aspect operator results in a map of slope aspect in 360 degrees (clockwise, North is to the top of the map and is 0 degrees). Type:

```
aspectMap = pcr.aspect(topoMap) <Enter>
```

### 1.5.3 Creating the local drain direction map

In different kinds of environmental studies, such as erosion, surface runoff and hydrological research, it is important to be able to delineate a river course and drainage area. Digital Elevation Models (such as `topoMap`) offer the possibility for automated extraction of catchment characteristics by creating a local drain direction map which gives the flow pattern over a DEM.

The lddcreate operator is used to generate a local drain direction map. Its syntax is: `Result = lddcreate(dem, outflowdepth, corevolume, corearea, precipitation)` where `dem` is the digital elevation model. The other input maps (or constant values) are so-called pit removing thresholds, these are explained later.

First, set the pit removing thresholds to zero:

```
ldd0Map = pcr.lddcreate(topoMap, 0, 0, 0, 0) <Enter>
```

Display `topoMap` and `ldd0Map`.

The lddcreate operator has determined for each cell the direction of the steepest (downhill) slope, which is the direction of local drainage. For each cell, the drain direction is assigned to the local drain direction map `ldd0Map`.

The cells with a black square at the edge of the map represent outflow points from the map (you may need to zoom in a bit to see them). There are two similar cells `ldd0Map` that are not at the edge. These are called pits. A pit cell is surrounded by cells at a greater elevation than the pit cell itself. As result, a pit cell cannot drain to a neighbouring cell. In PCRaster, the cells can be assigned unique nominal values with the pit operator. Try:

```
pit0Map = pcr.pit(ldd0Map) <Enter>
```

Display the maps, including `topoMap` (in one `aguila` command). Enlarge the window of `pit0Map` and check the elevations at and around pit cells by clicking with the mouse on these cells and reading the topoMap values from the data window.

You can find the downstream paths over a local drain direction map with the path operator. It uses an boolean input map. All cells that are downstream of a TRUE cell on the boolean map are assigned a boolean TRUE on the resulting map.

The map `pointsMap` contains some arbitrary points with a boolean value TRUE. Display `pointsMap`. After that, try:

```
pathzeroMap = pcr.path(ldd0Map, pointsMap) <Enter>
pcr.aguila(ldd0Map, pointsMap, pit0Map, pathzeroMap) <Enter>
```

Compare `ldd0Map` and `path0Map`. You will see that the paths stop at the pit cells on the local drain direction map.

Two kinds of pits may occur in a DEM. The first kind is caused by natural depressions and sinkholes in a landscape, see the Figure below.

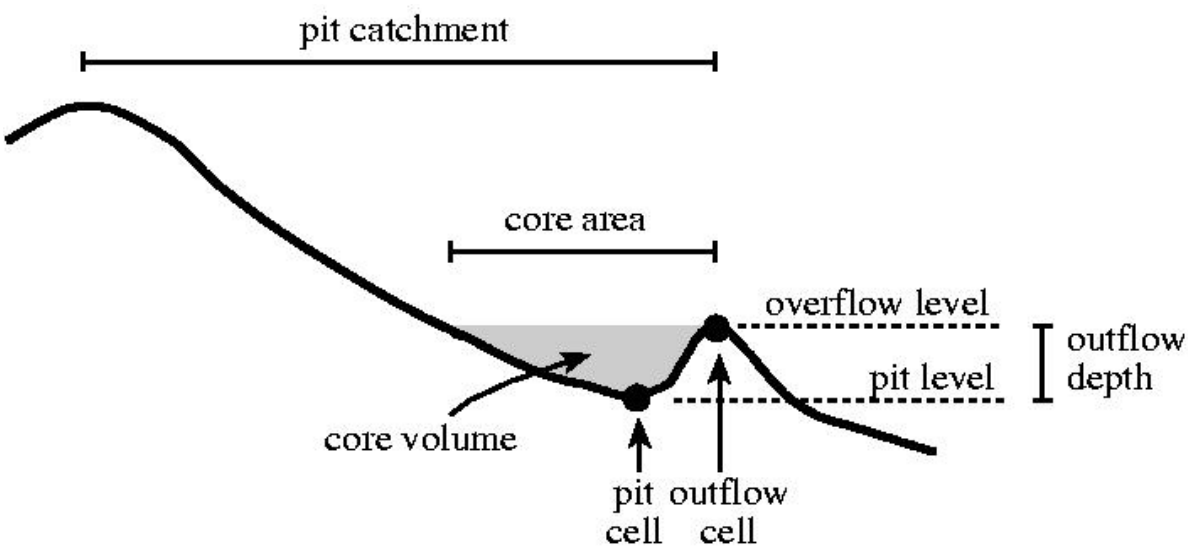


**Figure:** A sprinkling of the definitions used in theory of pit removing.

The pit will be at the cell with the smallest elevation in such a depression. The core of the pit is the area that will contain water if it would be filled with water until the overflow level is reached. The second kind of pits is introduced by the DEM itself. Due to the discretization of the elevation surface, the DEM is not an exact description of the landscape. As a result pits may occur on the DEM at locations without any real depressions in the landscape. These will have small and shallow cores on the DEM.

You can fill up the cores of all pits in a digital elevation model with the `lddcreatedem` operator. It generates a modified digital elevation model: the cores are filled up until the overflow level of the cores is reached. This can be compared with fluviatile sedimentation in the core depression until a maximum sedimentation level is reached (which is the overflow level of the pit). Type:

```
topomodiMap = pcr.lddcreatedem(topoMap, 1e31, 1e31, 1e31, 1e31)   <Enter>
```

The `1e31` values effect that all cores will be filled up. To find the difference between topoMap and topomodiMap (which is the depth of the pit cores), type:

```
coredeptMap = topomodiMap - topoMap <Enter>
```

---

**Question 17:** What is the maximum depth of the largest core in the map (in metres)?

a) 1.2

b) 4.0

c) 0.1

d) 3.0

Correct answer: d

Feedback: None

---

For further analysis it is necessary to remove the pits from the local drain direction map. This can be done by changing the pit remover thresholds given in the `lddcreate` operator. These are: the depth of the pit, `outflowdepth`; the volume or area of the pit core, `corevolume` and `corearea` respectively and the amount of rain needed to fill up the pit core, `precipitation`. With these thresholds you can define which pits must be removed from the map. Pits that have dimensions smaller than the threshold dimensions will be removed, pits that are larger than one (or more) of the thresholds remain in the local drain direction map. In this exercise, the pit depth (outflowdepth) will be used. Try:

```
ldd2Map = pcr.lddcreate(topoMap, 2, 1e31, 1e31, 1e31) <Enter>
```

This operation removes all pits except these with a depth that is larger than or equal to 2 metre. To find the pits in `ldd2Map`, type:

```
pit2Map = pcr.pit(ldd2Map) <Enter>
```

To remove all the pits, also the ones on the sides of the map, using very high threshold values (here 1e31, i.e. 10 to the power of 31), type:

```
lddMap = pcr.lddcreate(topoMap, 10, 1e31, 1e31, 1e31) <Enter>
```

Calculate the downstream paths and display:

```
pathMap = pcr.path(lddMap, pointsMap) <Enter>
```

You will see that all paths end at an outflow cell. For further analysis, `lddMap` will be used.

### 1.5.4 Catchment analysis and routing with the ldd map.

The catchments of all outflow points are determined with the catchment operator.

Use the pit operator to create a nominal map with the outflow points (call it `outpoMap`) from the study area, uniquely numbered. Use `lddMap` which you created in the previous exercise. Now, use `catchment`, type:

```
catchmsMap = pcr.catchment(lddMap, outpoMap) <Enter>
```

For each non zero cell value on `outpoMap` its catchment has been determined and all cells in its catchment have been assigned this non zero value on catchmsMap.

The main use of the local drain direction map is for routing of surface water in downstream direction. The accuflux operator transports an input of material (for instance rain) downstream over the local drain direction network to the

outflow cells. It calculates for each cell the material flux over the ldd during transport. All material is transported. The syntax of accuflux is: `Result = accuflux(ldd,material)` where `ldd` is the local drain direction map and `material` is a map of scalar data type that contains the input of material.

The map `rainstorMap` gives the amount and distribution of rain ($m^3$ per cell) that falls during a short rainstorm. Display the map. The discharge in the study area as a result of the rainstorm is calculated with the `accuflux` operator. It is assumed that no baseflow, no interception and no infiltration occurs. Type:

```
dischMap = pcr.accuflux(lddMap, rainstorMap) <Enter>
```

Display `dischMap`, `rainstorMap` and `lddMap`.

---

**Question 18:** What is the maximum value of the total discharge ($m^3$) as a result of the thunderstorm?

a) 1179.69

b) 12.69

c) 117969

d) 1.17969

Correct answer: a

Feedback: None

---

By taking the base10 logarithm of dischMap you can get a better picture of the spatial pattern of the discharge. Type:

```
dischlogMap = pcr.log10(dischMap) <Enter>
```

Alternatively, you could change the view of `dischMap` in Aguila, right click on its legend, select 'Edit draw properties' and change colour assignment to True logarithmic. You may need to set the minimum cutoff to 0.1 as log(0) is not possible.

The local drain direction map can also be used for the calculation of the upstream area map. The upstream area map contains for each cell the area of the cell its catchment (including the cell itself) that drains to the cell. The map is calculated with the `accuflux` operator. Instead of defining a material input for the material map in the operation, a map that contains for each cell the area of a cell (2500 $m^2$) is used for material. As a result, this area 'drains downstream' and the catchment area of each cell is calculated. Try:

```
upareMap = pcr.accuflux(lddMap, 2500)
```

Display `upareMap` and `lddMap`.

CHAPTER

TWO

# DYNAMIC MODELLING

Download this website as pdf.

Download this website as epub (for e-readers).

To subscribe to our courses visit http://www.pcraster.eu

## 2.1 Visualisation of spatio-temporal data

### 2.1.1 Static data

PCRaster includes the Aguila software for visualisation of spatio-temporal stochastic data. With Aguila, you can easily explore large multi-dimensional data sets. For the exercises below, you will need to look up some options in the Aguila manual, available at https://pcraster.geo.uu.nl/pcraster/latest/documentation/pcraster_aguila/index.html.

You will use the data from an Alpine area that will also be used for the construction of the distributed model in the dynamic modelling exercises.

Aguila is started from a command shell. On Linux or Mac this would be a standard terminal window, on Microsoft Windows you need to open the command prompt. Open a command shell and go to the directory containing your data for the exercises. Type cd to change directories. In the directory, you will find the folder `snowmelt` containing the data for the visualisation practical. In this directory, display the digitial elevation model and another map with Aguila:

```
aguila dem.map anArea.map <Enter>
```

Use the menu items (or right click on the legend to get more options) to zoom in/out, to change the color palette, and to change the drawing mode to contours. Also try changing the number of classes shown (or number of contours).

---

**Question 1:** What is shown on `anArea.map`?

a) True cells represent the higher part of the study area.

b) Idem, lower part.

Correct answer: a

Feedback: None

---

You can also combine multiple maps in one view:

```
aguila anArea.map + dem.map <Enter>
```

The second map is draped over the first map. Make `dem.map` transparent by changing its display mode to contours. Now you are able to see both maps in one view.

### 2.1.2 Temporal spatial data

Temporal spatial data can be animated in PCRaster. The dataset contains a timeseries of maps with the amount of precipitation (m/day) in the area. It consists of 181 maps, each map contains precipitation for one day. The first map represents July 1st. Type `dir <Enter>` (or `ls` if you are on Unix) to get a list with all files in your dataset. You will see it contains the files `precip00.001` up to `precip00.181`. These are the precipitation maps. To display them, type:

```
aguila --timesteps=[1,181,1] precip <Enter>
```

Animate the maps by clicking on the yellow menu item at the top of the view. Create a time series plot by right clicking on the legend, selecting 'Show time series'. The time series is plotted for the current cursor position on the map (so you can get a time series for any location, try it!).

Now, type:

```
aguila --timesteps=[60,181,10] precip <Enter>
```

Animate. It has opened a subset of maps now, i.e. map 60, 70, 80,.. up to 180. So the first value (60) that you provide between the square brackets is the first time step to be shown, the second value (181) is the last step, and the third value (10) is the number of time step between the maps that are shown.

---

**Question 2:** Compare the precipitation pattern with the elevation map (open them in one view, using a single Aguila command). What is the relation between precipitation and elevation?

a) There is no relation.

b) The higher the elevation, the higher the precipitation.

c) The lower the elevation, the higher the precipitation.

Correct answer: b

Feedback: None

---

### 2.1.3 Temporal non-spatial data

You can also use observed timeseries of data in Aguila, e.g. observed precipitation at a meteo station. Such data are non-spatial (single location, or sometimes multiple locations). Have a look at the contents of a rainfaill timeseries included in the data set by typing:

```
type precip.tss
```

(on a Unix system you will use `less`). The time series file is an ASCII file (just like for instance python programs being ASCII files) and thus can be displayed with the standard command `type`. You could also open it in an ASCII editor. The first column contains the time steps numbers (here each time step represents a day), the second column the observed value (here: precipitation in m/day). Display the timeseries by typing:

```
aguila precip.tss
```

## 2.2 The dynamic modelling framework

### 2.2.1 The dynamic modelling class

For dynamic (temporal) modelling, PCRaster comes with a predefined class. It will make it relatively simple to do time iterations, and to define inputs and outputs of a model. Open the file `dynMod.py` in your editor, on Microsoft Windows

we recommend for instance the standard Python editor (IDLE) that comes with Python (available at Start, Programs, Python, IDLE). The script is the most basic script for dynamic modelling, it does only contain the statements that are needed for the control flow of the program. When you start a project, you can take this template and add statements to construct a model. But first run this 'empty' model.

---

**Question 3:** What does it print?

a) It prints dots; each dot represents 0.1 second run time.

b) Idem, each dot represents a time step.

Correct answer: b

Feedback: None

---

When constructing the model, you will only add statements to the initial and dynamic methods. For more advanced models you may want to add also calculations outside these methods, but for now, you will see that the initial and dynamic methods provide most of what you need.

Now let's see how it executes things that you add to the initial and dynamic. Replace the `pass` statement in the initial with

```
print("running the initial")
```

Replace the `pass` statement in the dynamic with

```
timeStep = self.currentTimeStep()
print(f"running the dynamic for time step: {timeStep}")
```

Save and run the modified model.

---

**Question 4:** What is the order in which the initial and dynamic are executed?

a) The initial and dynamic are executed at the same time.

b) First the initial is executed once. Next, the dynamic is executed 10 times.

c) The initial is not executed, it only runs the dynamic.

Correct answer: b

Feedback: None

---

### 2.2.2 Modelling with feedback

Now, let's do forward modelling with feedback. We have a reservoir that is defined by the differential equation:

dx/dt = -cx

In hydrology, this is known as a linear reservoir. The variable x is the amount of water in the reservoir, and c is a rate constant. The script `feedback.py` solves the linear reservoir equation using an explicit solution. Open the script and run it.

The initial sets the initial content of the reservoir. Here it merely uses an initial value that is converted to another unit, using `conversionValue`. For the variable x in the differential equation, the script uses the variable name `reservoir`.

The dynamic solves the differential equation by subtracting each timestep the `outflow` from the `reservoir`. Note that the variable `reservoir` is preceded by `self.`. This is needed because all other variables, e.g. `conversionValue` are local variables, i.e. they exist only in the method (initial or dynamic) were they have been defined. Preceding a variable with `self.` makes it a member variable of the class. This means it exists (can be used) in all other methods.

Add to the dynamic the line

```
print(conversionValue)
```

and run the model.

---

**Question 5:** What is printed? How can you solve this error if you really want to print it in the dynamic?

a) An error message is printed, indicating there is a typo in the script.

b) An error message is printed, which indicates that `conversionValue` is defined as a local variable. The solution is to use `self.conversionValue` throughout the script.

c) An error message is printed, which indicates that `conversionValue` is defined as a global variable. The solution is to use `self.conversionValue` throughout the script.

Correct answer: b

Feedback: None

---

An extension to the reservoir equation would be to add constant inflow of 0.5:

dx/dt = -cx + 0.5

Modify the script to simulate this inflow.

---

**Question 6:** What did you add to the script? Where did you type it?

a) I added `+ 0.5` to `self.reservoir` in the `initial`.

b) I added `inflow = 0.5` to the `initial` and changed the line in the dynamic to: `self.reservoir = self.reservoir - outflow + inflow`

c) I added `self.inflow = 0.5` to the `initial` and changed the line in the dynamic to: `self.reservoir = self.reservoir - outflow + self.inflow`

Correct answer: c

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfrw


class MyFirstModel(pcrfrw.DynamicModel):
    def __init__(self):
        pcrfrw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
```

(continues on next page)

(continued from previous page)

```
        conversionValue = 3.0
        self.reservoir = 30.0 / conversionValue
        print(f"initial reservoir is: {self.reservoir}")
        self.inflow = 0.5

    def dynamic(self):
        outflow = 0.1 * self.reservoir
        self.reservoir = self.reservoir - outflow + self.inflow
        print(self.reservoir)


nrOfTimeSteps = 100
myModel = MyFirstModel()
dynamicModel = pcrfrw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

**Question 7:** If you run the model for a long time, what happens to the reservoir?

a) The reservoir decreases until it reaches 5.

b) The reservoir increases until it reaches 5.

c) The reservoir decreases until it reaches 3.

d) The reservoir increases until it reaches 3.

Correct answer: a

Feedback: None

---

## 2.3 Reading and writing spatio-temporal data

Python data types (e.g. floating points) can be printed, written to disk, or read from disk using functions that come with standard Python. Think of `print` to print a variable on the screen, or the file objects to read and write data to disk. As these functions do not work with PCRaster maps, PCRaster Python provides its own functions for reading and writing map data.

### 2.3.1 Reading and writing static spatial data

Open the script `dynMod.py` and save it under the name `staticMap.py`. In the initial, add the following lines of code:

```
dem = self.readmap("dem")
slopeOfDem = pcr.slope(dem)
self.report(slopeOfDem, "gradient")
```

Run the model script. The `readmap` is a function to open a map stored on disk, you do not need to give the suffix `.map` of the filename that is opened (`dem.map`). The `report` function writes a map (here, `slopeOfDem`) to disk under a name given as the second argument, which is of data type string. When used in the `initial` (as you did here), it stores a single map, here the filename will be `gradient.map`.

Check the content of `gradient.map` by displaying it.

### 2.3.2 Reading and writing temporal spatial data

A very similar approach can be followed for temporal map data. However now we need to add statements to the `dynamic`. Let's import the time series of precipitation maps that you displayed in the visualisation part of the exercises to the model, convert it to mm/hours and write the result as a timeseries of maps.

Open `staticMap.py` created in the previous section and save it as `dynamicMap.py`. In the dynamic section:

- Remove `pass`.
- Add a statement that reads the precipitation from disk using `self.readmap("precip")`.
- Add a statement converting from m/day to mm/day.
- Write the result to disk using `self.report` (syntax is the same as in the `initial`). As second argument of the `report` function, use `pmm`.

Also, change the number of timesteps that the model is run to 181.

Save and run the script. In the command shell, type `dir` (or `ls` on unix) to see what is stored. If everything is OK, you should see the filenames `pmm00000.001`, `pmm00000.002`, etc.

---

**Question 8:** Use aguila to animate the original precipitation timseries of maps that you read from disk, together with the same precipitation converted to mm/day (one Aguila command). Check the results. What command did you use?

a) `aguila --timesteps=[1,181,1] precip pmm`

b) `aguila --timesteps=1,181,1 precip pmm`

c) `aguila -timesteps=[1,181,1] precip pmm`

d) `aguila --timesteps=[1,181,1] precip.tss pmm.tss`

Correct answer: a

Feedback: None

---

Now, try to make an animation of maps that shows where precipitation is above 0.01 m/day. Add a comparison operation to the dynamic that calculates a Boolean map with True where cells are above the threshold and False elsewhere. If needed use the PCRaster man pages. Store the output on harddisk using `report`. Run the model and animate the results with Aguila.

---

**Question 9:** At what elevation do we find particularly high values of precipitation?

a) There is no relation with elevation.

b) Large precipitation is mainly found in the valleys.

c) Precipitation is zero throughout the area.

d) Precipitation is high in the higher areas.

Correct answer: d

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
        dem = self.readmap("dem")
        slopeOfDem = pcr.slope(dem)
        self.report(slopeOfDem, "gradient")

    def dynamic(self):
        precipitation = self.readmap("precip")
        precipitationMMPerHour = precipitation * 1000.0
        self.report(precipitationMMPerHour, "pmm")
        highPrecipitation = precipitation > 0.01
        self.report(highPrecipitation, "high")


nrOfTimeSteps = 181
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

### 2.3.3 Reading timeseries

Timeseries files can be imported to a script using the `timeinput..` function. As most PCRaster functions operate on maps, the `timeinput..` function reads the value(s) from the time series for the current time step, and directly converts it to a PCRaster map that can be used in calculations. Let's read from disk the precipitation timeseries file that you had a look at in the visualisation section of the exercises.

Open `dynMod.py` and save it as `dynamicTimeSeries.py`. Change the number of time steps to 181 and replace the `pass` in the dynamic with the line:

```
precipitation = pcr.timeinputscalar("precip.tss", 1)
```

As the `precip.tss` does not contain information on the data type of the data in the file, you need to provide this with the timeinput command. Precipitation are scalar data, so we use `timeinputscalar`. In case of nominal data, for instance, you would use `timeinputnominal`. The first input argument is the timeseries that is read. The second argument is the area to which the values in the timeseries is assigned. Here we want to assign the values in the timeseries value to the whole map, so we provide a *1* as second argument. This should be read as a map completely filled with one values. In case the timeseries contains multiple columns, i.e. multiple locations, you need to enter a map with regions as second argument. See the PCRaster man page on `timeinput..` for the details.

Add a `report` statement that writes `precipitation` to disk. Animate the results with Aguila.

---

**Question 10:** What is the content of the resulting timeseries?

a) The precipitation value from the `.tss` file is uniformly assigned to the whole map for each time step.

b) It contains a value one for each time step.

c) The precipitation is distributed according to the elevation.

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
        pass

    def dynamic(self):
        precipitation = pcr.timeinputscalar("precip.tss", 1)
        self.report(precipitation, "pFromTss")


nrOfTimeSteps = 181
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

## 2.4 Point operations: a snow melt model

### 2.4.1 Precipitation and temperature

In this exercise you will use the dataset introduced in the visualisation section to build a simple spatial hydrological model, including rainfall, snowfall, snowmelt, and runoff.

Open `dynamicTimeSeries.py` created in the previous exercises, save it under a new name, `temp.py`. It reads the precipitation timeseries from disk. Add a statement that reads the temperature timeseries `temp.tss` from disk and assigns the temperature values (degrees Celsius) to the map variable `temperatureObserved`. Write the results to disk, run the model, and have a look at the output.

---

**Question 11:** Which function did you use to read the temperature from disk?

a) `open`

b) `timeinputscalar`

c) `timeinputnominal`

d) `timeinput`

e) `report`

Correct answer: b

Feedback: In the dynamic (somewhere at the top), the following two lines are needed: `temperatureObserved = timeinputscalar('temp.tss',1) self.report(temperatureObserved,'tempObs')`

---

The temperature timeseries contains the temperature observed at a meteostation that is close by. However, temperature will be spatially variable, as it depends on the surface elevation. The temperature $t_i$ at a grid cell $i$ can be calculated as:

$t_i = t_{obs} - l\ (e_i - e_{obs})$

In the equation, $t_{obs}$ is the temperature observed at the meteostation, $e_i$ is the elevation at a grid cell, $e_{obs}$ is the elevation at the meteo station (here: 2058.1 m) and $l$ is the temperature lapse rate (here: 0.005).

Add statements to your script `temp.py` to calculate a map (call it `temperatureCorrection`) containing the term $l\ (e_i - e_{obs})$. Write the map to disk and check the results.

---

**Question 12:** Where did you add the statements to calculate `temperatureCorrection`?

a) In the `__init__`.

b) In the `initial`.

c) In the `dynamic`.

d) In the `initial` and in the `dynamic`

Correct answer: b

Feedback: As the temperature correction remains constant over time, it needs to be calculated only once, at the start of the model run. This is why it should be put in the initial. We use a variable name starting with self, because later on the variable will be used in the dynamic. Without the self, it would be a local variable that is not available outside the initial method.

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
        dem = self.readmap("dem")
        elevationMeteoStation = 2058.1
        elevationAboveMeteoStation = dem - elevationMeteoStation
        temperatureLapseRate = 0.005
        self.temperatureCorrection = elevationAboveMeteoStation * temperatureLapseRate
        self.report(self.temperatureCorrection, "tempCor")

    def dynamic(self):
        precipitation = pcr.timeinputscalar("precip.tss", 1)
        self.report(precipitation, "pFromTss")
        temperatureObserved = pcr.timeinputscalar("temp.tss", 1)
        self.report(temperatureObserved, "tempObs")
```

(continues on next page)

(continued from previous page)

```
nrOfTimeSteps = 181
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
```

---

**Question 13:** How much colder is it at the highest point in the area compared to the temperature at the meteostation (i.e., at 2058.1 m)?

a) About 2 degrees.

b) About 12 degrees.

c) About 7 degrees.

d) There is no difference.

Correct answer: c

Feedback: Use aguila to animate the time series of maps with the temperature correction. Select any time step. Open the data value window by clicking on the cross and click on the map to retrieve cell values.

---

As a next step, calculate and write to disk a time series of maps containing $t_i$. Save the script (`temp.py`), run and display the new output.

We assume that precipitation falls as snow when $t_i$is below zero. Add statements to calculate and store a time series of maps (use the variable name `freezing`) that is True at a cell if the temperature is below zero degrees and False elsewhere. Use a comparison operator. Run the script and check the results.

Create two time series of maps containing respectively snowfall (m/day, variable name `snowFall`) and rainfall (m/day, variable name `rainFall`). You will need the operations `ifthenelse` and a Boolean operator. Write them to disk and display the results.

---

**Question 14:** Which comparison operator did you use, and where?

a) ==, in the `dynamic`

b) `or`, in the `dynamic`

c) `while`, in the `initial`

d) > or < or >= or <=, in the `dynamic`

Correct answer: d

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
        dem = self.readmap("dem")
        elevationMeteoStation = 2058.1
        elevationAboveMeteoStation = dem - elevationMeteoStation
        temperatureLapseRate = 0.005
        self.temperatureCorrection = elevationAboveMeteoStation * temperatureLapseRate
        self.report(self.temperatureCorrection, "tempCor")

    def dynamic(self):
        precipitation = pcr.timeinputscalar("precip.tss", 1)
        self.report(precipitation, "pFromTss")
        temperatureObserved = pcr.timeinputscalar("temp.tss", 1)
        self.report(temperatureObserved, "tempObs")

        temperature = temperatureObserved - self.temperatureCorrection
        self.report(temperature, "temp")

        freezing = temperature < 0.0
        self.report(freezing, "fr")
        snowFall = pcr.ifthenelse(freezing, precipitation, 0.0)
        self.report(snowFall, "snF")
        rainFall = pcr.ifthenelse(pcr.pcrnot(freezing), precipitation, 0.0)
        self.report(rainFall, "rF")


nrOfTimeSteps = 181
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

### 2.4.2 The snow store

To simulate snow melt in each cell, a snow store is used containing a snow pack $a_i$(t) (m water equivalent) that changes over time:

$a_i$(t) = $a_i$(t-1) + $s_i$(t) - $b_i$(t)

In the equation above, (t) refers to the current time step and (t-1) to the previous time step (day). The variable $s_i$(t) is snowfall and $b_i$(t) is snowmelt. Actual snowmelt $b_i$(t) is modelled as:

$b_i$(t) = min($a_i$(t), $b_{p,i}$(t))

with, $b_{p,i}$(t), potential snowmelt. Potential snowmelt is:

$b_{p,i}$(t) = $mt_i$, for $t_i$ > 0

$b_{p,i}$(t) = 0, for $t_i$ <= 0

with $m$, a degree day factor (here: 0.01), and $t_i$ , the corrected termperature (see above).

Add this snow store in a stepwise manner to the model, in the following way. Let's first assume there is no snow melt. Open `temp.py` and save it as `snow.py`. Add the variable `snow` representing the snow store $a_i$, assuming there is no

snow at the start of the model run (as it is end of summer), and increasing the snowstore with snowfall in each time step.

---

**Question 15:** Without snow melt (as was assumed here), what is the maximum snowpack thickness (m water equivalent) found in the area at the end of the model run? Idem, the minimum thickness?

a) Maximum thickness: 0.11 m, minimum: 0.01 m

b) Maximum thickness: 1.12 m, minimum: 0.15 m

c) Maximum thickness: 0.85 m, minimum: 0.00 m

d) Maximum thickness: 1.96 m, minimum: 0.62 m

e) Maximum thickness: 0.57 m, minimum: 0.01 m

Correct answer: e

Feedback: You can get the answer by clicking on the map or by typing: `mapattr -p st000000.181`

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
        dem = self.readmap("dem")
        elevationMeteoStation = 2058.1
        elevationAboveMeteoStation = dem - elevationMeteoStation
        temperatureLapseRate = 0.005
        self.temperatureCorrection = elevationAboveMeteoStation * temperatureLapseRate
        self.report(self.temperatureCorrection, "tempCor")

        self.snow = 0.0

    def dynamic(self):
        precipitation = pcr.timeinputscalar("precip.tss", 1)
        self.report(precipitation, "pFromTss")
        temperatureObserved = pcr.timeinputscalar("temp.tss", 1)
        self.report(temperatureObserved, "tempObs")

        temperature = temperatureObserved - self.temperatureCorrection
        self.report(temperature, "temp")

        freezing = temperature < 0.0
        self.report(freezing, "fr")
        snowFall = pcr.ifthenelse(freezing, precipitation, 0.0)
        self.report(snowFall, "snF")
        rainFall = pcr.ifthenelse(pcr.pcrnot(freezing), precipitation, 0.0)
        self.report(rainFall, "rF")
```

(continues on next page)

(continued from previous page)

```
        self.snow = self.snow + snowFall
        self.report(self.snow, "snow")


nrOfTimeSteps = 181
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
```

---

Add snowmelt to the model. First, calculate a variable `potentialMelt`, containing $b_{p,i}(t)$. Write to disk and check the results. Second, calculate actual snowmelt $b_i(t)$. Check the results.

---

**Question 16:** How did you calculate actual snowmelt?

a) It is equal to the potential snow melt in this case.

b) It is a fraction of the potential melt.

c) It depends on the potential melt and the actual snow thickness.

d) It is constant, so it needs to be calculated in the `initial`.

Correct answer: c

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
        dem = self.readmap("dem")
        elevationMeteoStation = 2058.1
        elevationAboveMeteoStation = dem - elevationMeteoStation
        temperatureLapseRate = 0.005
        self.temperatureCorrection = elevationAboveMeteoStation * temperatureLapseRate
        self.report(self.temperatureCorrection, "tempCor")

        self.snow = 0.0

    def dynamic(self):
        precipitation = pcr.timeinputscalar("precip.tss", 1)
        self.report(precipitation, "pFromTss")
        temperatureObserved = pcr.timeinputscalar("temp.tss", 1)
        self.report(temperatureObserved, "tempObs")

        temperature = temperatureObserved - self.temperatureCorrection
        self.report(temperature, "temp")
```

(continues on next page)

(continued from previous page)

```
        freezing = temperature < 0.0
        self.report(freezing, "fr")
        snowFall = pcr.ifthenelse(freezing, precipitation, 0.0)
        self.report(snowFall, "snF")
        rainFall = pcr.ifthenelse(pcr.pcrnot(freezing), precipitation, 0.0)
        self.report(rainFall, "rF")

        self.snow = self.snow + snowFall

        potentialMelt = pcr.ifthenelse(pcr.pcrnot(freezing), temperature * 0.01, 0)
        self.report(potentialMelt, "pmelt")
        actualMelt = pcr.min(self.snow, potentialMelt)
        self.report(actualMelt, "amelt")
        self.report(self.snow, "snow")


nrOfTimeSteps = 181
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
```

---

And as the third step, update the snow store by subtracting actual snowmelt from the store. Write the snow store to disk (directly after subtracting actual snowmelt). Save the model under the same name `snow.py`.

---

**Question 17:** From what time step on is the main valley in the top-right corner snow-free, in spring? Idem, for the valley in the top-left corner?

a) Top-right corner: day 128. Top-left corner: day 130.

b) Top-right corner: day 110. Top-left corner: day 110.

c) Top-right corner: day 112. Top-left corner: day 140.

d) Top-right corner: day 92. Top-left corner: day 118.

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
        dem = self.readmap("dem")
        elevationMeteoStation = 2058.1
        elevationAboveMeteoStation = dem - elevationMeteoStation
```

(continues on next page)

(continued from previous page)

```
        temperatureLapseRate = 0.005
        self.temperatureCorrection = elevationAboveMeteoStation * temperatureLapseRate
        self.report(self.temperatureCorrection, "tempCor")

        self.snow = 0.0

    def dynamic(self):
        precipitation = pcr.timeinputscalar("precip.tss", 1)
        self.report(precipitation, "pFromTss")
        temperatureObserved = pcr.timeinputscalar("temp.tss", 1)
        self.report(temperatureObserved, "tempObs")

        temperature = temperatureObserved - self.temperatureCorrection
        self.report(temperature, "temp")

        freezing = temperature < 0.0
        self.report(freezing, "fr")
        snowFall = pcr.ifthenelse(freezing, precipitation, 0.0)
        self.report(snowFall, "snF")
        rainFall = pcr.ifthenelse(pcr.pcrnot(freezing), precipitation, 0.0)
        self.report(rainFall, "rF")

        self.snow = self.snow + snowFall

        potentialMelt = pcr.ifthenelse(pcr.pcrnot(freezing), temperature * 0.01, 0)
        self.report(potentialMelt, "pmelt")
        actualMelt = pcr.min(self.snow, potentialMelt)
        self.report(actualMelt, "amelt")

        self.snow = self.snow - actualMelt
        self.report(self.snow, "snow")


nrOfTimeSteps = 181
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
```

### 2.4.3 Runoff generation

Runoff generated in each cell (m/day) is the sum of the rainfall in that cell and the snowmelt. Add a statement to `snow.py` calculating a time series of maps containing the total amount of runoff generated. Write results to disk and evaluate the results.

**Question 18:** In what time of the year is the largest amount of runoff generated?

a) Melting snow generates a large amount of runoff around time step 70.

b) Melting snow generates a large amount of runoff around time step 90.

c) Melting snow generates a large amount of runoff around time step 110.

d) Melting snow generates a large amount of runoff around time step 130.

Correct answer: d

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
        dem = self.readmap("dem")
        elevationMeteoStation = 2058.1
        elevationAboveMeteoStation = dem - elevationMeteoStation
        temperatureLapseRate = 0.005
        self.temperatureCorrection = elevationAboveMeteoStation * temperatureLapseRate
        self.report(self.temperatureCorrection, "tempCor")

        self.snow = 0.0

    def dynamic(self):
        precipitation = pcr.timeinputscalar("precip.tss", 1)
        self.report(precipitation, "pFromTss")
        temperatureObserved = pcr.timeinputscalar("temp.tss", 1)
        self.report(temperatureObserved, "tempObs")

        temperature = temperatureObserved - self.temperatureCorrection
        self.report(temperature, "temp")

        freezing = temperature < 0.0
        self.report(freezing, "fr")
        snowFall = pcr.ifthenelse(freezing, precipitation, 0.0)
        self.report(snowFall, "snF")
        rainFall = pcr.ifthenelse(pcr.pcrnot(freezing), precipitation, 0.0)
        self.report(rainFall, "rF")

        self.snow = self.snow + snowFall

        potentialMelt = pcr.ifthenelse(pcr.pcrnot(freezing), temperature * 0.01, 0)
        self.report(potentialMelt, "pmelt")
        actualMelt = pcr.min(self.snow, potentialMelt)
        self.report(actualMelt, "amelt")

        self.snow = self.snow - actualMelt

        runoffGenerated = actualMelt + rainFall
        self.report(runoffGenerated, "rg")

        self.report(self.snow, "snow")
```

(continues on next page)

(continued from previous page)

```
nrOfTimeSteps = 181
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

## 2.5 Neighbourhood operations with defined topology: the snow melt model

### 2.5.1 The local drain direction map

In the previous exercise you created a snowmelt model. Your model calculated for each time step (day) the total amount of runoff that was generated in a cell. Here, you will transport this runoff downstream. One approach is to assume that runoff occurs towards a neighbouring cell with the steepest downhill path. These directions are stored in a so-called local drain direction map.

Open `snow.py` and save it as `runoff.py`. Add a `lddcreate` statement calculating the local drain direction map. Run the model and have a look at the local drain direction map.

---

**Question 19:** Where in the script did you enter the `lddcreate` function? Explain.

a) At the top.

b) In the `__init__`.

c) In the `initial`.

d) In the `dynamic`

Correct answer: c

Feedback: You need the following lines in the `initial`. The lddcreate function is added to the initial section, because it needs to be calculated only once, at the start. It needs to be defined as a global variable to be later used in the dynamic section for routing of water, so this is why a variable name starting with `self.` is used.

```
        self.ldd = pcr.lddcreate(dem, 1e31, 1e31, 1e31, 1e31)
        self.report(self.ldd, "ldd")
```

---

### 2.5.2 Drain all runoff within one time step: the accuflux function

When the size of a study area is sufficiently small and the time step duration sufficiently long that it can be assumed that all runoff generated in a time step reaches the outflow point in the same time step, runoff can be transported downstream using an *accu* function.

Add an `accuflux` statement to `runoff.py` calculating discharge for each grid cell. You need to calculate discharge as a result of the sum of rainfall (m/day) and snowmelt (m/day). Discharge should be calculated as m3/day. You can get the area of a cell (here in m3 as the length of the cells on the map is entered in metres when creating the map) with the function `cellarea`. Save the `runoff.py` script and run.

Animate the discharge using Aguila. Try a logarithmic scale for better visualisation (right-click on the legend, select 'Edit draw properties', set 'Colour assignment' to logarithmic *and* set the 'minimum cutoff' to a positive (non zero) value).

---

**Question 20:** What is the maximum discharge in the study area?

a) 0.3·10^6 m3/day

b) 1.3·10^6 m3/day

c) 2.3·10^6 m3/day

d) 1.3·10^7 m3/day

Correct answer: d

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
        dem = self.readmap("dem")
        elevationMeteoStation = 2058.1
        elevationAboveMeteoStation = dem - elevationMeteoStation
        temperatureLapseRate = 0.005
        self.temperatureCorrection = elevationAboveMeteoStation * temperatureLapseRate
        self.report(self.temperatureCorrection, "tempCor")

        self.snow = 0.0

        self.ldd = pcr.lddcreate(dem, 1e31, 1e31, 1e31, 1e31)
        self.report(self.ldd, "ldd")

    def dynamic(self):
        precipitation = pcr.timeinputscalar("precip.tss", 1)
        self.report(precipitation, "pFromTss")
        temperatureObserved = pcr.timeinputscalar("temp.tss", 1)
        self.report(temperatureObserved, "tempObs")

        temperature = temperatureObserved - self.temperatureCorrection
        self.report(temperature, "temp")

        freezing = temperature < 0.0
        self.report(freezing, "freez")
        snowFall = pcr.ifthenelse(freezing, precipitation, 0.0)
        self.report(snowFall, "snF")
        rainFall = pcr.ifthenelse(pcr.pcrnot(freezing), precipitation, 0.0)
```

(continues on next page)

(continued from previous page)

```
        self.report(rainFall, "rF")

        self.snow = self.snow + snowFall

        potentialMelt = pcr.ifthenelse(pcr.pcrnot(freezing), temperature * 0.01, 0)
        self.report(potentialMelt, "pmelt")
        actualMelt = pcr.min(self.snow, potentialMelt)
        self.report(actualMelt, "amelt")

        self.snow = self.snow - actualMelt

        runoffGenerated = actualMelt + rainFall
        self.report(runoffGenerated, "rg")

        discharge = pcr.accuflux(self.ldd, runoffGenerated * pcr.cellarea())
        self.report(discharge, "q")

        self.report(self.snow, "snow")


nrOfTimeSteps = 181
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

## 2.6 Calculating descriptive statistics: the snow melt model

### 2.6.1 Over the spatial domain

For analysis or application of model outputs, it is often needed to calculate statistical values (e.g., mean, maximum value) over regions or time periods. We focus first on regions.

Display the map `skiregio.map`. It contains two skiing regions in the study area.

Open the script `runoff.py` and save as `ski.py`. For evaluation of the regions for possibilities for skiing, calculate the average snowpack (m water equivalent) in each of these regions, for each time step. Use the function `areaaverage`. Write the results to disk and display. Plot a time series (right-click on the legend) and click on one of the regions to get the time series for that particular region.

---

**Question 21:** Which of the two regions has in general the largest amount of snow?

a) Region 1.

b) Region 2.

Correct answer: b

Feedback: None

---

Although average snow pack is important, it is more important that the snowpack thickness is above a certain threshold required for skiing. Calculate for each time step:

- A map of the study area that is True at cells with a snowpack that is thicker than 0.2 m (water equivalent thickness), and
- A map with the proportion (a value between zero and one) of each area that has a snowpack that is thicker than 0.2 m.

For the second step, convert the map created in the first step to scalar data type using `scalar`. Now you can use it in an `areatotal` function. To get the fraction, you need the number of cells for each region, which can be calculated by:

```
pcr.areaarea("skiregio.map") / pcr.cellarea()
```

Store under the same filename `ski.py` and run.

---

**Question 22:** Compare the two regions by plotting timeseries. For approximately how many days is each region fully covered with a snow pack that is sufficiently thick for skiing ?

a) Based on the results we can conclude that there is never sufficient snow in both areas.

b) 5 days.

c) 10 days.

d) 40 days.

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("dem.map")

    def initial(self):
        dem = self.readmap("dem")
        elevationMeteoStation = 2058.1
        elevationAboveMeteoStation = dem - elevationMeteoStation
        temperatureLapseRate = 0.005
        self.temperatureCorrection = elevationAboveMeteoStation * temperatureLapseRate
        self.report(self.temperatureCorrection, "tempCor")

        self.snow = 0.0

        self.ldd = pcr.lddcreate(dem, 1e31, 1e31, 1e31, 1e31)
        self.report(self.ldd, "ldd")
        self.skiRegions = self.readmap("skiregio")
        self.numberCellsRegion = pcr.areaarea(self.skiRegions) / pcr.cellarea()

    def dynamic(self):
        precipitation = pcr.timeinputscalar("precip.tss", 1)
        self.report(precipitation, "pFromTss")
        temperatureObserved = pcr.timeinputscalar("temp.tss", 1)
```

(continues on next page)

(continued from previous page)

```
        self.report(temperatureObserved, "tempObs")

        temperature = temperatureObserved - self.temperatureCorrection
        self.report(temperature, "temp")

        freezing = temperature < 0.0
        self.report(freezing, "freez")
        snowFall = pcr.ifthenelse(freezing, precipitation, 0.0)
        self.report(snowFall, "snF")
        rainFall = pcr.ifthenelse(pcr.pcrnot(freezing), precipitation, 0.0)
        self.report(rainFall, "rF")

        self.snow = self.snow + snowFall

        potentialMelt = pcr.ifthenelse(pcr.pcrnot(freezing), temperature * 0.01, 0)
        self.report(potentialMelt, "pmelt")
        actualMelt = pcr.min(self.snow, potentialMelt)
        self.report(actualMelt, "amelt")

        self.snow = self.snow - actualMelt

        runoffGenerated = actualMelt + rainFall
        self.report(runoffGenerated, "rg")

        discharge = pcr.accuflux(self.ldd, runoffGenerated * pcr.cellarea())
        self.report(discharge, "q")

        self.report(self.snow, "snow")

        snowInSkiRegion = pcr.areaaverage(self.snow, self.skiRegions)
        self.report(snowInSkiRegion, "avsn")
        thickEnough = pcr.ifthenelse(self.snow > 0.2, pcr.boolean(1), pcr.boolean(0))
        self.report(thickEnough, "th")
        numberCellsThickEnough = pcr.areatotal(pcr.scalar(thickEnough), self.skiRegions)
        self.report(numberCellsThickEnough, "nth")
        proportionThickEnough = numberCellsThickEnough / self.numberCellsRegion
        self.report(proportionThickEnough, "ski")


nrOfTimeSteps = 181
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
```

### 2.6.2 Over time

It is also possible to calculate statistics over time.

Calculate a map with the maximum snowpack thickness (water equivalent) over time. Use the function `max`. Add the required statements to `ski.py`.

**Question 23:** What is the maximum snow thickness in the bottom right corner cell?

a) 0.30 m

b) 0.91 m

c) 0.27 m

d) 1.90 m

Correct answer: c

Feedback: In order to answer this question we need to add a new global variable to the initial: `self.maxSnowThickness = 0.0` At the end of the dynamic we add: `self.maxSnowThickness = pcr.max(self.snow, self.maxSnowThickness) self.report(self.maxSnowThickness, 'max')` If we visualize the last time step (aguila max00000.181) we can read the result: 0.27 m.

---

## 2.7 Direct neighbourhood operations

### 2.7.1 Discrete attributes: cellular automata

#### Initializing Game of Life

Game of Life is the best known cellular automata model. It models a population using Boolean valued grid cells, where True represents an alive cell and False a dead cell. Although it uses very simple deterministic transition rules it results in pretty unexpected results, as you will see.

The following transition rules are applied for each time step (generation). For each cell, the number of eight direct neighbourcells that is True (i.e., alive) is counted. This count is used to determine what happens with the cell in the time step:

1. Birth: if the current cell is False and the count is 3, the current cell becomes True.
2. Surivival: if (a) the count is 2, or (b) the count is 3 and the current cell is True, the current cell is left unchanged.
3. Death: if the count is less than 2 or greater than 3, the current cell becomes False.

Go to the folder `neighbourhood/life` and open `life.py`. It is an almost empty model that runs only one time step. In a later phase you can increase the number of time steps. First you need to create an initial map of the distribution of alive cells. We assume that initially 10% percent of the cells is alive, at randomly chosen locations. In the initial section, add:

```
aUniformMap = pcr.uniform(1)
```

Store the result to disk and run.

---

**Question 24:** What is done by the `uniform` function?

a) It draws a realization between 0 and 1, for each cell independently.

b) It draws a realization between 0 and 1, for all cells together.

c) It sorts values on a map, and numbers the cells according to the order.

Correct answer: a

Feedback: None

---

Now, use `aUniformMap` in a calculation to create the initial distribution of alive cells. You need the PCRaster function less than (<). Name the result of this calculation `self.alive` and write it to disk. Run and look at the results.

---

**Question 25:** What other (in addition to <) PCRaster operations did you use to create `self.alive`?

a) `ifthen`

b) `ifthenelse`

c) `boolean`

d) No other operation was required.

Correct answer: d

Feedback: Note that < already results in a Boolean map (cells True or False), so and if-then statement is not needed. `self.alive = aUniformMap < 0.1 self.report(self.alive,'ini')`

---

### Implementing the update rules

To represent the Birth, Survival, and Death update rules, we need to add statements to the dynamic. To count the number of neighbouring cells that is True, the boolean map `self.alive` needs to be converted to a map with a scalar data type. In the dynamic, type:

```
aliveScalar = pcr.scalar(self.alive)
```

The function `scalar` assigns a scalar 1.0 to cells that are True on its input and 0.0 otherwise.

Now, add a statement to the dynamic calculating for each cell the number of neighbouring cells that is True. It should only count in a neighbourhood consisting of 8 direct neighbours (excluding the cell itself). You need to use at least the `windowtotal` function. Assign the result to the variable `numberOfAliveNeighbours` and write the result to disk. Run and compare the result with the map containing the alive cells (`self.alive`).

As a next step, let's add the birth update rule. First create a Boolean map that is True for cells with 3 alive neighbours. Write to disk and check the results. Add another statement resulting in a map (`birth`) that returns True for cells with birth. You will need the functions `pcrand` and `pcrnot`.

---

**Question 26:** What statement did you use?

a) `birth = pcr.pcrnot(threeAliveNeighbours, pcr.pcrand(self.alive))`

b) `birth = pcr.pcrand(self.alive, pcr.pcrnot(self.alive))`

c) `birth = pcr.pcrand(self.alive, pcr.pcrnot(threeAliveNeighbours))`

d) `birth = pcr.pcrand(threeAliveNeighbours, pcr.pcrnot(self.alive))`

Correct answer: d

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        aUniformMap = pcr.uniform(1)
        self.report(aUniformMap, "uni")
        self.alive = aUniformMap < 0.1
        self.report(self.alive, "ini")

    def dynamic(self):
        aliveScalar = pcr.scalar(self.alive)
        numberOfAliveNeighbours = pcr.windowtotal(aliveScalar, 3) - aliveScalar
        self.report(numberOfAliveNeighbours, "na")

        threeAliveNeighbours = numberOfAliveNeighbours == 3
        self.report(threeAliveNeighbours, "tan")
        birth = pcr.pcrand(threeAliveNeighbours, pcr.pcrnot(self.alive))


nrOfTimeSteps = 100
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

Now, add Survival. Add a statement that calculates a map (`survivalA`) that is True according to Survival rule (a) (i.e., count is 2). Also, calculate `survivalB` containing True according to Survival rule (b). Combine `survivalA` and `survivalB` to create a map (`survival`) with cells that survive.

Death does not need to be represented by a separate calculation, because it is implicit when Birth and Survival is combined.

As a last step, combine the maps `birth` and `survival` that updates the `self.alive` map. Write the `self.alive` map to disk at the bottom of the dynamic, and run the model again. By comparing the `self.alive` map in the initial and after the first timestep, check whether your script really follows the three update rules of Game of Life.

Now, run the model for 100 timesteps by changing the number of timesteps at the bottom of the script. Animate the `self.alive` maps. Do you see similar patterns given at at http://mathworld.wolfram.com/GameofLife.html ?

---

**Question 27:** How did you combine the `survivalA` AND `survivalB` maps?

a) `survival = pcr.pcrand(survivalA, survivalB)`

b) `survival = pcr.pcror(survivalA, survivalB)`

c) `survival = pcr.pcrxor(survivalA, survivalB)`

d) `survival = survivalA + survivalB`

Correct answer: b

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        aUniformMap = pcr.uniform(1)
        self.report(aUniformMap, "uni")
        self.alive = aUniformMap < 0.1
        self.report(self.alive, "ini")

    def dynamic(self):
        aliveScalar = pcr.scalar(self.alive)
        numberOfAliveNeighbours = pcr.windowtotal(aliveScalar, 3) - aliveScalar
        self.report(numberOfAliveNeighbours, "na")

        threeAliveNeighbours = numberOfAliveNeighbours == 3
        self.report(threeAliveNeighbours, "tan")
        birth = pcr.pcrand(threeAliveNeighbours, pcr.pcrnot(self.alive))

        survivalA = pcr.pcrand((numberOfAliveNeighbours == 2), self.alive)
        survivalB = pcr.pcrand((numberOfAliveNeighbours == 3), self.alive)
        survival = pcr.pcror(survivalA, survivalB)

        self.alive = pcr.pcror(birth, survival)
        self.report(self.alive, "alive")


nrOfTimeSteps = 100
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

### 2.7.2 Continuous attributes: spatial diffusion in vegetation modelling

#### A logistic growth model

Logistic growth of vegetation can be modelled as:

$$\frac{dB}{dt} = rB(1 - \frac{B}{k}) - (c_0 + c_{inc}t)\frac{B^2}{B^2 + 1} + dD_{x,y} + e$$

With:

$B$, biomass

$r$, growth rate (0.08)
$k$, carrying capacity (10)
$c_0$, initial grazing pressure (0.1)
$c_{inc}$, increase in grazing pressure (0.00006)
$d$, dipsersion rate (0.01)
$D_{x,y}$, dispersion
$e$, random noise

In your exercises directory, go to the directory `neighbourhood/growth` and open `growth.py`. Program the logistic growth model assuming $d$, $D_{x,y}$, and $e$ are zero. The initial value of $B$ is 8.5. Use an explicit solution, which means that for each timestep, you calculate $dB/dt$ according to the equation above and simply add it to $B$. Use `self.x` as variable name for the biomass. Initialize it in the initial by typing:

```
self.x = pcr.spatial(pcr.scalar(8.5))
```

This is needed to tell PCRaster that `self.x` is a map (`spatial`) of data type scalar. Another hint: you can update the grazing pressure, i.e. the term $(c_0 + c_{inc}t)$ by typing in the dynamic:

```
self.c = self.c + self.cI
```

where `c` is the grazing pressure and `cI` is the increase in grazing pressure. In the initial you need to assign the initial value to grazing pressure.

Run the model for 2500 time steps and look at the result.

---

**Question 28:** What is the change in the biomass over time?

a) It remains almost constant.

b) It gradually increases.

c) It increases exponentially.

d) It gradually decreases and then drops quickly.

Correct answer: d

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class Growth(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        # maximum growth rate
        self.r = 0.08

        # grazing rate
```

(continues on next page)

(continued from previous page)

```
        self.c = 0.1

        # increase in grazing rate
        self.cI = pcr.scalar(0.00006)

        # carrying capacity
        self.K = pcr.scalar(10)

        # state variable
        self.x = pcr.spatial(pcr.scalar(8.5))

    def dynamic(self):
        growth = self.r * self.x * (1 - self.x / self.K) - self.c * (
            (self.x * self.x) / ((self.x * self.x) + 1)
        )
        self.x = self.x + growth

        self.report(self.x, "x")

        self.c = self.c + self.cI

nrOfTimeSteps = 2500
myModel = Growth()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

The observed change is known as a critical shift or critical transition in the literature. This means that the state of the ecosystem abruptly changes while the change in the driver is gradual. It is notably hard to predict such critical shifts, because the change in biomass is very small until the shift (here: abrupt decrease) occurs. Early warning signals have been described that change well ahead of the shift. One early warning signal is the spatial variance of the biomass. This can only be observed when some random noise is added to the biomass.

Add a statement to the dynamic that adds `randomnoise` $e$ to the biomass. You can create a map with random noise by typing:

```
randomNoise = pcr.normal(1) / 10.0
```

Run the model again, and animate the map with the random noise, and the biomass.

---

**Question 29:** What is the effect of the random noise on the time series of maps of biomass?

a) The time step for the critical shift differs spatially.

b) The program hangs after some time steps due to random inputs.

c) The resulting timing of the shift changes but is the same for all cells.

d) The time series gets shorter.

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class Growth(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        # maximum growth rate
        self.r = 0.08

        # grazing rate
        self.c = 0.1

        # increase in grazing rate
        self.cI = pcr.scalar(0.00006)

        # carrying capacity
        self.K = pcr.scalar(10)

        # state variable
        self.x = pcr.spatial(pcr.scalar(8.5))

    def dynamic(self):
        growth = self.r * self.x * (1 - self.x / self.K) - self.c * (
            (self.x * self.x) / ((self.x * self.x) + 1)
        )
        self.x = self.x + growth

        self.x = pcr.max(self.x + pcr.normal(1) / 10, 0)
        self.report(self.x, "x")

        self.c = self.c + self.cI

nrOfTimeSteps = 2500
myModel = Growth()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

Extend your script with statements that calculate the variance of the biomass map values. Remember that variance is:

$$var(B) = \frac{\sum_{i=1}^{N} (B_i - B_{mean})^2}{N}$$

With:

$B_i$, a biomass value at a grid cell,
$B_{mean}$, the mean biomass value of all grid cells,

$N$, the number of grid cells.

First, calculate $B_{mean}$, using `maptotal`, and note that the map is 40 by 40 cells. Use this map containing the mean value to calculate a map with the variance of biomass. The calculation needs to be done each time step, so add the statements to the dynamic. Store the output, run the model, display, and display the time series of variance.

---

**Question 30:** What is the change in the variance over time? Do you think it can be use to detect the upcoming downward shift in biomass?

a) It decreases immediately, so it cannot be used to forecast the shift, which occurs much later.

b) It increases immediately, so it cannot be used to forecast the shift, which occurs much later.

c) It decreases well ahead of the shift, so it can be used to forecast the shift.

d) It increases well ahead of the shift, so it can be used to forecast the shift.

Correct answer: d

Feedback: The variance increases drastically at an earlier time step than the drastic decrease in biomass. Thus, variance can be used as an early warning indicator. Note that the variance gets so large, that Aguila may give a segmentation fault if you try to display the full time series. To prevent this, plot only the first part of the time series: `aguila --timesteps=[1,1700,1] var`

```
import pcraster as pcr
import pcraster.framework as pcrfw


class Growth(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        # maximum growth rate
        self.r = 0.08

        # grazing rate
        self.c = 0.1

        # increase in grazing rate
        self.cI = pcr.scalar(0.00006)

        # carrying capacity
        self.K = pcr.scalar(10)

        # state variable
        self.x = pcr.spatial(pcr.scalar(8.5))

    def dynamic(self):
        growth = self.r * self.x * (1 - self.x / self.K) - self.c * (
            (self.x * self.x) / ((self.x * self.x) + 1)
        )
```

(continues on next page)

(continued from previous page)

```
        self.x = self.x + growth

        self.x = pcr.max(self.x + pcr.normal(1) / 10, 0)
        self.report(self.x, "x")

        self.c = self.c + self.cI

        mean = pcr.maptotal(self.x) / (40 * 40)
        var = pcr.maptotal(pcr.sqr(self.x - mean)) / (40 * 40)
        self.report(var, "var")


nrOfTimeSteps = 2500
myModel = Growth()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

### Diffusion of biomass

Thus far, we neglected spatial interactions in the model. To represent clonal growth and other dispersal mechanisms of biomass, the model presented in the previous section includes dispersion $D$. It is calculated as:

$$D_{x,y} = B_{x,y-1} + B_{x+1,y} + B_{x+1,y} + B_{x-1,y} - n_{x,y}B_{x,y}$$

With,

$D_{x,y}$, diffusion term at a cell located at (x,y),
$B_{x,y-1}$, biomass at a direct neighbour in the negative y direction,
$n_{x,y}$, the number of direct neighbouring cells.

For all cells except cells at the edge of the map, $n$ is equal to four neighbours, however at the edge cells one neighbour is missing, so we need to adjust $n$ here. In PCRaster Python, you can represent the equation by typing in the dynamic:

```
diffusion = self.d * (pcr.window4total(self.x) - self.numberOfNeighbours * self.x)
```

The `window4total` function sums the cell values of the four direct (top, right, bottom, left) neighbouring cells, `self.d` is the diffusion or dispersion rate (0.01). In the initial you need to calculate:

```
self.numberOfNeighbours = pcr.window4total(pcr.spatial(pcr.scalar(1)))
```

You also need to assign 0.01 to `self.d` in the initial. Add these statements to the model, such that diffusion $D_{x,y}$ is added to the biomass for each time step. Run the model and animate the biomass maps, using timeseries plots to evaluate the results.

---

**Question 31:** As you can see, the diffusion process results in patches of lower and higher biomass. What happens with the patch size over time (e.g. compare the pattern at the start of the model run, with the pattern just before the shift towards a lower biomass)?

a) The patch size increases over time.

b) The patches change, but the size stays the same.

c) The patch size decreases over time.

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class Growth(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        # maximum growth rate
        self.r = 0.08

        # grazing rate
        self.c = 0.1

        # increase in grazing rate
        self.cI = pcr.scalar(0.00006)

        # carrying capacity
        self.K = pcr.scalar(10)

        # state variable
        self.x = pcr.spatial(pcr.scalar(8.5))

        # dispersion rate
        self.D = pcr.scalar(0.01)

        self.numberOfNeighbours = pcr.window4total(pcr.spatial(pcr.scalar(1)))

    def dynamic(self):
        growth = self.r * self.x * (1 - self.x / self.K) - self.c * (
            (self.x * self.x) / ((self.x * self.x) + 1)
        )
        diffusion = self.D * (
            pcr.window4total(self.x) - self.numberOfNeighbours * self.x
        )
        growth = growth + diffusion
        self.x = self.x + growth

        self.x = pcr.max(self.x + pcr.normal(1) / 10, 0)
        self.report(self.x, "x")

        self.c = self.c + self.cI
```

(continues on next page)

(continued from previous page)

```
        mean = pcr.maptotal(self.x) / (40 * 40)
        var = pcr.maptotal(pcr.sqr(self.x - mean)) / (40 * 40)
        self.report(var, "var")


nrOfTimeSteps = 2500
myModel = Growth()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

## 2.8 Probabilistic spatial models: forest fire and seed dispersal

### 2.8.1 Random variables in point models

#### Continuous random variables

PCRaster comes with a number of functions to draw realizations from continuous probability distributions.

Go to the folder /probab/fire in your exercises data set and open `randomVar.py` To the dynamic, add a line that draws a realization using the function `normal` Store the result. In a similar way, use `uniform` and store the result. Animate the outputs.

---

**Question 32:** What is calculated by the function `uniform`?

a) It draws realizations resulting in a uniform average value of 1.0.

b) It draws realizations from a uniform distribution between zero and one, independently for each cell and time step.

c) It draws a realization from a uniform distribution between zero and one, assigning it to all cells, independently for each time step.

Correct answer: b

Feedback: Look up the description of the function in the PCRaster manual pages if you do not understand the function.

---

Now, test the function `mapuniform` by adding it to the dynamic section and storing results. Look it up in the PCRaster manual pages if you need to find what inputs it requires.

---

**Question 33:** In what situations would you use `uniform` and in what situations would you use `mapuniform`?

a) The function `uniform` is used when error at a cell is independent from other cells. The function `mapuniform` represents uncertainty of an error source that is fully correlated over the area, i.e. over the whole area the error causes the value to be too high or too low; with `uniform`, the value would be too high or too low for each cell independently.

b) The function `mapuniform` is used when error at a cell is independent from other cells. The function `uniform` represents uncertainty of an error source that is fully correlated over the area, i.e. over the whole area the error causes the value to be too high or too low; with `mapuniform`, the value would be too high or too low for each cell independently.

c) The function `uniform` is used when the error is larger than indicated by the measurement instrument, as it adds uncertainty caused by spatial variation. The function `mapuniform` is used when the measurement instrument exactly represents the error over the whole area.

Correct answer: a

Feedback: None

---

Now, add a statement that draws realizations from a Gaussian distributed variable with mean a mean of 10 and standard deviation of 5.

---

**Question 34:** What is the range of the values in the realizations that you find if you run the model for all time steps?

a) The values are approximately between 5.0 and 10.0

b) The values are approximately between 0.0 and 20.0

c) The values are approximately between 5.0 and 15.0

d) The values are approximately between -15.0 and 35.0

Correct answer: d

Feedback: The result should be realizations for each cell from a Gaussian distribution with a mean 10 and a standard deviation of 5. In theory, any value could be drawn (as the Gaussian distribution is not bounded), but with this number of cells and time steps, one find mostly values in the range between -15 and 35. If you did not find this, you most likely made an error in the script. In the dynamic, the following two lines are needed:

```
        # draw from a normal distribution
        # with mean zero and standard deviation one
        nor = pcr.normal(1)
        # adjust the realizations:
        # to get a standard deviation of five (instead of one),
        # we first multiply by five, which will change the sd,
        # then we add 10.0, to move the mean to ten.
        rv = 10.0 + nor * 5.0
        self.report(rv, "rv")
```

---

### Discrete random variables

Functions to draw realizations of discrete (classified) random variables are not provided. These can be derived from the functions for continuous random variables that you used in the previous section.

Open `randomVar.py` and add (a) statement(s) to the script that draws realizations (for each cell independently and for each timestep) of a discrete random variable with two possible outcomes, True and False:

$P(\text{True}) = 0.8$
$P(\text{False}) = 0.2$

Write results to disk, run the script, and display to check the result.

---

**Question 35:** How could you create the discrete random variable?

a) By drawing realizations from a uniform distribution between 0.2 and 0.8, converting the result to a Boolean.

b) By drawing realizations from a uniform distribution between 0.2 and 0.8, selecting the values above 0.5.

c) By drawing realizations from a normal distribution with a mean of 0.5 and a standard deviation of 1.0, selecting values above 0.2.

d) By drawing realizations from a uniform distribution between 0.0 and 1.0, selecting values below 0.8.

Correct answer: d

Feedback: None

```
        uni = pcr.uniform(1)
        aOne = uni < 0.8
        self.report(aOne, "aone")
```

---

Now, add statements that draw realizations of a discrete random variable with three possible outcomes, coded as 1, 2, and 3:

$P(1) = 0.01$
$P(2) = 0.9$
$P(3) = 0.09$

---

**Question 36:** How could you create the discrete random variable?

a) Using realizations from a uniform distribution, a table, and lookupnominal.

b) By multiplication of the outcome of the function `uniform`, three times.

c) Using realizations from a uniform distribution, and the `maptotal` function, to select values.

d) Using a window function.

Correct answer: a

Feedback: Just like in the previous exercise, we will derive the realizations from realizations of a uniform distribution between zero and one. First you need to create an ASCII table to assign the correct values, save it as `threeCl.tbl`, for instance: `[0,0.01> 1 [0.01,0.91> 2 [0.91,1.0] 3`

```
        uni = pcr.uniform(1)
        classes = pcr.lookupnominal("threeCl.tbl", uni)
        self.report(classes, "classes")
```

---

### 2.8.2 Direct neighbourhood interaction: forest fire

#### Fire spreading

Direct neighbourhood functions (as used in cellular automata) are suitable for modelling forest fire. This is because the large scale spatio-temporal pattern of an evolving fire is driven by local interactions at the fire front. These local interactions will be represented here using a stochastic direct neighbourhood model. The model uses the following transition rules that are applied for each time step:

1. A cell that was burning at the previous time step stays burning.
2. Cells that are not burning at the previous time step potentially catch fire only when one or more neighbouring cells are burning at the previous time step. Neighbouring cells are the four direct neighbours in the x, y direction.
3. The cells under item 2 actually catch fire with a probability of 0.1.
4. A cell that was not burning at the previous time step and actually catched fire in the current timestep becomes a burning cell.

Display `dem.map` and `start.map`. The `start.map` contains True cells representing the starting area of a fire. Open `fire.py`. First add statements to represent rule 2.

- Set the initial distribution of the fire (use the variable name `fire`) to `start.map`.
- In the dynamic, add a `window4total` statement that calculates the number of neighbours that are burning.
- In the dynamic, use the result of `window4total` to calculate a map `neighbourBurns` that is True when one or more neighbouring cells are burning.
- Write the resulting maps to disk, save the script, and run.

Animate `neighbourBurns`. Do the cells with True also include cells that were already burning in the previous time step?

And finally for rule 2, add (a) statement(s) that calculate(s) `potentialNewFire`, a map that is True *only* for cells that potentially catch fire. To calculate this map, you need to combine `fire` and `neighbourBurns` using Boolean logic operations. Write to disk and check the result.

---

**Question 37:** How did you calculate `potentialNewFire`?

a) Using Boolean logic operations.

b) Using multiplication.

c) Using an additional `window4total` statement.

d) Using an additional `window4total` statement and `areaaverage`.

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class Fire(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")
```

(continues on next page)

(continued from previous page)

```
    def initial(self):
        self.fire = self.readmap("start")

    def dynamic(self):
        # nr of neighbours with fire
        nrOfBurningNeighbours = pcr.window4total(pcr.scalar(self.fire))
        # cells where at least one neighbour is burning
        neighbourBurns = nrOfBurningNeighbours > 0
        self.report(neighbourBurns, "nb")

        # cells that are not yet burning and where one neighbour burns
        potentialNewFire = neighbourBurns & ~self.fire
        # if you want to avoid the use of operators the previous line
        # can also be written as follows:
        # potentialNewFire = pcr.pcrand(neighbourBurns, pcr.pcrnot(self.fire))

        self.report(potentialNewFire, "pnf")


nrOfTimeSteps = 200
myModel = Fire()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

Now, let's add rule 3.

- Add the following statement to the dynamic calculating for each time step a map that is True with a probability of 0.1 and false elsewhere. Write the result to disk to check it.

```
realization = pcr.uniform(1) < 0.1
```

- Add a statement that calculates `newFire`, a map that is True for cells that actually catch fire in the current time step. `newFire` represents rule 3. Write to disk, run, and check the result.

And finally, update `fire` by combining `newFire` and `fire` from the previous time step. Write to disk, store your model script, run it, and check the results.

---

**Question 38:** How does the fire evolve over time?

a) Due to the window shape of the window4total function it is covers an approximately rectangular block which increases in size over time, having a ragged edge due to the random process.

b) It covers a circle which increases in size, ragged at the edge due to the random process.

c) It covers a circle which increases in size, with a smooth edge.

d) Due to the window shape of the window4total function it is covers an approximately rectangular block which increases in size over time, having a smooth edge.

Correct answer: b

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class Fire(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        self.fire = self.readmap("start")

    def dynamic(self):
        # nr of neighbours with fire
        nrOfBurningNeighbours = pcr.window4total(pcr.scalar(self.fire))
        # cells where at least one neighbour is burning
        neighbourBurns = nrOfBurningNeighbours > 0
        self.report(neighbourBurns, "nb")

        # cells that are not yet burning and where one neighbour burns
        potentialNewFire = neighbourBurns & ~self.fire
        # if you want to avoid the use of operators the previous line
        # can also be written as follows:
        # potentialNewFire = pcr.pcrand(neighbourBurns, pcr.pcrnot(self.fire))

        self.report(potentialNewFire, "pnf")

        realization = pcr.uniform(1) < 0.1
        self.report(realization, "real")

        newFire = potentialNewFire & realization
        # or alternative notation:
        # newFire = pcr.pcrand(potentialNewFire, realization)
        self.report(newFire, "nf")

        self.fire = self.fire | newFire
        # or:
        # self.fire = pcr.pcror(self.fire, newFire)
        self.report(self.fire, "fire")


nrOfTimeSteps = 200
myModel = Fire()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

### Fire spreading determined by the topography

In the previous exercise, we assumed a probability of a cell catching fire to be everywhere the same, i.e. 0.1. In reality, however, a fire front mainly extends uphill. A very simple representation of this process is used here. The transition rules are kept the same, except for the probability used in rule 2. Now we will calculate this probability as:

- All cells with a direct neighbour in downhill direction have a probability to catch fire of 0.8.
- For all other cells, this probability is 0.1.

Open the script `fire.py` that was created in the previous exercise and save it as `fireUphill.py`. In the initial, create the map `ldd` using the digital elevation model `dem.map`. You need `self.ldd` as variable name as it will be used in the dynamic. Display the local drain direction map on top of the digital elevation model, zoom in if you get an almost black map!

In the dynamic, you need to add statements calculating for each map the probability for cells to catch fire. As a start, add two statements (insert them directly above the calculation of `realization`:

- A statement calculating `downhillBurning`, a map that is True for cells with a cell in downhill direction that is burning. Use the function `downstream`.
- A statement calculating `probability`, a map with the probabilities that a cell catches fire. Use `downhillBurning` as input.

Let the model write the results of these two calculations to disk. Save and run the model to see the results.

Now, you need to change the calculation of `realization` as it needs to take into account the spatial pattern (for each time step) of the `probability`. Do this by changing the statement. Save and run the model. Animate the resulting fire pattern together with the local drain direction map and the digital elevation model.

**Question**
What is the pattern of the fire spreading over time? Do you think this is more realistic compared to the previous model? Give two suggestions to further improve the model.

---

**Question 39:** Have a look at the veg.map. At what time step does the fire reach the pine forest in the southern part of the area (approximately)?

a) At time step 50.

b) At time step 100.

c) At time step 150.

d) It does not reach the pine forest.

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class Fire(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        self.fire = self.readmap("start")
        #
        dem = self.readmap("dem")
        self.ldd = pcr.lddcreate(dem, 1e31, 1e31, 1e31, 1e31)
```

(continues on next page)

(continued from previous page)

```
    def dynamic(self):
        nrOfBurningNeighbours = pcr.window4total(pcr.scalar(self.fire))
        neighbourBurns = nrOfBurningNeighbours > 0
        self.report(neighbourBurns, "nb")
        potentialNewFire = neighbourBurns & ~self.fire
        self.report(potentialNewFire, "pnf")

        #
        downhillBurning = pcr.downstream(self.ldd, self.fire)
        self.report(downhillBurning, "db")

        probability = pcr.ifthenelse(downhillBurning, pcr.scalar(0.8), 0.1)
        self.report(probability, "prob")
        #

        # realization=uniform(1) < 0.1
        realization = pcr.uniform(1) < probability
        self.report(realization, "real")

        newFire = potentialNewFire & realization
        self.report(newFire, "nf")

        self.fire = self.fire | newFire
        self.report(self.fire, "fire")


nrOfTimeSteps = 200
myModel = Fire()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

### 2.8.3 Neighbourhood interaction over a distance: plant seed dispersal

Plant dispersal occurs either as clonal growth or through seed dispersal. Clonal growth can be modelled using direct neighbourhood operations as new plants only occur directly neighbouring existing cells. Thus, it is actually a quite similar process, from the modelling point of view, as forest fire. Seed dispersal, however, allows plants to disperse over longer distances: new plants may establish also at cells that are not neighbours of cells containing plants in the previous time step. Here we will construct a simple seed dispersal model.

We use the following set of rules:

1. A single plant species disperses over an area.
2. For each time step, the probability $P$(E) that new plants establish at cells that are not yet occupied by the species is modelled as:

$P(E) = 0.1^{(d/r)}$

with, d, the distance away (m) from the nearest cell containing the plant, and r, a parameter (m, r = 40 m).

3. For each time step, the new distribution of the plants consists of the distribution of the plants in the previous time step combined with the distribution of the plants that established in the current time step.

Go to the directory `probab/plants` and display `veg.map`. We assume that the plant species to be modelled is artificially introduced (before dispersal starts) at all pixels containing dense herbs (class with code 5). Open `plants.py`. In

the initial, create a Boolean map (use the variable name `self.plants`) that is True at dense herbs and false elsewhere. This is the map containing the plants that will disperse over time.

To represent the transition rules, add statements to the dynamic creating a map `probability` with $P$(E) for each time step. For now, you can assume that the plant does not yet spread (i.e., `self.plants` does not change), so `probability` will not change over time. Store the script, and run. Display the results.

---

**Question 40:** What is the probability in the left corner of the study area?

a) Below 0.01.

b) Between 0.01 and 0.8.

c) Above 0.8.

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class Plants(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        vegetation = self.readmap("veg")
        self.plants = vegetation == 5
        self.range = 40

    def dynamic(self):
        distance = pcr.spread(self.plants, 0, 1)
        self.report(distance, "dist")

        probability = 0.1 ** (distance / self.range)
        self.report(probability, "prob")


nrOfTimeSteps = 50
myModel = Plants()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

Using `probability`, create a new map `newPlants` that is True at cells where the plant establishes and false elsewhere. You need to use `probability` as input, and the `uniform` function. Write the result to disk and check the outcome.

Now, update `self.plants` by 'adding' the newly established plants to `self.plants`. Write `self.plants` to disk in the dynamic. Save the script, run, and animate the plants dispersal, together with the probability maps. Run the model two times and compare results.

---

**Question 41:** How long does it approximately take to reach the the bottom left side of the area?

a) 5 time steps.

b) 10 time steps.

c) 25 time steps.

d) 40 time steps.

Correct answer: c

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class Plants(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        vegetation = self.readmap("veg")
        self.plants = vegetation == 5
        self.range = 40

    def dynamic(self):
        distance = pcr.spread(self.plants, 0, 1)
        self.report(distance, "dist")

        probability = 0.1 ** (distance / self.range)
        self.report(probability, "prob")

        newPlants = probability > pcr.uniform(1)
        self.report(newPlants, "np")

        self.plants = pcr.pcrand(self.plants, self.biotope)

        self.report(self.plants, "plants")


nrOfTimeSteps = 50
myModel = Plants()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

---

We can make all kinds of interesting extensions to the model. Let's see what happens if we assume the plant can only grow at vegetation units Open Shrub (code 3) and Dense herbs (code 5). In the initial, create a map `biotope` that is True at cells containing Open shrub or Dense herbs. In the dynamic, modify the script such that the plant will only exist or establish at these open shrub or dens herbs cells. Save the script, run.

---

**Question 42:** How long does it approximately take to reach the Open shrub patch on the left side of the area? Explain.

a) It will not be reached for sure as the probabilities are too low.

b) Almost certainly within 10 years.

c) It might be reached although probabilities are extremely low.

d) Almost certainly within 10 to 50 years.

Correct answer: c

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class Plants(pcrfw.DynamicModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

    def initial(self):
        vegetation = self.readmap("veg")
        self.plants = vegetation == 5
        self.range = 40

        self.biotope = pcr.pcror(vegetation == 5, vegetation == 3)
        self.report(self.biotope, "bio")

    def dynamic(self):
        distance = pcr.spread(self.plants, 0, 1)
        self.report(distance, "dist")

        probability = 0.1 ** (distance / self.range)
        self.report(probability, "prob")

        newPlants = probability > pcr.uniform(1)
        self.report(newPlants, "np")

        self.plants = pcr.pcror(self.plants, newPlants)
        self.plants = pcr.pcrand(self.plants, self.biotope)

        self.report(self.plants, "plants")


nrOfTimeSteps = 50
myModel = Plants()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
dynamicModel.run()
```

## 2.9 Beyond the Model class

Thus far we only used predefined Model class for control flow in the script. Of course you can use all Python constructs inside and outside this Model class. You may need this either to better structure your model, or to calculate things that cannot be done with using the time iteration provided by the Model class.

For instance, it is often wise to rewrite blocks of code as functions. Go to the subdirectory morePython and open `functions.py`. It is the fire model you programmed earlier. At the top of the dynamic are a number of lines finally calculating `potentialNewFire`. To make this script better readable, rewrite the program such that this upper block is encapsulated in a function `calculatePotentialNewFire`. Define this function at the top of the program (just below `import pcraster.framework as pcrfw` and call it in the dynamic. Be sure to test the program!

---

**Question:** Give another example of how Python constructs can be used in modelling.

- Functions
- Conditional execution

---

CHAPTER

THREE

# STOCHASTIC MODELLING

Download this website as pdf.

Download this website as epub (for e-readers).

To subscribe to our courses visit http://www.pcraster.eu

## 3.1 Visualisation of stochastic data

### 3.1.1 Realizations

PCRaster includes the Aguila software for visualisation of spatio-temporal stochastic data. With Aguila, you can easily explore large multi-dimensional data sets. For the exercises below, you will need to look up some options in the Aguila manual, available at https://pcraster.geo.uu.nl/pcraster/latest/documentation/pcraster_aguila/index.html.

To learn how to visualize stochastic data you will use hydrological model outputs from a model of a catchment in Spain.

Aguila is started from a command shell. Open a command shell and go to the directory containing your data for the exercises. Type cd to change directories. In the directory, you will find the folder `visualisation` containing the data for the visualisation practical. In this directory, display the digital elevation model, the vegetation map, and the local drain direction map:

```
aguila dem.map veg.map ldd.map <Enter>
```

Use the menu items (or right click on the legend to get more options) to zoom in/out, to change the color palette, and to change the drawing mode to contours. Also try changing the number of classes shown (or number of contours).

To represent saturated conductivity as a stochastic variable, realizations were drawn from probability distributions. These were used in the stochastic hydrological model. In PCRaster Python, realizations are stored in subdirectories numbered 1, 2,..,N, with N the number of realizations. Use the command `dir` (or `ls` on Linux) to check the number of realizations available in the data set.

Also, go to the folder `1` and type `dir` to have a look at its contents. You will see that it contains a `ksat.map` which is a realization of saturated conductivity (m/h).

Go back to the main folder `visualisation`.

You can display realizations of static data (without time component) by typing:

```
aguila --scenarios='{1,2,3,4,5,6,7,8,9,10,11,12}' ksat
```

Windows users can omit the quotes:

```
aguila --scenarios={1,2,3,4,5,6,7,8,9,10,11,12} ksat
```

This opens the realizations in separate windows. It is more convenient to display in one window, by typing:

```
aguila --scenarios='{1,2,3,4,5,6,7,8,9,10,11,12}' --multi=3x4 ksat
```

You can combine the visualisation with the vegetation map:

```
aguila veg.map --scenarios='{1,2,3,4,5,6,7,8,9,10,11,12}' --multi=3x4 ksat
```

---

**Question 1:** What is the value of saturated conductivity at the 25th, 50th and 75th percentile, respectively, in the unit dense herbs? Make a rough estimate. Use the realizations at a single cell location.

a) 25th percentile: 0.064; 50th percentile: 0.061; 75th percentile: 0.059.

b) 25th percentile: 0.059; 50th percentile: 0.061; 75th percentile: 0.064.

c) 25th percentile: 0.059; 50th percentile: 0.059; 75th percentile: 0.059.

d) 25th percentile: 0.061; 50th percentile: 0.059; 75th percentile: 0.064.

Correct answer: b

Feedback: None

---

Now, lets have a look at temporal data. Display the timeseries of precipitation:

```
aguila precip.tss
```

It contains precipitation (mm/h) for 10 hours. It was used as input to a stochastic model calculating actual infiltration for each time step, using the realizations of infiltration capacity.

The folders 1 to 12 contain realizations of actual infiltration (mm/h), numbered from 1 to 10 (`i0000000.001`, `i0000000.002`, etc), i.e. for each time step of an hour one map. Check this.

To animate these realizations, type in the `visualisation` folder:

```
aguila veg.map --scenarios='{1,2,3,4,5,6,7,8,9,10,11,12}' --multi=3x4  --timesteps=[1,10,
↪1] i
```

Plot a timeseries by right clicking on the legend of the infiltration maps `i`.

---

**Question 2:** By clicking on cells in the 'Dense herbs' area, have a look at the timeseries for this region. When is the uncertainty largest, with large amounts of rain or with low amounts of rain, or does rain not affect the uncertainty? Can you explain this?

a) The uncertainty is largest for low amounts of rainfall. This is because for these cases the variation in rainfall is largest.

b) The uncertainty is largest for high amounts of rainfall. This is because for these cases the variation in rainfall is largest.

c) The uncertainty does not vary with rainfall amount.

d) The uncertainty is largest during peak rainfall. With large amounts of rain, the differences among realizations of infiltration become clear.

Correct answer: d

Feedback: None

---

It is also possible to plot maps over each other, e.g. type:

```
aguila --scenarios='{1,2,3,4,5,6,7,8,9,10,11,12}' --multi=3x4  --timesteps=[1,10,1] q +␣
↪ldd.map
```

The maps `q` contain discharge (m3/h). Zoom in to see the content of the map. Click on the main channel and plot a timeseries. Animate.

### 3.1.2 Probability distributions and exceedance probabilities

Probability distributions are stored as a set of percentile maps. Unlike realizations, these are stored in the main folder `visualisation`. Type:

```
dir ksat_*map
```

Or in linux you would use `ls`. Note that * is a wildcard, representing all possible characters. These are maps containing the percentile values. The percentiles are given as fractional values between zero and 1 in the file name. You can display these maps just like other maps, e.g.:

```
aguila ksat_0.5.map
```

However, it is also possible to display them at once, resulting in a plot of the probability density distribution:

```
aguila --quantiles=[0.025,0.975,0.005] ksat veg.map
```

This opens all ksat percentile maps (and the vegetation map), and displays the probability distribution between the 2.5% and 97.5% percentiles, with a step of 0.5%. Note that, as the information is only available at a limited number of percentile values, aguila interpolates. You can open the probability density view by right clicking on the legend. Read through the Aguila manual at https://pcraster.geo.uu.nl/pcraster/latest/documentation/pcraster_aguila/Views.html#probability-graph-view for an explanation.

---

**Question 3:** Give the boundaries of the 95% confidence interval (i.e., between 2.5 and 97.5% percentiles) of saturated conductivity in the dense herbs area.

a) 2.5%: 0.0345 m/h; 97.5%: 0.0694 m/h

b) 2.5%: 0.0485 m/h; 97.5%: 0.0485 m/h

c) 2.5%: 0.0485 m/h; 97.5%: 0.0694 m/h

d) 2.5%: 0.0694 m/h; 97.5%: 0.0485 m/h

Correct answer: c

Feedback: None

---

You can retrieve cumulative probabilities by clicking on the + in the menu at the top of the probability density distribution view. After this, you can slide the vertical line in the probability density view, to get cumulative probabilities for different threshold values. If exceedance intervals are needed, right-click on the legend, select Edit draw properties, and select 'Exceedance probabilities'.

---

**Question 4:** Give the probability that saturated conductivity exceeds 0.056 in the dense herbs area.

a) The probability is 0.71

b) The probability is 0.29

c) The probability is 0.0071

d) The probability is 0.0029

Correct answer: a

Feedback: None

---

Probability density distributions can also be displayed for dynamic data.

```
dir i_*
```

These are the percentile maps of modelled actual infiltration, for each time step of an hour. Filenames refer to time steps and the percentile. To display and animate these, type:

```
aguila --quantiles=[0.025,0.975,0.005] --timesteps=[1,10,1] i veg.map
```

By right-clicking on the legend of `i`, open timeseries and the probability distribution plot. Animate. Click on different locations of the map to evaluate the uncertainty in infiltration.

In a similar way, create an animation of the probability distribution of the discharge, `q`.

---

**Question 5:** Change the map view such that it shows the cumulative probability for a threshold value of 20000 m3/h. Note that the probability that `q` exceeds the threshold is one minus the cumulative probability. Animate the cumulative probability maps. Describe the pattern in the exceedance probabilities (in space and time).

a) The higher the discharge, the lower the exceedance probability.

b) The higher the discharge, the higher the exceedance probability.

Correct answer: b

Feedback: None

---

### 3.1.3 Confidence intervals

To plot confidence intervals, type:

```
aguila --quantiles=[0.025,0.975,0.005] ksat veg.map
```

Open the probability density plot. Click on the '+' in the probability density plot window to switch to cumulative probabilities. Now, right click on the legend, select Edit draw properties, and in the colour assignment panel, select 'Confidence interval'. If you want you can change the alpha level of the confidence interval in the field below. Click 'OK'. Now, move the threshold value in the probability density plot (the vertical line), and Aguila interactively shows the confidence interval for that threshold value.

---

**Question 6:** Which vegetation units are certainly (i.e. with an alpha value of 0.05) above a threshold value of 0.01?

a) All units.

b) Pine forest, dense herbs, and open shrubs.

c) Open shrubs.

d) All units except open shrubs.

Correct answer: b

Feedback: None

---

## 3.2 The stochastic modelling framework

### 3.2.1 The static stochastic modelling class

Go to the folder `framework` in your folder containing the exercise data. For static stochastic modelling, you can use the template script `stochStaticMod.py`. Open the script, and run it. Type `dir` to see the contents of the folder.

---

**Question 7:** The model class contains a number of methods. In which section (method) do we need to put calculations representing processes (or importing data) that are stochastic, without time? Idem, purely deterministic calculations without uncertainty?

a) Stochastic without time: premcloop. Deterministic: initial.

b) Stochastic without time: postmcloop. Deterministic: initial.

c) Stochastic without time: initial. Deterministic: initial.

d) Stochastic without time: initial. Deterministic: premcloop.

Correct answer: d

Feedback: Premcloop: deterministic calculations done at the start. Initial: is run for each Monte Carlo realization, in this case for 10 MC runs. Postmcloop: calculates sample statistics at the end.

---

### 3.2.2 The dynamic stochastic modelling class

In the same `framework` folder, open and run `stochDynamicMod.py`. It is a template script for dynamic stochastic modelling.

---

**Question 8:** What is the dynamic method doing in the `stochDynamicMod.py` script?

a) It is run for each Monte Carlo realization, and each time step.

b) It is run for the first Monte Carlo realization, for each time step.

c) It is run for each Monte Carlo realization, without time steps.

d) It passes information from the initial to the postmcloop.

Correct answer: a

Feedback: None

---

## 3.3 Writing to disk, drawing realizations

### 3.3.1 Writing realizations to disk

Just like in deterministic modelling, `report` can be used to write data to disk.

In the `framework` folder, Open `stochDynamicMod.py` and save it as `real.py`. Add the following lines to the premcloop:

```
a = pcr.normal(1)
self.report(a, "a")
```

Add the same two statements to the initial and dynamic, using variable names `b` and `c`, respectively (instead of `a`). Save the script and run. Use `dir` to check the contents of the main folder `framework` and the folders containing the realizations, e.g. `1`. Use aguila to display `a`, `b`, and `c`.

---

**Question 9:** What is stored by reporting in the dynamic?

a) It stores files in subdirectories referring to the Monte Carlo realizations, where each file has a number referring to the time step. Each file is a time series.

b) It stores files in subdirectories referring to the time steps, where each file has a number referring to the Monte Carlo realization. Each file is a time series.

c) It stores files in subdirectories referring to the Monte Carlo realizations, where each file has a number referring to the time step. Each file is a map.

d) It stores files in subdirectories referring to the time steps, where each file has a number referring to the Monte Carlo realization. Each file is a map.

Correct answer: c

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("clone.map")

    def premcloop(self):
        a = pcr.normal(1)
        self.report(a, "a")

    def initial(self):
        b = pcr.normal(1)
        self.report(b, "b")

    def dynamic(self):
        c = pcr.normal(1)
        self.report(c, "c")
```

(continues on next page)

(continued from previous page)

```
    def postmcloop(self):
        print("running postmcloop")


nrOfSamples = 10
nrOfTimeSteps = 3
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
mcModel = MonteCarloFramework(dynamicModel, nrOfSamples)
```

---

Opening files from disk can be done with `self.readmap`, just like in dynamic modelling.

### 3.3.2 Drawing realizations

The function `normal` draws realizations from a stochastic variable. PCRaster includes a number of other functions that draw realizations from stochastic variables, e.g. `mapnormal`, `uniform`. These were explained in the dynamic modelling exercises. Here, you will learn to use `areanormal`.

Open the map `regions.map`. Boolean True cells are ski regions. For the ski regions, vegetation height is a stochastic variable with a mean 0.2 and a standard deviation of 0.05 (m). For the remaining area, the mean is 0.3 with a standard deviation 0.1.

Open `real.py` and add statements (in the initial) to create realizations of vegetation height using `areanormal`. Use the variable name `vegetationHeight` and write to disk as `h`. Save the script and run. Use aguila to display 10 realizations of `v`. Hints:

- Read `regions.map` from disk in the premcloop. It is a deterministic map thus needs to be opened there.
- Use `ifthenelse` to create factors for mean and standard deviation.

---

**Question 10:** What is done by the `areanormal` function?

a) It draws a random value for each cell independently.

b) It draws a random value for each area independently, these are different between Monte Carlo realizations.

c) It normalizes input values within the boundaries of a normal distribution.

d) It draws a random value for each area independently, these are the same over the Monte Carlo realizations.

Correct answer: b

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("clone.map")
```

(continues on next page)

(continued from previous page)

```
    def premcloop(self):
        a = pcr.normal(1)
        self.report(a, "a")
        self.regions = self.readmap("regions")

    def initial(self):
        b = pcr.normal(1)
        self.report(b, "b")

        normalAreas = pcr.areanormal(self.regions)
        sdFactor = pcr.ifthenelse(self.regions, 0.05, pcr.scalar(0.1))
        meanFactor = pcr.ifthenelse(self.regions, 0.2, pcr.scalar(0.3))
        vegetationHeight = normalAreas * sdFactor + meanFactor
        self.report(vegetationHeight, "h")

    def dynamic(self):
        c = pcr.normal(1)
        self.report(c, "c")

    def postmcloop(self):
        print("running postmcloop")


nrOfSamples = 10
nrOfTimeSteps = 3
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
mcModel = MonteCarloFramework(dynamicModel, nrOfSamples)
mcModel.run()
```

## 3.4 Functions calculating statistics over realizations

### 3.4.1 Mean and variance, static data

In the folder `framework`, open `real.py`. It is the script created in the previous exercise drawing realizations from stochastic variables. PCRaster includes functions to calculate statistics over these realizations. This is done in the postmcloop. These functions read the data written to disk in the initial or the dynamic, and calculate the statistics over these data, writing the results to disk.

The function `mcaveragevariance` calculates the mean and variance of realizations, on a cell-by-cell basis.

Save `real.py` under the new name `statistics.py`. Add the following lines to the postmcloop:

```
names = ['b']
sampleNumbers = self.sampleNumbers()
print(sampleNumbers)
timeSteps = [0]
pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)
```

The `mcaveragevariance` function requires three inputs:

- `names` is a list of filenames (given as strings) of the map variables for which the statistics are needed. As these are read from disk, you need to have stored these in the initial or the dynamic using report. Here we use a list of length 1, containing one variable.
- `sampleNumbers` is a list of Monte Carlo samples over which statistics are calculated. It is also printed to check it.
- `timesteps` is a list of time steps for which the statistics are created. As `b` represents a static variable, we provide a list containing one zero element. This tells `mcaveragevariance` that it gets static data.

After adding the lines, change the number of Monte Carlo samples to use to 100, for more precision. Save the script and run. Type `dir` in the main folder `framework`. The `mcaveragevariance` function has created the maps `b-ave.map` and `b-var.map`, containing the mean and variance. Display these maps.

Now, change `names` to calculate the statistics of vegetation height also. Save and run the script. Type `dir` in the main folder `framework` to see what it has stored. Display the mean and variance maps of vegetation height.

---

**Question 11:** How well does your Monte Carlo script calculate the original statistics of the stochastic variables?

a) For the variable `b`, the variance is approximately between 50% and 150% of the expected value.

b) Idem, between 10% and 190%.

c) Idem, between 1% and 1000%.

d) Idem, between 99% and 101%.

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("clone.map")

    def premcloop(self):
        a = pcr.normal(1)
        self.report(a, "a")
        self.regions = self.readmap("regions")

    def initial(self):
        b = pcr.normal(1)
        self.report(b, "b")

        normalAreas = pcr.areanormal(self.regions)
        sdFactor = pcr.ifthenelse(self.regions, 0.05, pcr.scalar(0.1))
        meanFactor = pcr.ifthenelse(self.regions, 0.2, pcr.scalar(0.3))
        vegetationHeight = normalAreas * sdFactor + meanFactor
        self.report(vegetationHeight, "h")
```

(continues on next page)

(continued from previous page)

```
        self.d = pcr.scalar(0)

    def dynamic(self):
        c = pcr.normal(1)
        self.report(c, "c")

    def postmcloop(self):
        names = ["b", "h"]
        sampleNumbers = self.sampleNumbers()
        print(sampleNumbers)
        timeSteps = [0]
        pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)

nrOfSamples = 100
nrOfTimeSteps = 10
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
mcModel = pcrfw.MonteCarloFramework(dynamicModel, nrOfSamples)
mcModel.run()
```

### 3.4.2 Percentiles, static data

The calculation of percentile data is almost just as easy.

Open `statistics.py` and add the following two lines to the postmcloop, below the other statements:

```
percentiles = [0.025, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.975]
pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)
```

Save the script and run. The functions `percentiles` works similar to `mcaveragevariance`, with the exception that it needs a list providing the percentiles that need to be calculated. In the `framework` folder, examine what the script has stored.

Create a probability density distribution plot of the vegetation height and the variable `b`. Click on the map to retreive the plot for different cell locations.

---

**Question 12:** What is the 25% percentile value of `b`? What is the probability that `b` exceeds 1.0?

a) The 25% percentile value is about -0.05, the exceedance probability for 1.0 is about 0.2 (values depend on cell location).

b) The 25% percentile value is about -0.5, the exceedance probability for 1.0 is about 0.2 (values depend on cell location).

c) The 25% percentile value is about -0.01, the exceedance probability for 1.0 is about 0.2 (values depend on cell location).

d) The 25% percentile value is about -0.005, the exceedance probability for 1.0 is about 2.0 (values depend on cell location).

Correct answer: b

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("clone.map")

    def premcloop(self):
        a = pcr.normal(1)
        self.report(a, "a")
        self.regions = self.readmap("regions")

    def initial(self):
        b = pcr.normal(1)
        self.report(b, "b")

        normalAreas = pcr.areanormal(self.regions)
        sdFactor = pcr.ifthenelse(self.regions, 0.05, pcr.scalar(0.1))
        meanFactor = pcr.ifthenelse(self.regions, 0.2, pcr.scalar(0.3))
        vegetationHeight = normalAreas * sdFactor + meanFactor
        self.report(vegetationHeight, "h")

        self.d = pcr.scalar(0)

    def dynamic(self):
        c = pcr.normal(1)
        self.report(c, "c")

    def postmcloop(self):
        names = ["b", "h"]
        sampleNumbers = self.sampleNumbers()
        print(sampleNumbers)
        timeSteps = [0]
        pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)

        percentiles = [
            0.025,
            0.05,
            0.1,
            0.2,
            0.3,
            0.4,
            0.5,
            0.6,
            0.7,
            0.8,
            0.9,
            0.95,
            0.975,
        ]
```

(continues on next page)

(continued from previous page)

```
        pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)

nrOfSamples = 100
nrOfTimeSteps = 10
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
mcModel = pcrfw.MonteCarloFramework(dynamicModel, nrOfSamples)
mcModel.run()
```

---

### 3.4.3 Dynamic data

A very similar approach can be used when calculating statistics over dynamic data.

Open `statistics.py`. Let's first create a more interesting dynamic variable. At the top of the dynamic is already written:

```
c = pcr.normal(1)
self.report(c, "c")
```

Now, type at the bottom of the dynamic:

```
self.d = self.d + c + 1
self.report(self.d, "d")
```

Initialize `self.d` in the initial by adding:

```
self.d = pcr.scalar(0)
```

Increase the number of time steps to 10. Save `statistics.py` and run the script. Check what is stored in the folder 1.

In the main folder `framework` animate a visualisation of 12 realizations of `c` and `d`. Use the Aguila options `--scenarios`, `--timesteps` and `--multi`.

---

**Question 13:** What happens with the uncertainty in `c` and `d` over time?

a) The uncertainty in `c` and `d` increases with time.

b) The uncertainty in `c` and `d` decreases with time.

c) The uncertainty in `c` is time-independent, while the uncertainty in `d` increases with time.

d) The uncertainty in `c` increases with time, while the uncertainty in `d` is time-independent.

Correct answer: c

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel, pcrfw.MonteCarloModel):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("clone.map")

    def premcloop(self):
        a = pcr.normal(1)
        self.report(a, "a")
        self.regions = self.readmap("regions")

    def initial(self):
        b = pcr.normal(1)
        self.report(b, "b")

        normalAreas = pcr.areanormal(self.regions)
        sdFactor = pcr.ifthenelse(self.regions, 0.05, pcr.scalar(0.1))
        meanFactor = pcr.ifthenelse(self.regions, 0.2, pcr.scalar(0.3))
        vegetationHeight = normalAreas * sdFactor + meanFactor
        self.report(vegetationHeight, "h")

        self.d = pcr.scalar(0)

    def dynamic(self):
        c = pcr.normal(1)
        self.report(c, "c")

        self.d = self.d + c + 1
        self.report(self.d, "d")

    def postmcloop(self):
        names = ["b", "h"]
        sampleNumbers = self.sampleNumbers()
        print(sampleNumbers)
        timeSteps = [0]
        pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)

        percentiles = [
            0.025,
            0.05,
            0.1,
            0.2,
            0.3,
            0.4,
            0.5,
            0.6,
            0.7,
            0.8,
            0.9,
            0.95,
            0.975,
        ]
        pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)
```

(continues on next page)

(continued from previous page)

```
nrOfSamples = 100
nrOfTimeSteps = 10
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
mcModel = pcrfw.MonteCarloFramework(dynamicModel, nrOfSamples)
mcModel.run()
```

---

To calculate the sample statistics, add the following block of code at the bottom of the postmcloop:

```
names = ["c", "d"]
timeSteps = self.timeSteps()
print(timeSteps)
pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)
pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)
```

Note that this is very similar to the code needed for static data. We only need to change the `names` and the `timeSteps`. Note that `self.timeSteps()` generates a list of the timesteps in the model. It is printed here.

Save the script and run. It will take some time to run as it needs to calculate statistics for each time step.

Type `dir c*` in the command prompt to see what is stored by the script. Also try `dir d*`.

Create probability distribution plots of `c` and `d`. Also, plot time series and animate.

---

**Question 14:** Set the map view of `d` to retrieve the exceedance probability of a threshold value of 5. What happens with this probability over time?

a) It increases over time up to a value of about 0.01 at the last time step.

b) It increases over time up to a value of about 0.1 at the last time step.

c) It increases over time up to a value of about 1 at the last time step.

d) It increases over time up to a value of about 0.5 at the last time step.

Correct answer: c

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("clone.map")

    def premcloop(self):
        a = pcr.normal(1)
        self.report(a, "a")
```

(continues on next page)

(continued from previous page)

```
        self.regions = self.readmap("regions")

    def initial(self):
        b = pcr.normal(1)
        self.report(b, "b")

        normalAreas = pcr.areanormal(self.regions)
        sdFactor = pcr.ifthenelse(self.regions, 0.05, pcr.scalar(0.1))
        meanFactor = pcr.ifthenelse(self.regions, 0.2, pcr.scalar(0.3))
        vegetationHeight = normalAreas * sdFactor + meanFactor
        self.report(vegetationHeight, "h")

        self.d = pcr.scalar(0)

    def dynamic(self):
        c = pcr.normal(1)
        self.report(c, "c")

        self.d = self.d + c + 1
        self.report(self.d, "d")

    def postmcloop(self):
        names = ["b", "h"]
        sampleNumbers = self.sampleNumbers()
        print(sampleNumbers)
        timeSteps = [0]
        pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)

        percentiles = [
            0.025,
            0.05,
            0.1,
            0.2,
            0.3,
            0.4,
            0.5,
            0.6,
            0.7,
            0.8,
            0.9,
            0.95,
            0.975,
        ]
        pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)

        names = ["c", "d"]
        timeSteps = self.timeSteps()
        print(timeSteps)
        pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)
        pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)
```

(continues on next page)

(continued from previous page)

```
nrOfSamples = 100
nrOfTimeSteps = 10
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
mcModel = pcrfw.MonteCarloFramework(dynamicModel, nrOfSamples)
mcModel.run()
```

---

## 3.5 Static modelling with point operations: forest fire

### 3.5.1 Introduction and gradient calculation

In a static model, we can calculate the time that the front of a forest fire reaches a certain cell location using weighted (relative) distance calculation, assuming the fire spreading is restricted to neighbouring cells in uphill direction only. This approach, further explained in the next section, requires a map with cell values representing the velocity of the forest fire spreading at that cell, given in hours per metre distance. The value of this velocity (h/m) is calculated as:

$h = be^{s/a}$

with, $b$, a parameter, $s$, the gradient of the topographical surface (m/m), $a$ a parameter, and $h$ velocity of forest fire spreading at the cell. The value of $b$ is known, $b = 0.002$, and it is assumed here that the gradient can be calculated from the digital elevation model (assumed to have zero uncertainty). The value of $a$ is known with a certain uncertainty, and thus $a$ is modelled as a stochastic parameter (constant over the whole area and over time) with a probability distribution, defined by a mean of 0.5 and a variance of 0.01.

Go to the folder `fire` and display all maps in the folder, including `dem.map`, the digital elevation model that can be used to calculate the gradient. Open the script `stochStaticMod.py`. Add statements to calculate a variable `gradient` containing the slope of the topographical surface, write to disk under the file name `gradient`. Save and run the script, and display `gradient`.

---

**Question 15:** Where did you type the statements?

a) In the premcloop.

b) In the initial.

c) In the postmcloop.

d) At the top of the script, below the from.. statements.

Correct answer: a

Feedback: The gradient is calculated in the premcloop method, because it is a deterministic map, that needs to be derived only before the Monte Carlo simulations.

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.StaticModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.StaticModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
```

(continues on next page)

(continued from previous page)

```
        pcr.setclone("clone.map")

    def premcloop(self):
        dem = self.readmap("dem")
        self.gradient = pcr.slope(dem)
        self.report(self.gradient, "gradient")

    def initial(self):
        pass

    def postmcloop(self):
        pass


nrOfSamples = 100
myModel = MyFirstModel()
staticModel = pcrfw.StaticFramework(myModel)
mcModel = pcrfw.MonteCarloFramework(staticModel, nrOfSamples)
mcModel.run()
```

---

### 3.5.2 Point operation to calculate fire front velocity

In `stochStaticMod.py`, add a statement that calculates realizations of $a$. Use the variable name `a` written to disk with a file name `a`. You will need to use the function `mapnormal`. Make sure that the value assigned to `a` is larger than 0. Save the script and run. Check the output.

Calculate the mean, variance, and percentiles of `a` in the postmcloop. Save the script and run. Display the output. Be sure to check whether the mean and variance of `a` are correct.

Calculate $h$, save the variable as `hpm` (i.e., hours per minute) and calculate mean, variance, and percentiles. Display the output.

---

**Question 16:** What is the probability density distribution of the velocity of the forest fire, i.e. what is the shape? Explain the shape using the equation used to calculate it.

a) The probability density distribution has a zero standard deviation, due to the exponent, reducing the variation.

b) The shape of the probability density distribution is normal, because the equation uses a normally distributed variable in the exponent of a natural logarithm.

c) The shape of the probability density distribution is log-normal, because the equation uses a normally distributed variable in the exponent of a natural logarithm.

d) The shape of the probability density distribution is uniform, which is due to the exponent.

Correct answer: c

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw
```

(continues on next page)

(continued from previous page)

```
class MyFirstModel(pcrfw.StaticModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.StaticModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("clone.map")

    def premcloop(self):
        dem = self.readmap("dem")
        self.gradient = pcr.slope(dem)
        self.report(self.gradient, "gradient")

    def initial(self):
        a = pcr.max(0.5 + pcr.mapnormal() / 10, 0.0001)
        self.report(a, "a")

        hoursPerMetre = 0.002 * pcr.exp(self.gradient / a)
        self.report(hoursPerMetre, "hpm")

    def postmcloop(self):
        names = ["a", "hpm"]
        sampleNumbers = self.sampleNumbers()
        print(sampleNumbers)
        timeSteps = [0]
        pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)
        percentiles = [
            0.025,
            0.05,
            0.1,
            0.2,
            0.3,
            0.4,
            0.5,
            0.6,
            0.7,
            0.8,
            0.9,
            0.95,
            0.975,
        ]
        pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)


nrOfSamples = 100
myModel = MyFirstModel()
staticModel = pcrfw.StaticFramework(myModel)
mcModel = pcrfw.MonteCarloFramework(staticModel, nrOfSamples)
mcModel.run()
```

**Question 17:** What is the variance and the standard deviation of the velocity of forest fire (h/m) at the grid cell location indicated on `house.map`? Write down and store, you will need it later on.

a) average = 0.045, variance = 0.43·10-9, standard deviation = 2·1-10

b) average = 0.45, variance = 4.3·10-9, standard deviation = 2·1-9

c) average = 0.0045, variance = 7.6·10-7, standard deviation = 8.7·10-4

d) average = 0.0045, variance = 4.3·10-9, standard deviation = 2·1-9

Correct answer: c

Feedback: None

---

---

**Question 18:** Display `hpm` together with the slope map `gradient`. Plot the 95% confidence interval for a (threshold) value of 0.005. A velocity below 0.005 is considered a high forest fire front spreading velocity. For what areas in terms of slope values is it not distinguishable whether the value is above or below the threshold?

a) For areas with a slope of about 0.1.

b) For areas with a slope of about 0.2.

c) For areas with a slope of about 0.5.

d) For areas with a slope of about 0.7.

Correct answer: c

Feedback: None

---

## 3.6 Static stochastic modelling: neighbourhood operations

### 3.6.1 Drawing realizations of a stochastic digital elevation model

In the previous exercise, you used the following equation to calculate the velocity of forest fire spreading at a given cell, given in hours per metre. It was calculated as:

$h = be^{s/a}$

with, $b$, a parameter, $s$, the gradient of the topographical surface (m/m), $a$ a parameter, and $h$ velocity of forest fire spreading at the cell.

We assumed only $a$ was unknown. However, as the digital elevation model will include error, we need to propagate the error in the digital elevation model to the output of the model $h$. Let's assume the error in the elevation (m) is for each pixel modelled by a stochastic variable $D$:

$D = e + Z$

with $e$ the digital elevation and $Z$ a stochastic variable with zero mean and a variance of 4.0.

In the folder `fire`, open `stochStaticMod.py` and save as `slope.py`. First, calculate realizations of $D$ in the initial. In the premcloop, you still need to read the deterministic dem from disk. However, you need to add `self.` as the map is needed in the initial. So, the premcloop should look like this:

```
self.dem = self.readmap("dem")
self.gradient = pcr.slope(self.dem)
self.report(self.gradient, "gradient")
```

In the initial, you need to calculate the realizations by adding realizations of $Z$ to the deterministic dem. As we need realizations on a cell-by-cell basis, you need to use `normal`. For $D$, use a variable name `demStoch` and write to disk as `dS`. Save the script and run.

Use Aguila to display 12 realizations of `dS`. Are the results approximately correct?

Now, calculate mean, variance, and percentiles of `dS`. Save the script and run. Display the variance map and the probability distribution plots.

---

**Question 19:** What is calculated by the script for the variance of `dS`? Is it similar to the value of 4.0 provided?

a) On average it is about 4, but it ranges between 3 and 5 depending on the cell location.

b) On average it is about 4, but it ranges between 1 and 7 depending on the cell location.

c) On average it is about 4, but it ranges between 3.9 and 4.1 depending on the cell location.

d) On average it is about 4, but it ranges between 3.99 and 4.01 depending on the cell location.

Correct answer: b

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.StaticModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.StaticModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("clone.map")

    def premcloop(self):
        self.dem = self.readmap("dem")
        self.gradient = pcr.slope(self.dem)
        self.report(self.gradient, "gradient")

    def initial(self):
        demStoch = self.dem + pcr.normal(1) * 2.0
        self.report(demStoch, "dS")

        a = pcr.max(0.5 + pcr.mapnormal() / 10, 0.0001)
        self.report(a, "a")

        hoursPerMetre = 0.002 * pcr.exp(self.gradient / a)
        self.report(hoursPerMetre, "hpm")

    def postmcloop(self):
        names = ["a", "hpm", "dS"]
        sampleNumbers = self.sampleNumbers()
        print(sampleNumbers)
        timeSteps = [0]
        pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)
        percentiles = [
```

(continues on next page)

(continued from previous page)

```
            0.025,
            0.05,
            0.1,
            0.2,
            0.3,
            0.4,
            0.5,
            0.6,
            0.7,
            0.8,
            0.9,
            0.95,
            0.975,
        ]
        pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)


nrOfSamples = 100
myModel = MyFirstModel()
staticModel = pcrfw.StaticFramework(myModel)
mcModel = pcrfw.MonteCarloFramework(staticModel, nrOfSamples)
mcModel.run()
```

---

### 3.6.2 Propagate DEM uncertainty through slope calculation

Using the stochastic digital elevation model defined in the previous section, you can now calculate the effect of this uncertainty on the uncertainty in the slope calculation. Thus far, slope was calculated in the premcloop. You need to move this calculation to the initial, using the realizations of the digital elevation model as input.

Open `slope.py` and make the required modifications. Be sure to calculate mean, variance, and percentiles of the slope. Save the script and run. Display the slope results to check your calculations.

---

**Question 20:** The uncertainty in fire spreading velocity of includes now the uncertainty of the parameter *a and* the uncertainty of the digital elevation model. Display `house.map` and the variance map of the forest fire spreading velocity (h/m). What is the variance and standard deviation of the forest fire spreading velocity (m/h) at the location of the house? Compare the result with the run where we only included error in the *a* parameter (previous exercise). What is the contribution to uncertainty in forest fire spreading of the uncertainty of the digital elevation model, approximately?

a) The uncertainty becomes zero.

b) The uncertainty has not changed (changes are due to numerical error).

c) The uncertainty is much lower now.

d) The uncertainty is much higher now.

Correct answer: d

Feedback: The uncertainty is much bigger. The variance is 1.696 x 10-6 and the standard deviation is 1.3 x 10-3.

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.StaticModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.StaticModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("clone.map")

    def premcloop(self):
        self.dem = self.readmap("dem")

    def initial(self):
        demStoch = self.dem + pcr.normal(1) * 2.0
        self.report(demStoch, "dS")

        gradient = pcr.slope(demStoch)
        self.report(gradient, "gradient")

        a = pcr.max(0.5 + pcr.mapnormal() / 10, 0.0001)
        self.report(a, "a")

        hoursPerMetre = 0.002 * pcr.exp(gradient / a)
        self.report(hoursPerMetre, "hpm")

    def postmcloop(self):
        names = ["a", "hpm", "dS", "gradient"]
        sampleNumbers = self.sampleNumbers()
        print(sampleNumbers)
        timeSteps = [0]
        pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)
        percentiles = [
            0.025,
            0.05,
            0.1,
            0.2,
            0.3,
            0.4,
            0.5,
            0.6,
            0.7,
            0.8,
            0.9,
            0.95,
            0.975,
        ]
        pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)


nrOfSamples = 100
myModel = MyFirstModel()
staticModel = pcrfw.StaticFramework(myModel)
```

(continues on next page)

(continued from previous page)

```
mcModel = pcrfw.MonteCarloFramework(staticModel, nrOfSamples)
mcModel.run()
```

---

### 3.6.3 Neighbourhood defined by topology: relative distance calculation

With the fire spreading velocity (h/m) you can calculate a map with the time it takes until the fire reaches a certain location, given a specified starting location of the fire. We oversimplify here somewhat, by assuming that fire burns only in uphill direction. The `ldddist` function can be used to calculate a relative distance, i.e. the absolute distance (m) that is travelled through a cell multiplied by a relative 'weighting' for that cell, here the fire spreading velocity (h/m). The result is the time (h) for the fire to reach a location. It restricts the distances calculated to uphill local drain directions.

Have a look at the description of the function `ldddist` in the PCRaster manual. Have a look at `start.map`, it is the starting point of a fire.

Open `slope.py` and save as `fire.py`. Make the following additions or changes:

- Read `start.map` from disk in the premcloop.
- Calculate the local drain direction map in premcloop and write to disk. Note that we herewith neglect the DEM uncertainty in the calculation of the relative distance to keep the exercise concise.
- In initial, use `ldddist` to calculate a map with the time the fire takes to reach a cell. As inputs, you need the local drain direction map, and the starting point of the fire, and the fire velocity map in hours per metre. Name the resulting map `time` and write to disk with report as `time`.
- In the postmcloop, calculate sampling statistics (mean, variance, percentiles) of fire.

Save the script and run. Display the mean, variance, and the probability density function of `time`, together with the local drain direction map and `house.map`.

---

**Question 21:** What is the mean and standard deviation of the time it takes until the fire reaches the house?

a) Mean: 2.5 h. Standard deviation: 0.30 h.

b) Mean: 2.5 h. Standard deviation: 0.92 h.

c) Mean: 2.5 h. Standard deviation: 3.7 h.

d) Mean: 4.5 h. Standard deviation: 0.37 h.

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.StaticModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.StaticModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("clone.map")
```

(continues on next page)

(continued from previous page)

```
    def premcloop(self):
        self.dem = self.readmap("dem")
        self.ldd = pcr.lddcreate(self.dem, 1e31, 1e31, 1e31, 1e31)
        self.report(self.ldd, "ldd")
        self.startPoint = self.readmap("start")

    def initial(self):
        demStoch = self.dem + pcr.normal(1) * 2.0
        self.report(demStoch, "dS")

        gradient = pcr.slope(demStoch)
        self.report(gradient, "gradient")

        a = pcr.max(0.5 + pcr.mapnormal() / 10, 0.0001)
        self.report(a, "a")

        hoursPerMetre = 0.002 * pcr.exp(gradient / a)
        self.report(hoursPerMetre, "hpm")

        time = pcr.ldddist(self.ldd, self.startPoint, hoursPerMetre)
        self.report(time, "time")

    def postmcloop(self):
        names = ["a", "hpm", "dS", "gradient", "time"]
        sampleNumbers = self.sampleNumbers()
        print(sampleNumbers)
        timeSteps = [0]
        pcrfwmcaveragevariance(names, sampleNumbers, timeSteps)
        percentiles = [
            0.025,
            0.05,
            0.1,
            0.2,
            0.3,
            0.4,
            0.5,
            0.6,
            0.7,
            0.8,
            0.9,
            0.95,
            0.975,
        ]
        pcrfwmcpercentiles(names, percentiles, sampleNumbers, timeSteps)


nrOfSamples = 100
myModel = MyFirstModel()
staticModel = pcrfw.StaticFramework(myModel)
mcModel = pcrfw.MonteCarloFramework(staticModel, nrOfSamples)
mcModel.run()
```

---

**Question 22:** Assume that it takes two hours (after the initiation of the fire) until the owners of the house can be informed they should leave the house. Create the confidence interval plot for a threshold of two hours, using an alpha value of 0.1. Can we be certain with this model that the owners can be informed in time?

a) Yes. The alpha value is below 2 hours, so we do even not need to run the model.

b) Well, it is unclear. The fire might be there within two hours, but it can also take longer.

c) No, according to the model we can be certain that the fire takes less than two hours to reach the house. The owners cannot be informed in time.

d) Yes, according to the model we can be certain that the fire takes more than two hours to reach the house. The owners can be informed in time.

Correct answer: d

Feedback: None

---

## 3.7 Dynamic stochastic modelling: snow melt model

### 3.7.1 Snow melt as a stochastic model

In the dynamic modelling exercises you created a deterministic snow melt model. Here we will built upon this work by propagating input errors to its outputs. As a start, the data set contains a script that is exactly the same as the script created in the deterministic modelling exercises, apart from that the script has been modified to be able to do stochastic modelling. Also, we will use smaller input maps to speed up calculations.

Go to the folder `snow` in the folder containing your exercise data, and open the script `snow.py`. Be sure to remind what calculations were included (if needed, have a look at the deterministic modelling exercises). It uses the stochastic dynamic modelling framework, and thus it will run for 50 Monte Carlo loops (as you can see at the bottom). However in the `snow.py`, no stochastic inputs are used (yet).

Run the script. Display the input timeseries, two realizations of snow cover and discharge, and check whether they are the same. Also display `dem.map` and you will see that we are using a smaller area now.

### 3.7.2 Stochastic dynamic input: precipitation

To represent uncertainty in precipitation, we define the following error model:

$p(\mathrm{t}) = p_{\mathrm{m}}(\mathrm{t})$ x $e(\mathrm{t})$

With $p_{\mathrm{m}}(\mathrm{t})$ the observed precipitation, $p(\mathrm{t})$ the stochastic variable precipitation (including uncertainty), and $e(\mathrm{t})$ a random non-spatial (i.e., the same for all cells) variable for each time step t with mean one and variance 0.04.

Open `snow.py` and save as `precip.py`. Modify `precip.py` to represent rainfall as a stochastic variable. You need to use the function `mapnormal`. And note that you will need to convert variance to standard deviation.

Save the script and run the model. Display 12 realizations of precipitation and snow cover. You need the `scenarios`, `multi`, and `timesteps` Aguila options. Animate them, and create timeseries for different locations.

---

**Question 23:** Do you think our error model results in a large error in the total amount of rain over the whole modelling period? Explain.

a) No, the errors are assigned each time step thus resulting in a large error (additive error).

b) Yes, the errors are assigned each time step thus resulting in a large error (additive error).

c) No, the errors cancel each other out as they are assigned each time step independently.

d) Yes, the errors cancel each other out as they are assigned each time step independently.

Correct answer: c

Feedback: None

---

To be able to create plots of probability density distributions, you can calculate the mean, variance, and percentiles in the postmcloop. Do this for precipitation, snow depth (`snow`), and discharge (`q`). To reduce the calculation time, use a limited set of percentiles:

```
percentiles = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
```

Save the `precip.py` script and run. Note that it will take some time at the end to calculate the percentiles. If the script is really too slow, you can consider using less Monte Carlo samples (e.g. 25 instead of 50), but note that this will result in a less precise calculation of the errors.

Use aguila to display and animate probability density distributions of precipitation, snow depth, and discharge, together with the map `locations.map`. Also, plot the variance of these attributes using Aguila, together with `locations.map`.

---

**Question 24:** What is the variance in snow depth at the two locations on `locations.map`, for time step 120 and 170 days? Write it down too.

a) Day 120: Location 1, 6.16, Location 2, 1.55; Day 170: Location 1, 7.90; Location 2, 1.12

b) Day 120: Location 1, 8.06 x 10-5, Location 2, 0.36 x 10-7; Day 170: Location 1, 0.12; Location 2, 0.0

c) Day 120: Location 1, 4.99, Location 2, 1.41; Day 170: Location 1, 6.44; Location 2, 1.35

d) Day 120: Location 1, 3.06 x 10-5, Location 2, 2.32 x 10-7; Day 170: Location 1, 0.0; Location 2, 0.0

Correct answer: d

Feedback: None

---

---

**Question 25:** What is the variance in discharge at the two locations on `locations.map`, for time step 120 and 170 days? Write it down too.

a) Day 120: Location 1, 0.01, Location 2, 0.00; Day 170: Location 1, 1.75 * 10^10; Location 2, 3.87 * 10^13

b) Day 120: Location 1, 0.04, Location 2, 0.00; Day 170: Location 1, 2.75 * 10^4; Location 2, 1.61 * 10^9

c) Day 120: Location 1, 0.09, Location 2, 0.00; Day 170: Location 1, 3.75 * 10^10; Location 2, 9.61 * 10^13

d) Day 120: Location 1, 0.00, Location 2, 0.00; Day 170: Location 1, 7.83 * 10^4; Location 2, 2.52 * 10^9

Correct answer: d

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("dem.map")

    def premcloop(self):
        dem = self.readmap("dem")
        elevationMeteoStation = 2058.1
        self.elevationAboveMeteoStation = dem - elevationMeteoStation
        self.ldd = pcr.lddcreate(dem, 1e31, 1e31, 1e31, 1e31)
        self.report(self.ldd, "ldd")

    def initial(self):
        temperatureLapseRate = 0.005
        self.temperatureCorrection = (
            self.elevationAboveMeteoStation * temperatureLapseRate
        )
        self.report(self.temperatureCorrection, "tempCor")

        self.snow = 0.0

    def dynamic(self):
        # precipitation = timeinputscalar('precip.tss',1)
        precipitationFactor = pcr.max(0, pcr.mapnormal() * 0.2 + 1.0)
        precipitation = pcr.timeinputscalar("precip.tss", 1) * precipitationFactor

        self.report(precipitation, "p")
        temperatureObserved = pcr.timeinputscalar("temp.tss", 1)
        temperature = temperatureObserved - self.temperatureCorrection
        self.report(temperature, "temp")

        freezing = temperature < 0.0
        snowFall = pcr.ifthenelse(freezing, precipitation, 0.0)
        rainFall = pcr.ifthenelse(pcr.pcrnot(freezing), precipitation, 0.0)

        self.snow = self.snow + snowFall

        potentialMelt = pcr.ifthenelse(pcr.pcrnot(freezing), temperature * 0.01, 0)
        actualMelt = pcr.min(self.snow, potentialMelt)

        self.snow = self.snow - actualMelt
        self.report(self.snow, "snow")

        runoffGenerated = actualMelt + rainFall

        discharge = pcr.accuflux(self.ldd, runoffGenerated * pcr.cellarea())
        self.report(discharge, "q")
```

(continues on next page)

(continued from previous page)

```
    def postmcloop(self):
        names = ["p", "q", "snow"]
        sampleNumbers = self.sampleNumbers()
        timeSteps = self.timeSteps()
        percentiles = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
        pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)
        pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)


nrOfTimeSteps = 181
nrOfSamples = 50
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
mcModel = pcrfw.MonteCarloFramework(dynamicModel, nrOfSamples)
# dynamicModel.run()
mcModel.run()
```

---

### 3.7.3 Stochastic parameters: temperature lapse rate

To represent uncertainty in the temperature lapse rate parameter, we replace the deterministic value of $l$ by a non-spatial (i.e. the same for all grid cells) stochastic variable with mean 0.005 and variance 0.000001.

Open `precip.py` and save as `lapse.py`. Modify the script using the stochastic $l$ instead of the deterministic fixed value. As the uncertainty in lapse rate has effect on the uncertainty in temperature, also calculate the mean, variance, and percentiles for `temp`. Run the script.

Display the probability density function of temperature, to see the effect of uncertain lapse rate on temperature.

---

**Question 26:** Write down the variance in snow depth at the two locations on `locations.map`, for time step 120 and 170 days. Compare the result with the version of the model that ignored uncertainty in lapse rate.

a) Day 120: Location 1, 3.91 x 10-5, Location 2, 8.02 x 10-7; Day 170: Location 1, 0; Location 2, 0

b) Day 120: Location 1, 4.99, Location 2, 1.41; Day 170: Location 1, 6.44; Location 2, 0.35

c) Day 120: Location 1, 3.94 x 10-8, Location 2, 6.02 x 10-8; Day 170: Location 1, 0; Location 2, 0

d) Day 120: Location 1, 5.16, Location 2, 4.41; Day 170: Location 1, 1.75; Location 2, 6.28

Correct answer: a

Feedback: None

```
import pcraster as pcr
import pcraster.framework as pcrfw


class MyFirstModel(pcrfw.DynamicModel, pcrfw.MonteCarloModel):
    def __init__(self):
        pcrfw.DynamicModel.__init__(self)
        pcrfw.MonteCarloModel.__init__(self)
        pcr.setclone("dem.map")
```

(continues on next page)

(continued from previous page)

```
    def premcloop(self):
        dem = self.readmap("dem")
        elevationMeteoStation = 2058.1
        self.elevationAboveMeteoStation = dem - elevationMeteoStation
        self.ldd = pcr.lddcreate(dem, 1e31, 1e31, 1e31, 1e31)
        self.report(self.ldd, "ldd")

    def initial(self):
        # temperatureLapseRate = 0.005
        temperatureLapseRate = 0.005 + pcr.mapnormal() * 0.001
        self.temperatureCorrection = (
            self.elevationAboveMeteoStation * temperatureLapseRate
        )
        self.report(self.temperatureCorrection, "tempCor")

        self.snow = 0.0

    def dynamic(self):
        precipitationFactor = pcr.max(0, pcr.mapnormal() * 0.2 + 1.0)
        precipitation = pcr.timeinputscalar("precip.tss", 1) * precipitationFactor
        self.report(precipitation, "p")

        temperatureObserved = pcr.timeinputscalar("temp.tss", 1)
        temperature = temperatureObserved - self.temperatureCorrection
        self.report(temperature, "temp")

        freezing = temperature < 0.0
        snowFall = pcr.ifthenelse(freezing, precipitation, 0.0)
        rainFall = pcr.ifthenelse(pcr.pcrnot(freezing), precipitation, 0.0)

        self.snow = self.snow + snowFall

        potentialMelt = pcr.ifthenelse(pcr.pcrnot(freezing), temperature * 0.01, 0)
        actualMelt = pcr.min(self.snow, potentialMelt)

        self.snow = self.snow - actualMelt
        self.report(self.snow, "snow")

        runoffGenerated = actualMelt + rainFall

        discharge = pcr.accuflux(self.ldd, runoffGenerated * pcr.cellarea())
        self.report(discharge, "q")

    def postmcloop(self):
        names = ["p", "q", "snow", "temp"]
        sampleNumbers = self.sampleNumbers()
        timeSteps = self.timeSteps()
        percentiles = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
        pcrfw.mcaveragevariance(names, sampleNumbers, timeSteps)
        pcrfw.mcpercentiles(names, percentiles, sampleNumbers, timeSteps)
```

(continues on next page)

(continued from previous page)

```
nrOfTimeSteps = 181
nrOfSamples = 50
myModel = MyFirstModel()
dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
mcModel = pcrfw.MonteCarloFramework(dynamicModel, nrOfSamples)
mcModel.run()
```

## 3.8 Dynamic stochastic modelling: infiltration model

### 3.8.1 Introduction

Important note: questions in this section are NOT available in Blackboard so please answer in a text document that you hand in afterwards. This section is also not part of the Land Surface Process Modelling course.

### 3.8.2 Model structure and field data

In the exercises, you will use a rainfall-runoff model simulating the Hortonian overland flow during a single rainstorm event. This section provides a short introduction to the model. Note that you need at least 2 GB diskspace for the exercises.

You will use data from an approximately rectangular 7500 m2 sandy loam hillslope with a vineyard, located in the Ouveze river basin, S. France. The model simulates rainfall, infiltration and overland flow. Interception is ignored here for reasons of simplicity (note that it is rather small anyway on a vineyard), as is surface storage.

Open the model `runoff.py` and have a look at the code. Run the model.

Display the input maps

```
aguila dem.map + ldd.map out.map + ldd.map &
```

The map *dem.map* is the digital elevation model of the modelling area, *ldd.map* is the local drain direction map with the drainage directions. *out.map* gives the outflow location of the catchment.

Rainfall is modelled as a block rainstorm (constant rainfall at the start of the modelling period abruptly finishing half way):

```
aguila --timesteps=[1,360,1] --scenarios={1} p
```

Right-click on the legend for options and to open a timeseries (click on map to get the values for a location).

It is assumed that for each cell and each time step, infiltration capacity (m/h) equals the saturated conductivity ($Z(\mathbf{s})$, m/h). For each cell, the actual infiltration ($I(\mathbf{s})$, m/h) is the minimum value of the saturated conductivity and the water available for infiltration (m/h). The water available for infiltration is rainfall ($P(\mathbf{s})$, m/h) plus runon from upstream cells ($R(\mathbf{s})$, m/h). In an equation:

$$I(\mathbf{s}) = \mathbf{min}(\mathbf{K}(\mathbf{s}), \mathbf{P}(\mathbf{s}) + \mathbf{R}(\mathbf{s}))$$

where min(a,b) assigns the minimum value of a and b.

Runoff is routed using the Manning equation and the kinematic wave in downstream direction over the local drain direction map (*ldd.map*).

Saturated conductivity was measured at 10 locations on the hillslope using ring infiltrometer experiments. The average saturated conductivity was 0.05 m/h (n=10), which is quite normal for this type of soil. The standard deviation of

the saturated conductivity data set was very high. In addition, discharge was measured at the outflow point of the catchment. With rainstorms having similar characteristics as the one used here, peak discharge was mostly about 0.02 m3/s. In the following exercises, you will try to simulate the discharge from the catchment, using an average K value of 0.02 m/h. The question is whether simulated discharge is in the same order of magnitude as measured discharge at the outflow point.

### 3.8.3 Deterministic modelling using mean K

The run with the model you did in the previous section was done using a saturated conductivity value of 0.05 m/h for all cells. This value is the same as the average value derived from the ring infiltrometre measurements. The question is how much runoff was generated by this run.

Let's first have a look at the change through time of actual infiltration (m/h, the files `i0000000.001`, representing the first time step, up to `i0000000.360`, representing the last time step), runoff (m3/h, `q0000000.001` up to `q0000000.360`), and actual infiltration as a percentage of the saturated conductivity (%, `ip000000.001`, ..). Note that these are written to the folder `1` but you do not need to care about this as it is indicated by `scenarios` in the aguila commane. Type from the folder where you were running the script:

```
aguila --timesteps=[1,360,1] --scenarios={1} i ip q + ldd.map &
```

and animate the maps. By right clicking on the legend you can open a time series view for a location on the map (just click on the map to change the location for which you want the time series). Note that the time step duration is 10 seconds. Click on the location of the outflow point to show the time series for that location.

---

**Question**

What is the peak discharge at the outflow point?

*Correct answers:* a.

*Feedback:*

0.000016 m3/h, which is a negligible amount

---

---

**Question**

Explain why such a small amount of discharge is generated (compared to the measured peak discharges for these kind of rainstorms).

*Correct answers:* Precipitation equals infiltration capacity and as a result everything infiltrates.

*Feedback:*

None

---

### 3.8.4 The stochastic model

The deterministic model used in the previous exercise does not generate runoff since it ignores spatial variation of saturated conductivity on the field. To give a more realistic representation of runoff, it is needed to include this spatial variation in our model. The problem is that there is insufficient data to represent this in a deterministic way, using a map with the actually occurring pattern of saturated conductivity on the field. The only way to solve the problem is to represent spatial variation of saturated conductivity as a stochastic variable, having a defined spatial probability distribution.

We assume a lognormal distribution of saturated conductivity by defining:

$$K(\mathbf{s}) = \mathbf{e}^{\mathbf{Z}(\mathbf{s})}$$

It is assumed that $Z(\mathbf{s})$ is a multivariable normal and stationary random spatial function. Thus, $Z(\mathbf{s})$ is defined by its expectation $m_{Z(\mathbf{s})}$, its variance $\sigma_{Z(\mathbf{s})}$ and its spatial correlation structure. The spatial correlation structure could be defined by a variogram, but we will use a simplified approach here to create random fields very similar to thos created with a variogram.

The expectation (mean) of $K(\mathbf{s})$ is:

$$m_{K(\mathbf{s})} = e^{m_{Z(\mathbf{s})}+0.5\sigma_{Z(\mathbf{s})}}$$

Increasing $\sigma_{Z(\mathbf{s})}$ results in a probability distribution of $K(\mathbf{s})$ with both a higher variance and a higher skewness.

The model can be run in stochastic mode, using user defined spatial probability distributions of $K(\mathbf{s})$. By modifying the following components of the script `runoff.py`, you can create scenario's with different spatial probability distributions of $K(\mathbf{s})$. In `runoff.py` you can define:

1) The number of Monte Carlo loops. The number of Monte Carlo loops is set in the following line at the upper part of the script:

```
nrSamples = 2
```

For instance, to run 50 loops, change the line to:

```
nrSamples = 50
```

2) Somewhat down, the value of $m_{K(\mathbf{s})}$:

```
# mean of Z
mean = -2.9957
```

3) The value of $\sigma_{Z(\mathbf{s})}$:

```
# variance of Z
var = 0.000001
```

4) The spatial correlation structure of $Z(\mathbf{s})$. The random field is generated by drawing independent realizations for each pixel. This results in a random field corresponding to a variogram without range (only nugget). Spatially correlated versions are created by applying a smoothing window. This creates realizations with a spatial correlation structure. The larger the window, the larger the scale of variation (larger variogram range). This is set by the value of `approximateRangeInPixels`. A value of one corresponds to no spatial correlation (nugget only variogram), other possible values are 3, 5, 7, etc. It is now set to 1 (no spatial correlation):

```
# approximate range of random field (pixels), use 1, 3, 5, 7, ..
# note that 1 implies 'white noise', i.e. no spatial correlation
approximateRangeInPixels = 1
```

Using the information given above, make a copy of `runoff.py` and save it as `runoff_stoch.py` to run the model with:

- 50 Monte Carlo loops
- white noise, no spatial correlation
- a value of $\sigma_{Z(\mathbf{s})}$ of 1.0
- an expectation (mean) of $K(\mathbf{s})$ of $m_{K(\mathbf{s})} = 0.05$ m/h

**Question:**

With the input values given above, what value needs to be used for $m_{Z(\mathbf{s})}$?

*Correct answers:* Use the equation in the course text to solve the equation. # mean of Z mean = -3.4957 # variance of Z var = 1.0

*Feedback:*

```
import shutil, sys
import pcraster.pcr as pcr
from pcraster import *
from pcraster.framework import *

# number of Monte Carlo samples
nrSamples = 50

# number of time steps
nrTimeSteps = 360

class Runoff(DynamicModel, MonteCarloModel):
  def __init__(self):
    DynamicModel.__init__(self)
    MonteCarloModel.__init__(self)
    setclone("clone.map")

  def initial(self):
    self.stepLength = 10               # Length of one timestep (s).

    dem = readmap("dem.map")
    self.landUnits = readmap("landunit.map")
    self.ldd = readmap("ldd.map")

    porosity = 0.4                     # Porosity (-).
    initMoistureContent = 0.05         # Initial moisture content.

    initInterceptStoreContent = 0      # (m, for area covered, 'Cov area').
    self.interceptCorrectionFactor = 1 # Correction factor canopy (0.046 * lai,
↪ is unit correct?).

    duration = 20                      # End of rainstorm (minutes).
    duration *= 60                     # End of rainstorm (s).

    self.precip = 0.017                # Precipitation (m).
    self.precip = (self.precip / duration) * self.stepLength # Per timestep␣
↪(m).

    self.lastPrecipStep = duration / self.stepLength  # End of rainstorm␣
↪(timesteps).

    self.cumPrecip = 0.0               # Cumulative rain (m).
```

(continues on next page)

(continued from previous page)

```
    initStreamFlow = 0.000000000001     # Initial streamflow (m3/s).
    self.beta = 0.6                     # Beta.
    areaFraction = 1                    # Area fraction of all channels ???.
    self.nrChannelsPerCell = 1          # Nr of channels per cell ???.


    # random field generation
    # random field with expectation zero and standard deviation one
    whiteNoise = normal(1)
    # approximate range of random field (pixels), use 1, 3, 5, 7, ..
    # note that 1 implies 'white noise', i.e. no spatial correlation
    approximateRangeInPixels = 1
    # number of pixels in window
    nrPixels = windowtotal(spatial(scalar(1)),
↪approximateRangeInPixels*celllength())
    self.report(nrPixels,"np")
    # std corrector
    factor = sqrt(nrPixels)
    # random field with expectation zero, standard deviation one, and range
    # equal to approximateRange
    randomField = windowaverage(whiteNoise,␣
↪approximateRangeInPixels*celllength()) * factor
    self.report(randomField,"rf")

    # mean of Z
    mean = -3.4957
    # variance of Z
    var = 1.0

    # standard deviation of Z
    std = sqrt(var)

    # random field Z
    Z = randomField * std + mean
    self.report(Z,"Z")

    ## INFILTRATION GREEN & AMPT
    # Sat. inf. rate (m/h).
    ks = exp(Z)
    self.report(ks, "ks")
    #meanKs = areaaverage(ks, "clone.map")
    #self.report(meanKs, "meanKs")

    # initial surface water
    surfaceWater = 0.0                  # Total amount of water on surface (m).

    # Saturated conductivity (m/timestep).
    self.ksPerStep = (ks / 3600) * self.stepLength

    # Actual initial infiltration per timestep ('rate', m/timestep).
    self.initInfil = 0.00000001
```

(continues on next page)

(continued from previous page)

```
    # Cumulative initial infiltration ('rate', m/timestep).
    self.cumInitInfil = 0.0000000001

    # Initial content of interception store (m, for area covered, 'Cov area').
    self.interceptStoreContent = initInterceptStoreContent

    lai = lookupscalar("lai.tbl", self.landUnits) # Leaf area index.

    # Maximum content of interception store (m, for area covered, 'Cov area'), m
    self.maxInterceptStoreContent = \
         max(0.935 + 0.498 * lai - 0.00575 * sqr(lai), 0.0000001) / 1000

    self.cumIntercept = 0.0       # Initial intercepted water, cumulative (m).

    # Area covered with vegetation.
    self.areaCovered = lookupscalar("cov.tbl", self.landUnits)

    # Amount of water in surface storage (m).
    self.surfaceWaterStorage = 0.0

    # Distance to downstream cell (m).
    self.distToDownstreamCell = max(downstreamdist(self.ldd), celllength())

    # Slope to downstream neighbour.
    slopeToDownStreamCell = \
         (dem - downstream(self.ldd, dem)) / downstreamdist(self.ldd)
    slope = cover(max(0.0001, slopeToDownStreamCell), 0.0001)

    self.discharge = initStreamFlow            # Initial streamflow (m3/s).
    self.cumQ = 0.0          # Cum amount of water added to streamflow (m).

    n = lookupscalar("n.tbl", self.landUnits)    # Manning's n.
    self.alphaTerm = (n / sqrt(slope)) ** self.beta        # Term for Alpha.
    self.alphaPower = (2.0 / 3) * self.beta      # Power for Alpha.

    waterDepth = 0.000000001                     # Initial water height (m).
    # Bottom width for routing (m).
    self.bottomWidth = areaFraction * celllength()

    # Initial approximation for Alpha.
    # Wetted perimeter (m) assume 8 channels per cell!
    wettedPerimeter = self.bottomWidth + 2 * self.nrChannelsPerCell *␣
↪waterDepth
    self.alpha = self.alphaTerm * (wettedPerimeter ** self.alphaPower)

    # Conversion factor from m/step to m/h.
    self.stepToHour = 3600 / self.stepLength

  def dynamic(self):
    # Rain per timestep, constant rain (m/timestep).
    precipPerStep = ifthenelse(self.currentTimeStep() <= self.lastPrecipStep,
         scalar(self.precip), 0.0)
```

(continues on next page)

(continued from previous page)

```
    # Rain (m/h).
    precipPerHour= precipPerStep * self.stepToHour * \
         scalar(defined(self.landUnits))
    self.report(precipPerHour, "p")

    self.cumPrecip = precipPerStep + self.cumPrecip   # Cumulative rain (m).

    # Intercepted water per timestep (m, spreaded over whole cell).
    intercept = precipPerStep * self.areaCovered

    # Intercepted water, cumulative (m, spreaded over whole cell).
    self.cumIntercept = self.cumIntercept + intercept

    # Amount in interception store (m, for area covered (for 'Cov area')).
    previousInterceptStoreContent = self.interceptStoreContent
    self.interceptStoreContent = self.maxInterceptStoreContent * (
         1 - exp((-self.interceptCorrectionFactor * self.cumPrecip) /
         self.maxInterceptStoreContent))

    # To interception store (m/timestep, for area covered (for 'Cov area')).
    toInterceptStore = self.interceptStoreContent - \
         previousInterceptStoreContent

    # To interception store (m/timestep, spreaded over whole cell).
    toInterceptStore = self.areaCovered * toInterceptStore

    # Throughfall (m, spreaded over whole cell).
    throughFall = intercept - toInterceptStore
    # self.report(throughFall, "TF")

    # Total net rain per timestep (m, spreaded over whole cell).
    netPrecip = throughFall + (precipPerStep - intercept)

    # Amount in interception store (m, spreaded over whole cell).
    interceptStoreContentCell = self.areaCovered * self.interceptStoreContent

    # Flow out off the cell (m/timestep).
    flowOutOfCell = (self.discharge * self.stepLength) / cellarea()
    # self.report(spatial(flowOutOfCell), "QR")

    # Total amount of water on surface, waterslice (m).
    surfaceWater = netPrecip + self.surfaceWaterStorage + flowOutOfCell
    # self.report(surfaceWater, "surfwat")

    # Cumulative infiltration (m).
    self.cumInitInfil = self.cumInitInfil + self.initInfil

    # Potential infiltration per timestep ('rate', m/timestep).
    potentialInfil = self.ksPerStep

    # Actual infiltration per timestep ('rate', m/timestep).
```

(continues on next page)

(continued from previous page)

```
    self.initInfil = ifthenelse(surfaceWater > potentialInfil, potentialInfil,
         surfaceWater)

    # Actual infiltration as percentage of potential infiltration (percent).
    percInfil = (self.initInfil / potentialInfil) * 100
    assert cellvalue(mapmaximum(percInfil), 1)[0] <= 100.0
    assert cellvalue(mapminimum(percInfil), 1)[0] >= 0.0
    self.report(percInfil, "ip")

    # Actual infiltration ('rate', m/h).
    self.report(self.initInfil * self.stepToHour, "i")

    # Total amount of water on surface after infiltration (m)
    surfaceWater = max(surfaceWater - self.initInfil, 0.0)

    self.surfaceWaterStorage = 0.0     # Amount of water in surface storage.
    fluxToSurfaceStorage = 0.0         # Flux to surface storage (m/timestep).

    # Amount of water on surface after infiltration and surface storage (m).
    surfaceWater = max(surfaceWater - self.surfaceWaterStorage, 0.0)

    # Amount of water added to streamflow (m/timestep).
    q = surfaceWater - flowOutOfCell

    # Cumulative amount of water added to streamflow (m).
    self.cumQ += q

    # Fluxes.
    # Rain minus (to interception store + act. infil + surface stor).
    fluxes = precipPerStep - (
         toInterceptStore + self.initInfil + fluxToSurfaceStorage+q)

    # Storages.
    # Rain minus (to interception store + act. infil + surface stor + avai ro).
    storages = self.cumPrecip - (
         interceptStoreContentCell + self.cumInitInfil +
         self.surfaceWaterStorage + self.cumQ)

    # Amount of water added to streamflow (m3/s).
    qAddedToStreamFlow = q * cellarea() / self.stepLength

    # Discharge (m3/s).
    self.discharge = kinematic(self.ldd, self.discharge,
         qAddedToStreamFlow / self.distToDownstreamCell, self.alpha, self.beta,
↪ 1,
         self.stepLength, self.distToDownstreamCell)
    self.report(self.discharge, "q")
    logDischarge = log10(self.discharge + 0.00001)
    self.report(logDischarge, "logq")

    # Water depth (m).
    waterDepth = self.alpha * (self.discharge ** self.beta) / self.bottomWidth
```

(continues on next page)

(continued from previous page)

```
    # Wetted perimeter (m).
    wettedPerimeter = self.bottomWidth + 2 * self.nrChannelsPerCell *␣
↪waterDepth

    # Alpha
    self.alpha = self.alphaTerm * (wettedPerimeter ** self.alphaPower)

  def postmcloop(self):
    # Variables to calculate statistics for
    names = ["q", "i", "ip"]
    # Sample numbers to calculate statistics for
    sampleNumbers = self.sampleNumbers()
    # Timesteps to calculate percentiles for.
    timeSteps = range(10, self.nrTimeSteps() + 1, 10)
    # Percentiles to calculate.
    percentiles = [0.05, 0.25, 0.50, 0.75, 0.95]

    # Calculate average and variance
    mcaveragevariance(names, sampleNumbers, timeSteps)
    # Calculate percentiles
    mcpercentiles(names,percentiles,sampleNumbers,timeSteps)

    # model inputs
    names = ["rf", "ks"]
    sampleNumbers = self.sampleNumbers()
    timeSteps = [0]
    mcaveragevariance(names, sampleNumbers, timeSteps)
    mcpercentiles(names,percentiles,sampleNumbers,timeSteps)


runoffModel = Runoff()
dynamicModel = DynamicFramework(runoffModel, nrTimeSteps)
mcModel = MonteCarloFramework(dynamicModel, nrSamples)
mcModel.run()
```

---

Write down all changes you have made to create `runoff_stoch.py`. You will need these in the next exercises!

Run the script `runoff_stoch.py`.

After running the script, display realizations of $K(\mathrm{s})$:

```
aguila --scenarios='{1,2,3,4,5,6,7,8,9,10,11,12}' --multi=3x4 ks
```

Right-click on the legend for options.

You can also display the cumulative probability distributions as they are calculated at the bottom of the script from the realisations:

```
aguila --quantiles=[0.05,0.95,0.05] ks
```

And the mean value:

```
aguila ks-ave.map
```

---

**Question**

Based on visual interpretation of the values and patterns on the maps of $K(\mathbf{s})$, do you think these agree with the parameters of the spatial probability distribution of $K(\mathbf{s})$ which you defined in the script?

*Correct answers:* Yes they agree, although a bit noisy due to the exponential distribution and the relatively small number of realizations.

*Feedback:* None

---

### 3.8.5 Evaluating individual realizations of the model

For each Monte Carlo loop, the rainfall-runoff model is run with the realization of $K(\mathbf{s})$ for that loop, resulting in a realization of all outputs of the model. To minimize the runtime of the model, only limited number of output variables is stored on the harddisk.

Animate the actual infiltration `i`, runoff `q`, actual infiltration as a percentage of K `ip`, and the saturated conductivity `ks`, for 12 Monte Carlo loops, type:

```
aguila --timesteps=[1,360,1] --scenarios='{1,2,3,4,5,6,7,8,9,10,11,12}' --multi=3x4 i +␣
↪ldd.map  q + ldd.map ip + ldd.map ks + ldd.map
```

Right click on the legend for options, e.g. to open a timeseries plot.

---

**Question**

While the deterministic model did not generate runoff, it is clear that most realizations of the stochastic model do generate runoff (using the same mean saturated conductivity). Explain this difference.

---

---

**Question**

In an animation, stop at time step 100. Explain in detail the relation between infiltration, runoff, and saturated conductivity for this timestep. Include the spatial patterns in your answer.

---

### 3.8.6 Evaluating the stochastic output

Since interpretation of individual realizations is difficult, it is better to calculate sampling statistics of the realizations. In a dynamic (temporal) model, this means that sample statistics are calculated for each time step (over all Monte Carlo loops). Files with these sample statistics are stored in the main directory (where your Python script is stored). The script calculates these statistics only for eacht 10-th time step to reduce runtime and storage.

Plot cumulative probability distributions of the discharge `q` (m3/s) and infiltration `i` (m/h):

```
aguila --timesteps=[10,360,10] --quantiles=[0.05,0.95,0.05] q + ldd.map i + ldd.map
```

Use the options explained previously in stochastic modelling to plot the timeseries of the 25 percentile at the outflow point. You will need to get the probability plot (right click on the menu).

---

**Question**

Explain how the 25 percentile of the hydrograph is calculated.

---

---

**Question**

Somebody has bought quite a small flume for discharge measurements to be installed at the outflow point of the catchment. The manual of the flume says it cannot measure discharge above 0.009 m3/s. During which period(s) of the rainstorm is it certain (using a 50% percent confidence interval) that the discharge at the outflow point is above 0.009 m3/s?

*Correct answers:* Between time step 97 and 185, approximately.

*Feedback:* None

---

**Question**

Plot the median discharge curve (50% percentile) at the outflow point. Write down the modelled median peak flow. Compare it with the measured peak flow of 0.05 m3/s. Do you think our model is 'good'?

*Correct answers:* 0.19 m3/h, not too bad

*Feedback:* 0.19 m3/h, not too bad

---

---

**Question**

Write down the boundaries (discharge, m3/h) of the 50% confidence interval of the peak discharge and calculate the width of the confidence interval.

*Correct answers:* 25% percentile: 0.016 m3/h, 75% percentile: 0.023. Width is 0.023-0.016 = 0.007 m3/h

*Feedback:* None

---

### 3.8.7 The effect of the distribution of saturated conductivity on discharge

It is interesting to study the relation between 1) the parameters describing the spatial probability distribution of $K(\mathbf{s})$ and 2) the hydrograph. Both the variance of $K(\mathbf{s})$ as well as the scale of the variation in $K(\mathbf{s})$ has an effect on the hydrograph.

Let's first look at the effect of the variance in saturated conductivity. Copy `runoff_stoch.py` and save as `runoff_stoch_low_var.py`. In `runoff_stoch_low_var.py`, make changes such that the value of $\sigma_{Z(\mathbf{s})}$ becomes 0.5 (it was 1.0). Keep the expectation (mean) of $K(\mathbf{s})$ at $m_{K(\mathbf{s})} = 0.05$ m/h. Note that you need to change both the values of `var` and `mean` in the script. Run the script.

---

**Question**

What is the effect of decreasing the variance of saturated conductivity on 1) the median and 2) the 50% confidence interval of the peak discharge?

*Correct answers:* median: 0.010 m3/h (lower) 25% percentile: 0.0078 m3/h, 75% percentile: 0.0126. Width is 0.0048 m3/h (lower)

*Feedback:*

```
import shutil, sys
import pcraster.pcr as pcr
from pcraster import *
from pcraster.framework import *

# number of Monte Carlo samples
nrSamples = 50

# number of time steps
nrTimeSteps = 360

class Runoff(DynamicModel, MonteCarloModel):
  def __init__(self):
    DynamicModel.__init__(self)
    MonteCarloModel.__init__(self)
    setclone("clone.map")

  def initial(self):
    self.stepLength = 10               # Length of one timestep (s).

    dem = readmap("dem.map")
    self.landUnits = readmap("landunit.map")
    self.ldd = readmap("ldd.map")

    porosity = 0.4                     # Porosity (-).
    initMoistureContent = 0.05         # Initial moisture content.

    initInterceptStoreContent = 0      # (m, for area covered, 'Cov area').
    self.interceptCorrectionFactor = 1 # Correction factor canopy (0.046 * lai,
↪ is unit correct?).

    duration = 20                      # End of rainstorm (minutes).
    duration *= 60                     # End of rainstorm (s).

    self.precip = 0.017                # Precipitation (m).
    self.precip = (self.precip / duration) * self.stepLength # Per timestep␣
↪(m).

    self.lastPrecipStep = duration / self.stepLength  # End of rainstorm␣
↪(timesteps).

    self.cumPrecip = 0.0               # Cumulative rain (m).


    initStreamFlow = 0.000000000001    # Initial streamflow (m3/s).
    self.beta = 0.6                    # Beta.
```

(continues on next page)

(continued from previous page)

```
    areaFraction = 1                    # Area fraction of all channels ???.
    self.nrChannelsPerCell = 1          # Nr of channels per cell ???.


    # random field generation
    # random field with expectation zero and standard deviation one
    whiteNoise = normal(1)
    # approximate range of random field (pixels), use 1, 3, 5, 7, ..
    # note that 1 implies 'white noise', i.e. no spatial correlation
    approximateRangeInPixels = 1
    # number of pixels in window
    nrPixels = windowtotal(spatial(scalar(1)),
↪approximateRangeInPixels*celllength())
    self.report(nrPixels,"np")
    # std corrector
    factor = sqrt(nrPixels)
    # random field with expectation zero, standard deviation one, and range
    # equal to approximateRange
    randomField = windowaverage(whiteNoise,␣
↪approximateRangeInPixels*celllength()) * factor
    self.report(randomField,"rf")

    # mean of Z
    mean = -3.2457
    # variance of Z
    var = 0.5

    # standard deviation of Z
    std = sqrt(var)

    # random field Z
    Z = randomField * std + mean
    self.report(Z,"Z")

    ## INFILTRATION GREEN & AMPT
    # Sat. inf. rate (m/h).
    ks = exp(Z)
    self.report(ks, "ks")
    #meanKs = areaaverage(ks, "clone.map")
    #self.report(meanKs, "meanKs")

    # initial surface water
    surfaceWater = 0.0                  # Total amount of water on surface (m).

    # Saturated conductivity (m/timestep).
    self.ksPerStep = (ks / 3600) * self.stepLength

    # Actual initial infiltration per timestep ('rate', m/timestep).
    self.initInfil = 0.00000001

    # Cumulative initial infiltration ('rate', m/timestep).
    self.cumInitInfil = 0.0000000001
```

(continues on next page)

(continued from previous page)

```
    # Initial content of interception store (m, for area covered, 'Cov area').
    self.interceptStoreContent = initInterceptStoreContent

    lai = lookupscalar("lai.tbl", self.landUnits) # Leaf area index.

    # Maximum content of interception store (m, for area covered, 'Cov area'), m
    self.maxInterceptStoreContent = \
         max(0.935 + 0.498 * lai - 0.00575 * sqr(lai), 0.0000001) / 1000

    self.cumIntercept = 0.0       # Initial intercepted water, cumulative (m).

    # Area covered with vegetation.
    self.areaCovered = lookupscalar("cov.tbl", self.landUnits)

    # Amount of water in surface storage (m).
    self.surfaceWaterStorage = 0.0

    # Distance to downstream cell (m).
    self.distToDownstreamCell = max(downstreamdist(self.ldd), celllength())

    # Slope to downstream neighbour.
    slopeToDownStreamCell = \
         (dem - downstream(self.ldd, dem)) / downstreamdist(self.ldd)
    slope = cover(max(0.0001, slopeToDownStreamCell), 0.0001)

    self.discharge = initStreamFlow            # Initial streamflow (m3/s).
    self.cumQ = 0.0          # Cum amount of water added to streamflow (m).

    n = lookupscalar("n.tbl", self.landUnits)    # Manning's n.
    self.alphaTerm = (n / sqrt(slope)) ** self.beta        # Term for Alpha.
    self.alphaPower = (2.0 / 3) * self.beta      # Power for Alpha.

    waterDepth = 0.000000001                     # Initial water height (m).
    # Bottom width for routing (m).
    self.bottomWidth = areaFraction * celllength()

    # Initial approximation for Alpha.
    # Wetted perimeter (m) assume 8 channels per cell!
    wettedPerimeter = self.bottomWidth + 2 * self.nrChannelsPerCell *␣
↪waterDepth
    self.alpha = self.alphaTerm * (wettedPerimeter ** self.alphaPower)

    # Conversion factor from m/step to m/h.
    self.stepToHour = 3600 / self.stepLength

  def dynamic(self):
    # Rain per timestep, constant rain (m/timestep).
    precipPerStep = ifthenelse(self.currentTimeStep() <= self.lastPrecipStep,
         scalar(self.precip), 0.0)

    # Rain (m/h).
```

(continues on next page)

(continued from previous page)

```
    precipPerHour= precipPerStep * self.stepToHour * \
         scalar(defined(self.landUnits))
    self.report(precipPerHour, "p")

    self.cumPrecip = precipPerStep + self.cumPrecip   # Cumulative rain (m).

    # Intercepted water per timestep (m, spreaded over whole cell).
    intercept = precipPerStep * self.areaCovered

    # Intercepted water, cumulative (m, spreaded over whole cell).
    self.cumIntercept = self.cumIntercept + intercept

    # Amount in interception store (m, for area covered (for 'Cov area')).
    previousInterceptStoreContent = self.interceptStoreContent
    self.interceptStoreContent = self.maxInterceptStoreContent * (
         1 - exp((-self.interceptCorrectionFactor * self.cumPrecip) /
         self.maxInterceptStoreContent))

    # To interception store (m/timestep, for area covered (for 'Cov area')).
    toInterceptStore = self.interceptStoreContent - \
         previousInterceptStoreContent

    # To interception store (m/timestep, spreaded over whole cell).
    toInterceptStore = self.areaCovered * toInterceptStore

    # Throughfall (m, spreaded over whole cell).
    throughFall = intercept - toInterceptStore
    # self.report(throughFall, "TF")

    # Total net rain per timestep (m, spreaded over whole cell).
    netPrecip = throughFall + (precipPerStep - intercept)

    # Amount in interception store (m, spreaded over whole cell).
    interceptStoreContentCell = self.areaCovered * self.interceptStoreContent

    # Flow out off the cell (m/timestep).
    flowOutOfCell = (self.discharge * self.stepLength) / cellarea()
    # self.report(spatial(flowOutOfCell), "QR")

    # Total amount of water on surface, waterslice (m).
    surfaceWater = netPrecip + self.surfaceWaterStorage + flowOutOfCell
    # self.report(surfaceWater, "surfwat")

    # Cumulative infiltration (m).
    self.cumInitInfil = self.cumInitInfil + self.initInfil

    # Potential infiltration per timestep ('rate', m/timestep).
    potentialInfil = self.ksPerStep

    # Actual infiltration per timestep ('rate', m/timestep).
    self.initInfil = ifthenelse(surfaceWater > potentialInfil, potentialInfil,
         surfaceWater)
```

(continues on next page)

(continued from previous page)

```
    # Actual infiltration as percentage of potential infiltration (percent).
    percInfil = (self.initInfil / potentialInfil) * 100
    assert cellvalue(mapmaximum(percInfil), 1)[0] <= 100.0
    assert cellvalue(mapminimum(percInfil), 1)[0] >= 0.0
    self.report(percInfil, "ip")

    # Actual infiltration ('rate', m/h).
    self.report(self.initInfil * self.stepToHour, "i")

    # Total amount of water on surface after infiltration (m)
    surfaceWater = max(surfaceWater - self.initInfil, 0.0)

    self.surfaceWaterStorage = 0.0     # Amount of water in surface storage.
    fluxToSurfaceStorage = 0.0         # Flux to surface storage (m/timestep).

    # Amount of water on surface after infiltration and surface storage (m).
    surfaceWater = max(surfaceWater - self.surfaceWaterStorage, 0.0)

    # Amount of water added to streamflow (m/timestep).
    q = surfaceWater - flowOutOfCell

    # Cumulative amount of water added to streamflow (m).
    self.cumQ += q

    # Fluxes.
    # Rain minus (to interception store + act. infil + surface stor).
    fluxes = precipPerStep - (
         toInterceptStore + self.initInfil + fluxToSurfaceStorage+q)

    # Storages.
    # Rain minus (to interception store + act. infil + surface stor + avai ro).
    storages = self.cumPrecip - (
         interceptStoreContentCell + self.cumInitInfil +
         self.surfaceWaterStorage + self.cumQ)

    # Amount of water added to streamflow (m3/s).
    qAddedToStreamFlow = q * cellarea() / self.stepLength

    # Discharge (m3/s).
    self.discharge = kinematic(self.ldd, self.discharge,
         qAddedToStreamFlow / self.distToDownstreamCell, self.alpha, self.beta,
↪ 1,
         self.stepLength, self.distToDownstreamCell)
    self.report(self.discharge, "q")
    logDischarge = log10(self.discharge + 0.00001)
    self.report(logDischarge, "logq")

    # Water depth (m).
    waterDepth = self.alpha * (self.discharge ** self.beta) / self.bottomWidth

    # Wetted perimeter (m).
```

(continues on next page)

(continued from previous page)

```
    wettedPerimeter = self.bottomWidth + 2 * self.nrChannelsPerCell *␣
↪waterDepth

    # Alpha
    self.alpha = self.alphaTerm * (wettedPerimeter ** self.alphaPower)

  def postmcloop(self):
    # Variables to calculate statistics for
    names = ["q", "i", "ip"]
    # Sample numbers to calculate statistics for
    sampleNumbers = self.sampleNumbers()
    # Timesteps to calculate percentiles for.
    timeSteps = range(10, self.nrTimeSteps() + 1, 10)
    # Percentiles to calculate.
    percentiles = [0.05, 0.25, 0.50, 0.75, 0.95]

    # Calculate average and variance
    mcaveragevariance(names, sampleNumbers, timeSteps)
    # Calculate percentiles
    mcpercentiles(names,percentiles,sampleNumbers,timeSteps)

    # model inputs
    names = ["rf", "ks"]
    sampleNumbers = self.sampleNumbers()
    timeSteps = [0]
    mcaveragevariance(names, sampleNumbers, timeSteps)
    mcpercentiles(names,percentiles,sampleNumbers,timeSteps)


runoffModel = Runoff()
dynamicModel = DynamicFramework(runoffModel, nrTimeSteps)
mcModel = MonteCarloFramework(dynamicModel, nrSamples)
mcModel.run()
```

---

Now let's look at the effect of the spatial pattern in saturated conductivity. Copy `runoff_stoch.py` and save it as `runoff_stoch_large_range.py`. Change the value of `approximateRangeInPixels` to a value of 3.

After running the script, display realizations of $K(\mathbf{s})$:

```
aguila --scenarios='{1,2,3,4,5,6,7,8,9,10,11,12}' --multi=3x4 ks
```

You will notice that the spatial pattern has changed to a spatially correlated pattern, with a larger patch size.

---

### Question

What is the effect of increasing the spatial range of saturated conductivity on 1) the median and 2) the 50% confidence interval of the peak discharge?

*Correct answers:* median: 0.027 m3/h (higher) 25% percentile: 0.017 m3/h, 75% percentile: 0.033. Width is 0.016 m3/h (higher)

*Feedback:*

```
import shutil, sys
import pcraster.pcr as pcr
import pcraster.framework as pcrfw

# number of Monte Carlo samples
nrSamples = 50

# number of time steps
nrTimeSteps = 360

class Runoff(DynamicModel, MonteCarloModel):
  def __init__(self):
    DynamicModel.__init__(self)
    MonteCarloModel.__init__(self)
    setclone("clone.map")

  def initial(self):
    self.stepLength = 10                # Length of one timestep (s).

    dem = readmap("dem.map")
    self.landUnits = readmap("landunit.map")
    self.ldd = readmap("ldd.map")

    porosity = 0.4                      # Porosity (-).
    initMoistureContent = 0.05          # Initial moisture content.

    initInterceptStoreContent = 0       # (m, for area covered, 'Cov area').
    self.interceptCorrectionFactor = 1 # Correction factor canopy (0.046 * lai,
↪ is unit correct?).

    duration = 20                       # End of rainstorm (minutes).
    duration *= 60                      # End of rainstorm (s).

    self.precip = 0.017                 # Precipitation (m).
    self.precip = (self.precip / duration) * self.stepLength # Per timestep␣
↪(m).

    self.lastPrecipStep = duration / self.stepLength  # End of rainstorm␣
↪(timesteps).

    self.cumPrecip = 0.0                # Cumulative rain (m).


    initStreamFlow = 0.000000000001     # Initial streamflow (m3/s).
    self.beta = 0.6                     # Beta.
    areaFraction = 1                    # Area fraction of all channels ???.
    self.nrChannelsPerCell = 1          # Nr of channels per cell ???.


    # random field generation
    # random field with expectation zero and standard deviation one
    whiteNoise = normal(1)
    # approximate range of random field (pixels), use 1, 3, 5, 7, ..
```

(continues on next page)

(continued from previous page)

```
    # note that 1 implies 'white noise', i.e. no spatial correlation
    approximateRangeInPixels = 3
    # number of pixels in window
    nrPixels = windowtotal(spatial(scalar(1)),
↪approximateRangeInPixels*celllength())
    self.report(nrPixels,"np")
    # std corrector
    factor = sqrt(nrPixels)
    # random field with expectation zero, standard deviation one, and range
    # equal to approximateRange
    randomField = windowaverage(whiteNoise,␣
↪approximateRangeInPixels*celllength()) * factor
    self.report(randomField,"rf")

    # mean of Z
    mean = -3.4957
    # variance of Z
    var = 1.0

    # standard deviation of Z
    std = sqrt(var)

    # random field Z
    Z = randomField * std + mean
    self.report(Z,"Z")

    ## INFILTRATION GREEN & AMPT
    # Sat. inf. rate (m/h).
    ks = exp(Z)
    self.report(ks, "ks")
    #meanKs = areaaverage(ks, "clone.map")
    #self.report(meanKs, "meanKs")

    # initial surface water
    surfaceWater = 0.0                 # Total amount of water on surface (m).

    # Saturated conductivity (m/timestep).
    self.ksPerStep = (ks / 3600) * self.stepLength

    # Actual initial infiltration per timestep ('rate', m/timestep).
    self.initInfil = 0.00000001

    # Cumulative initial infiltration ('rate', m/timestep).
    self.cumInitInfil = 0.0000000001

    # Initial content of interception store (m, for area covered, 'Cov area').
    self.interceptStoreContent = initInterceptStoreContent

    lai = lookupscalar("lai.tbl", self.landUnits) # Leaf area index.

    # Maximum content of interception store (m, for area covered, 'Cov area'), m
    self.maxInterceptStoreContent = \
```

(continues on next page)

(continued from previous page)

```
        pcr.max(0.935 + 0.498 * lai - 0.00575 * sqr(lai), 0.0000001) / 1000

    self.cumIntercept = 0.0       # Initial intercepted water, cumulative (m).

    # Area covered with vegetation.
    self.areaCovered = lookupscalar("cov.tbl", self.landUnits)

    # Amount of water in surface storage (m).
    self.surfaceWaterStorage = 0.0

    # Distance to downstream cell (m).
    self.distToDownstreamCell = pcr.max(downstreamdist(self.ldd), pcr.
↪celllength())

    # Slope to downstream neighbour.
    slopeToDownStreamCell = \
        (dem - downstream(self.ldd, dem)) / downstreamdist(self.ldd)
    slope = cover(max(0.0001, slopeToDownStreamCell), 0.0001)

    self.discharge = initStreamFlow            # Initial streamflow (m3/s).
    self.cumQ = 0.0          # Cum amount of water added to streamflow (m).

    n = lookupscalar("n.tbl", self.landUnits)    # Manning's n.
    self.alphaTerm = (n / sqrt(slope)) ** self.beta        # Term for Alpha.
    self.alphaPower = (2.0 / 3) * self.beta      # Power for Alpha.

    waterDepth = 0.000000001                    # Initial water height (m).
    # Bottom width for routing (m).
    self.bottomWidth = areaFraction * celllength()

    # Initial approximation for Alpha.
    # Wetted perimeter (m) assume 8 channels per cell!
    wettedPerimeter = self.bottomWidth + 2 * self.nrChannelsPerCell *␣
↪waterDepth
    self.alpha = self.alphaTerm * (wettedPerimeter ** self.alphaPower)

    # Conversion factor from m/step to m/h.
    self.stepToHour = 3600 / self.stepLength

  def dynamic(self):
    # Rain per timestep, constant rain (m/timestep).
    precipPerStep = ifthenelse(self.currentTimeStep() <= self.lastPrecipStep,
        scalar(self.precip), 0.0)

    # Rain (m/h).
    precipPerHour= precipPerStep * self.stepToHour * \
        scalar(defined(self.landUnits))
    self.report(precipPerHour, "p")

    self.cumPrecip = precipPerStep + self.cumPrecip   # Cumulative rain (m).

    # Intercepted water per timestep (m, spreaded over whole cell).
```

(continues on next page)

(continued from previous page)

```
    intercept = precipPerStep * self.areaCovered

    # Intercepted water, cumulative (m, spreaded over whole cell).
    self.cumIntercept = self.cumIntercept + intercept

    # Amount in interception store (m, for area covered (for 'Cov area')).
    previousInterceptStoreContent = self.interceptStoreContent
    self.interceptStoreContent = self.maxInterceptStoreContent * (
         1 - exp((-self.interceptCorrectionFactor * self.cumPrecip) /
         self.maxInterceptStoreContent))

    # To interception store (m/timestep, for area covered (for 'Cov area')).
    toInterceptStore = self.interceptStoreContent - \
         previousInterceptStoreContent

    # To interception store (m/timestep, spreaded over whole cell).
    toInterceptStore = self.areaCovered * toInterceptStore

    # Throughfall (m, spreaded over whole cell).
    throughFall = intercept - toInterceptStore
    # self.report(throughFall, "TF")

    # Total net rain per timestep (m, spreaded over whole cell).
    netPrecip = throughFall + (precipPerStep - intercept)

    # Amount in interception store (m, spreaded over whole cell).
    interceptStoreContentCell = self.areaCovered * self.interceptStoreContent

    # Flow out off the cell (m/timestep).
    flowOutOfCell = (self.discharge * self.stepLength) / cellarea()
    # self.report(spatial(flowOutOfCell), "QR")

    # Total amount of water on surface, waterslice (m).
    surfaceWater = netPrecip + self.surfaceWaterStorage + flowOutOfCell
    # self.report(surfaceWater, "surfwat")

    # Cumulative infiltration (m).
    self.cumInitInfil = self.cumInitInfil + self.initInfil

    # Potential infiltration per timestep ('rate', m/timestep).
    potentialInfil = self.ksPerStep

    # Actual infiltration per timestep ('rate', m/timestep).
    self.initInfil = ifthenelse(surfaceWater > potentialInfil, potentialInfil,
         surfaceWater)

    # Actual infiltration as percentage of potential infiltration (percent).
    percInfil = (self.initInfil / potentialInfil) * 100
    assert cellvalue(mapmaximum(percInfil), 1)[0] <= 100.0
    assert cellvalue(mapminimum(percInfil), 1)[0] >= 0.0
    self.report(percInfil, "ip")
```

(continues on next page)

(continued from previous page)

```
    # Actual infiltration ('rate', m/h).
    self.report(self.initInfil * self.stepToHour, "i")

    # Total amount of water on surface after infiltration (m)
    surfaceWater = max(surfaceWater - self.initInfil, 0.0)

    self.surfaceWaterStorage = 0.0     # Amount of water in surface storage.
    fluxToSurfaceStorage = 0.0         # Flux to surface storage (m/timestep).

    # Amount of water on surface after infiltration and surface storage (m).
    surfaceWater = max(surfaceWater - self.surfaceWaterStorage, 0.0)

    # Amount of water added to streamflow (m/timestep).
    q = surfaceWater - flowOutOfCell

    # Cumulative amount of water added to streamflow (m).
    self.cumQ += q

    # Fluxes.
    # Rain minus (to interception store + act. infil + surface stor).
    fluxes = precipPerStep - (
         toInterceptStore + self.initInfil + fluxToSurfaceStorage+q)

    # Storages.
    # Rain minus (to interception store + act. infil + surface stor + avai ro).
    storages = self.cumPrecip - (
         interceptStoreContentCell + self.cumInitInfil +
         self.surfaceWaterStorage + self.cumQ)

    # Amount of water added to streamflow (m3/s).
    qAddedToStreamFlow = q * cellarea() / self.stepLength

    # Discharge (m3/s).
    self.discharge = kinematic(self.ldd, self.discharge,
         qAddedToStreamFlow / self.distToDownstreamCell, self.alpha, self.beta,
↪ 1,
         self.stepLength, self.distToDownstreamCell)
    self.report(self.discharge, "q")
    logDischarge = log10(self.discharge + 0.00001)
    self.report(logDischarge, "logq")

    # Water depth (m).
    waterDepth = self.alpha * (self.discharge ** self.beta) / self.bottomWidth

    # Wetted perimeter (m).
    wettedPerimeter = self.bottomWidth + 2 * self.nrChannelsPerCell *␣
↪waterDepth

    # Alpha
    self.alpha = self.alphaTerm * (wettedPerimeter ** self.alphaPower)

  def postmcloop(self):
```

(continues on next page)

(continued from previous page)

```
    # Variables to calculate statistics for
    names = ["q", "i", "ip"]
    # Sample numbers to calculate statistics for
    sampleNumbers = self.sampleNumbers()
    # Timesteps to calculate percentiles for.
    timeSteps = range(10, self.nrTimeSteps() + 1, 10)
    # Percentiles to calculate.
    percentiles = [0.05, 0.25, 0.50, 0.75, 0.95]

    # Calculate average and variance
    mcaveragevariance(names, sampleNumbers, timeSteps)
    # Calculate percentiles
    mcpercentiles(names,percentiles,sampleNumbers,timeSteps)

    # model inputs
    names = ["rf", "ks"]
    sampleNumbers = self.sampleNumbers()
    timeSteps = [0]
    mcaveragevariance(names, sampleNumbers, timeSteps)
    mcpercentiles(names,percentiles,sampleNumbers,timeSteps)


runoffModel = Runoff()
dynamicModel = DynamicFramework(runoffModel, nrTimeSteps)
mcModel = MonteCarloFramework(dynamicModel, nrSamples)
mcModel.run()
```

For a detailed explanation of the simulated relations between saturated conductivity and infiltration (and thus discharge), refer to https://doi.org/10.1016/j.advwatres.2005.06.012

CHAPTER

FOUR

# DATA PRE-PROCESSING WITH GDAL

Download this website as pdf.

Download this website as epub (for e-readers).

To subscribe to our courses visit http://www.pcraster.eu

## 4.1 Introduction

In this tutorial, you will use GDAL to pre-process data from different sources to use in PCRaster-Python. GDAL is a software library for creating and manipulating geospatial data.

PCRaster operations work on raster datasets which have the same grid size and have the same spatial extent. GIS-data from different sources need to be clipped to the same region and resampled to the same grid size for usage in PCRaster models. Furthermore, when combining maps with a different map projections, maps have to be projected to a common map projection.

Besides GDAL and PCRaster-Python, you will use QGIS to visualize results. Both GDAL an QGIS needs to be installed on your computer, in addition to PCRaster-python. If you have QGIS on Windows, this comes with the OSGeo4W Shell. This is a set of command-line tools with all the required GDAL commands. Installation instructions for PCRaster-Python are available at http://www.pcraster.eu/, QGIS is available at https://www.qgis.org/

Instructions in this tutorial are focussed on using OSGeo4W on Windows. When using Linux, add `.py` to the following commands: `gdal_merge`, `gdal_edit` and `gdal_calc`.

An extended manual of the different utilities in GDAL is available at: http://www.gdal.org/gdal_utilities.html. You will need this manual for the assignments.

For this tutorial, you need several datasets. These datasets are available in a .zip-file on the Blackboard community of this course. Download and extract the .zip-file associated with this tutorial.

## 4.2 Merge, Crop en Project Elevation Model

### 4.2.1 Merge DEM tiles

In this section, you will use DEM tiles to create a single elevation model for a catchment in the Alps. The data used comes from the Alos World 3D DEM dataset. This is an elevation model at 30 m resolution that is available for the entire world. This dataset can be downloaded as individual tiles that together cover the world.

Open an OSGeo4W shell (or another terminal window with access to GDAL). In the shell, browse to the `DEM/Tiles` Folder and display the files in this folder:

```
cd C:\Exercises\dataprocessing\ <Enter>
dir DemTiles <Enter> [on Windows]
ls DemTiles <Enter> [on linux]
```

Adjust the folder location after cd to where the data is located. This will show a list of files in the folder. The folder contains the following files: `N046E008_AVE_DSM.tif`, `N046E007_AVE_DSM.tif`, `N045E008_AVE_DSM.tif`, and `N045E007_AVE_DSM.tif`. These files are already a selection of the original dataset to save space.

Open QGIS and start a new project. Add all four files to the project, these tiles are next to each other. In this tutorial, you will make a model for the Anza Catchment in Italy. The contour of this catchment is available in the file `anzacatchment.geojson`. Add this file to QGIS. You will see that the catchment is on the intersection of the four tiles.

Using the mouse-pointer and coordinate box at the bottom of the screen in QGIS, you can lookup coordinates on the map.

---

**Question 1:** Tiled datasets have systematic filenames, which makes it easy to retreive the right data for the right location. The filenames contain a latitude (N...) and longitude (E...). What coordinates do the nummers of the filename represent?

a) center of the tile

b) lower-left hand corner

c) upper-left hand corner

d) lower-right hand corner

e) upper-right hand corner

Correct answer: b

Feedback: The filename of the tiles indicate the lower-left hand corner of the tiles. Since adjecent tile-names increase with 1, all tiles have a size of 1x1 degree.

---

Switch back to the command prompt and merge the DEM-files using GDAL with the following command (on a single line). This command has to be executed from the `dataprocessing` folder: :

```
gdal_merge -of GTiff -o dem-merged.tif DemTiles/N046E008_AVE_DSM.tif
DemTiles/N046E007_AVE_DSM.tif DemTiles/N045E008_AVE_DSM.tif
DemTiles/N045E007_AVE_DSM.tif <Enter>
```

In this command, `-of GTiff` defines the output format of the merged DEM dataset. In this case GeoTiff, a widely used GIS dataformat. `-o dem_merged.tif` states the location and name of the output file. The last part of the command is list of filenames to be merged.

Now, add the merged file to QGIS to see the result of the merge operation. Adjust the visualisation to your liking.

### 4.2.2 Define target coordinate system

The next step is to crop the DEM to the modelling area (the catchment). Furthermore, the provided DEM is stored in a global coordinate system, WGS84. For modelling of smaller areas, it is more convenient to work in a local coordinate system. Both cropping and projecting can be done with one command.

First, we must determine which coordinate system to use for the final maps. For this tutorial, we will be using a UTM projection. This projection is available everywhere on Earth, but with different projection zones for different locations on Earth. To determine the right zone for our catchment, add the file `UTM-zones-EPSG-codes.geojson` to QGIS, and adjust the layer style so the polygons are transparent.

The UTM-Zones dataset shows all available UTM projections in the world on the map. If you zoom out to show a large part of Europe, you see there are adjacent strips on the map. The best projection to use is the one that contains the research area. For large areas that cover multiple UTM zones, other coordinate systems are better suited.

Use the identify tool in QGIS to determine the UTM zone. The UTM zone consist of a number and a letter (N or S) indicating if you are on the northern or southern hemisphere. Additionally, the UTM-zone dataset contains the associated EPSG identifier, which we use later to project the data.

---

**Question 2:** In which UTM-zone is the Anza Catchment located?

a) 31N

b) 32N

c) EPSG:32632

d) EPSG:32631

Correct answer: b

Feedback: The Anza catchment on the Italian-Suisse border is located in UTM zone 32, and is on the northern hemisphere, hence 32N. The EPSG identifier of this map-projection in EPSG:32632

---

### 4.2.3 Define target extent

Next, we need to convert the extent, or bounding box coordinates, of our catchment to the UTM coordinate system. So, we need to convert the lower-left and upper-right coordinate to the new coordinate system. In QGIS, you can see the coordinates of the mouse pointer in the Coordinate-box at the bottom of the screen. For the Anza Catchment, the lower-left and upper-right coordinates are approximately: 7.855, 45.895 and 8.265, 46.045, respectively. These coordinates are in the WGS84 reference system.

We can convert these coordinates to UTM coordinates using the program cs2cs, which is part of the GDAL command line tools:

```
cs2cs +proj=latlong +datum=WGS84 +to +init=EPSG:32632 <enter>
```

This command start a cs2cs session, in which you can enter coordinate-pairs. In this case (as defined by the above command), coordinates will be projected from Latitude-Longitude format (WGS84) to UTM ZONE 32N coordinates. Continue by entering the coordinates:

```
7.855 45.895 <enter>
8.265 46.045 <enter>
```

Close the cs2cs program by pressing `CTRL-C` simultaneous. The program has outputted two sets of coordinates. Note these two sets of coordinates. These consist of three values, but we will only need the first two (x and y). The last is elevation, which we do not use. The UTM coordinates are in meters. At a catchment-scale using no decimals is accurate enough.

---

**Question 3:** What is the y-coordinate in UTM Zone 32N for the coordinates 8.265, 46.045?

a) 411173

b) 5083019

c) 443133

d) 5099310

Correct answer: d

Feedback: If you got other results, check command and numbers.

---

### 4.2.4 Project and crop DEM

Now we know the coordinate system and bounding box of the target dataset. Next, we will project the merged DEM to the new coordinate system, and crop the DEM to our research area. We do this by using the GDAL command gdalwarp.

Analyse the following command:

```
gdalwarp -of gtiff -t_srs EPSG:32632 -te .. .. .. ..
-tr 90 90 -tap -r average dem-merged.tif dem-cropped-utm.tif <enter>
```

In this command, `-of gtiff` defines the output format (as before, we use GeoTiff), `-t_srs EPSG:32632` defines the target coordinate system. `-te .. .. .. ..` defines the extent coordinates of the final dataset. Here you have to enter the coordinates of the bounding box as defined before with cs2cs. In the GDAL manual for gdalwap (http://www.gdal.org/gdalwarp.html), determine how to input the coordinates for the '-te' argument.

---

**Question 4:** What values do you have to insert after -te .. .. .. .. in the gdalwarp command?

a) 7.855 45.895 8.265 46.045

b) 7.855 8.265 46.045 45.895

c) 411172 5083019 443134 5099310

d) 411172 443134 5099310 5083019

Correct answer: c

Feedback: By default, coordinates have to be entered in the target coordinate system, so with UTM coordinates. The syntax for the -te argument is '-te xmin ymin xmax ymax', So these are the x- and y-cordinates of the lower-left hand corner followed by the coordinates of the upper-right hand corner.

---

The argument `-tr 90 90` set a fixed cell size to 90x90 m. The argument `-tap` forces the output extent to a multiple of the grid size, this is important for PCRaster later as multiple input files need to be exactly aligned and of the same size. `-r average` tells gdal to average values during interpolation. This smooths the results a bit, but prevents artefacts in the final DEM.

Finalize the command above with the coordinates. Add the file `dem-cropped-utm.tif` to QGIS, and remove the larger DEM and DEM tiles if they are still present. If all went fine, the catchment boundary should fall just within the new DEM.

### 4.2.5 Convert DEM to PCRaster format

To use the DEM in PCRaster, it must be converted to the PCRaster file format. As stated in the introduction lecture on PCRaster, the way values are stored in a PCRaster map is very important. When converting to this format, the type of the data has to be defined explicitly by setting the so-called valuescale.

Enter and analyse the following command that converts the cropped DEM to the PCRaster format:

```
gdal_translate -ot Float32 -of PCRaster -mo PCRASTER_VALUESCALE=VS_SCALAR
dem-cropped-utm.tif dem.map <enter>
```

In this command, three things are important. We set `-of` (Output Format) to `PCRaster`, but the combination of `-ot` and `-mo` is different for different types of data. The following table shows which combinations of `-ot` and `-mo` you need for the various types of data that can be used in PCRaster. Since elevation data is scalar data, we use `Float32` and `VS_SCALAR`.

| Type of Data | -ot | -mo PCRASTER_VALUESCALE=... | -r |
|---|---|---|---|
| Boolean | Byte | VS_BOOLEAN | near |
| Nominal | Int32 | VS_NOMINAL | near |
| Ordinal | Int32 | VS_ORDINAL | near |
| Scalar | Float32 | VS_SCALAR | average |

The new pcraster map is accessible from PCRaster-Python, and we can view the map using aguila. Create a `testdem.py` file in the dataprocessing folder were dem.map is located, and add the following lines:

```
import pcraster as pcr
dem = pcr.readmap("dem.map")
slopemap = pcr.slope(dem)
pcr.aguila(slopemap)
```

If required use commands like `import os` and `os.chdir(r"C:\Exercises\dataprocessing")`. Run this file to display the slopemap of the catchment. In the following sections of this tutorial, you will create more input maps for the same research area to run a forest fire model.

## 4.3 NoData and Clone Maps

### 4.3.1 Remove values outside catchment

The DEM is a rectangle that is a little larger than the catchment, and there are still values outside the catchment. Using gdal, we can use the polygon of the catchment to wipe out all values outside the catchment. We use the command gdal_rasterize and gdal_edit to update the DEM:

```
gdal_rasterize -burn -32768 -i -at anzacatchment.geojson dem-cropped-utm.tif <enter>
gdal_edit -a_nodata -32768 dem-cropped-utm.tif
```

The gdal_rasterize uses the catchment polygon (`anzacatchment.geojson`) to edit the DEM. A value of -32768 is written to the map outside the polygon. The gdal_edit command sets this value as the 'nodata' value, indicating this part of the raster file contains no data.

---

**Question 5:** What is the purpose of the `-i` argument in the gdal_rasterize command?

a) To edit the values outside the catchment polygon.

b) To edit the values inside the catchment polygon.

c) To remove the previously set nodata value

d) To set the nodata value to -32768

Correct answer: a

Feedback: The option `-i` stands for invert. By default, gdal_rasterize edits the values inside the polygon. By using `-i` values outside the catchment polygon are overwritten with the new value of -32768.

---

Open the cropped dem again in QGIS, and remove the previous version. Run the `testdem.py` script again to see the result on the slopemap.

### 4.3.2 Create a clonemap

It is considered good practice to have a clonemap of your catchment for PCRaster scripts. Such map is a Boolean map with True-values inside your catchment and NoData-values outside your catchment.

In this tutorial, we first create a map with all zeros. We do this by multiplying the DEM with value 0. By using the DEM as a starting point, we ensure the new map is exactly of the same size as the DEM. This is essential of PCRaster modelling. First close any open maps in QGIS, then execute the following command:

```
gdal_calc --type Byte -A dem-cropped-utm.tif --outfile=clonemap.tif
--calc "A*0" --NoDataValue=0 <enter>
```

Then, similarly as in the previous section, we burn a value of 1 inside the polygon of the catchment:

```
gdal_rasterize -burn 1 -at anzacatchment.geojson clonemap.tif
```

For visualisation purposes, gdal can pre-compute statistics of the dataset. This step is required to correctly visualize the created `clonemap.tif` in QGIS:

```
gdal_edit -stats clonemap.tif
```

Finally, we need to convert this raster file to a PCRaster format, using the command gdal_translate. Like before, we need to select the right parameters for `-ot` and `-mo`.

---

**Question 6:** For the clone map, which are the right parameter values for `-ot` and `-mo` in the gdal_translate command?

a) -ot Byte, -mo PCRASTER_VALUESCALE=VS_BOOLEAN

b) -ot Boolean, -mo PCRASTER_VALUESCALE=VS_NOMINAL

c) -ot Byte, -mo PCRASTER_VALUESCALE=VS_NOMINAL

d) -ot Boolean, -mo PCRASTER_VALUESCALE=VS_BOOLEAN

Correct answer: a

Feedback: A clonemap in PCRaster is stored as a Boolean map. In PCRaster, the output type (`-ot`) of such map is Byte, and the corresponding `PCRASTER_VALUESCALE` is `VS_BOOLEAN`. Note, that the output type Boolean does not exist! See the table earlier in this document.

---

Complete the following command by replacing … with your answer from the previous question:

```
gdal_translate -ot ... -of PCRaster -mo ... clonemap.tif clone.map
```

Adjust the content of the script `testdem.py` to read and display clone.map.

---

**Question 7:** In the clone.map, as shown in aquila. What are the values inside the catchment, and what are the values outside the catchment?

a) inside: true, outside: false.

b) inside: 1, outside: 0.

c) inside: true, outside: no data.

d) inside: 1, outside: no data.

Correct answer: c

Feedback: The clonemap has no data outside the catchment and the value 'true' inside the catchment.

---

## 4.4 Forest Fire Model

In the chapter 'Dynamic modelling', you have used a simple forest-fire model on an arbitrary map. In this section, you will use the data from the previous sections and additional data to model forest fire in a real catchment.

### 4.4.1 Global land cover data

In this section, we will convert two datasets to the same extent and projection as the dem we created earlier. The procedure is the same, but we have to deal with different data types again.

The two datasets we will use are classified land cover data from the Global Land Cover (GLC30) dataset, and tree cover fraction data from the USGS Global Land Cover datasets. Although these datasets have a similar name, they are maintained by different instituted (see the section at the end of this chapter with information on the used data).

First, we will project and crop the available tiles to our catchment. We use the same extent and coordinate system as earlier for the DEM:

```
gdalwarp -of gtiff -t_srs EPSG:32632 -te 411172 5083019 443134 5099310
-tr 90 90 -tap -r near GlobalLandCover/n32_45_2010lc030.tif landcover-crop.tif <enter>

gdalwarp -of gtiff -t_srs EPSG:32632 -te 411172 5083019 443134 5099310
-tr 90 90 -tap -r average LandCover/50N_000E_treecover2010_v3.tif treecover-crop.tif
↪<enter>
```

Open the tiles `landcover-crop.tif` and `treecover-crop.tif` in QGIS.

---

**Question 8:** What are the data types of `landcover-crop.tif` and `treecover-crop.tif`?

a) landcover: Nominal, treecover: Ordinal

b) landcover: Scalar, treecover: Scalar

c) landcover: Scalar, treecover: Ordinal

d) landcover: Nominal, treecover: Scalar

Correct answer: d

Feedback: The landcover dataset contains classified data (e.g. 'Forest' or 'Grassland'), represented by a number. Such scale is a Nominal Scale. The treecover dataset contains fractional values (between 0 to 1) of the fraction of trees. Such continuous scale is stored as Scalar values.

---

Based on the datatypes of the different datasets, complete the following commands to convert the landcover and treecover datasets to PCRaster maps:

```
gdal_translate -ot ... -of PCRaster -mo PCRASTER_VALUESCALE=...
landcover-crop.tif landcover.map <enter>

gdal_translate -ot ... -of PCRaster -mo PCRASTER_VALUESCALE=...
treecover-crop.tif treecover.map <enter>
```

### 4.4.2 Biomass and land use

There are a few updated aspects of the forest fire model, because we have access to the landcover and treecover data. We can now use a spatially variable probability for a cell catching fire, based on the land cover. For example, forest gets a higher probability of catching fire than shrubs. Secondly, the treecover dataset is used as a proxy for biomass, which diminishes when a cell is on fire. The lower the biomass, the higher the probability that fire stops in that cell.

The updated model is available in the python script fire.py. Several report functions are commented out in this script to prevent an overload of model output. Feel free to change parts of the model. Comments in the dynamic section with the text 'Updated' indicate parts of the model that are new compared the model you used earlier

### 4.4.3 Fire start location

Have a look at the initial section the `fire.py script`. From all the `.map` files required by the `self.readmap` commands, we already created all expect `firestart.map`. This map is a Boolean map with only one True cell, which indicates the starting position of the fire. The file `firestart.gpkg` in the folder FireStart contains a vector point with a fictive starting location of our forest fire.

This point dataset can be converted to a raster with similar commands as before. We convert the DEM to all zeros using gdal_calc, then burn in a value of 1 at the location of the point:

```
gdal_calc --type Byte -A dem-cropped-utm.tif --outfile=firestart.tif --calc
"A*0" --NoDataValue=0 <enter>
gdal_rasterize -burn 1 -at FireStart/firestart.gpkg firestart.tif <enter>
gdal_edit -stats firestart.tif <enter>
gdal_translate -ot Byte -of PCRaster -mo PCRASTER_VALUESCALE=VS_BOOLEAN
firestart.tif firestart.map <enter>
```

### 4.4.4 Land use to fire probability

The final file we need is the `landcover-fireprob.txt` file, used in the lookupscalar command. This file is lookuptable that converts the landcover map to probabilities of catching fire. The landcover dataset contains the following values, which associated land use types:

| nr | Land use |
|---|---|
| 10 | Cultivated land |
| 20 | Forest |
| 30 | Grassland |
| 40 | Shrubland |
| 50 | Wetlands |
| 60 | Water bodies |
| 70 | Tundra |
| 80 | Artificial surface |
| 90 | Bare land |
| 100 | Permanent snow/ice |

To show which classes are present in the study area, open the landcover map in aquila:

```
aguila landcover.map <enter>
```

Create a text-file with the name `landcover-fireprob.txt` using a text editor (e.g. notepad on Windows or leafpad or gedit on Linux). In this file, create a lookuptable to convert all the landcover classes to probabilities of catching fire. Below in an example / start, complete this text file yourself. Several class get a value of 0 (like snow and water) and more forested land types get a higher value (approximately 0.2 at the most).

```
10  0.05
20  0.2
30  0.05
```

If your model is finished, run the script. The model creates timeseries, which can be visualized using aguila from the command-line. Note that the OS4GeoW does not by default include the pcaster command, so you may need to run these commands from a command-line terminal which does support pcraster. For example:

```
aguila --timesteps=[1,200,1] fire <enter>
aguila --timesteps=[1,200,1] tree <enter>
```

## 4.5 Common errors

This list shows a few common GDAL-errors, and how to solve them.

```
ERROR 1: Output dataset data.tif exists
```

Cause: the file is already there. Remove the file first with the command del data.tif. If that does not work, the file is likely still open in a software package like QGIS. If that is the case, close the file first and try again.

```
ERROR 4: data.tif: No such file or directory
```

Cause: the file you are trying to read does not exist. Probably a typo in the name, or you are not in the right folder.

## 4.6 Data sources

The following data sources have been used in this tutorial. These are all global datasets at 30 m resolution which are free to use to create your own model. In most cases, you should register for an account, and the data comes in small tiles.

- USGS 30 Meter Global Land Cover: https://landcover.usgs.gov/glc/
- Global Land Cover, GLC30L http://www.globallandcover.com/GLC30Download/index.aspx
- Alos World 3D 30m DEM (AW3D30) http://www.eorc.jaxa.jp/ALOS/en/aw3d30/

CHAPTER

FIVE

# PROGRAMMING WITH PYTHON

Download this website as pdf.

Download this website as epub (for e-readers).

To subscribe to our courses visit http://www.pcraster.eu

## 5.1 Starting python, command line and programming mode

### 5.1.1 Starting up Python, the command line

In Microsoft Windows, start the python interpreter by selecting Start, Programs, Python 3.x, IDLE (Python GUI). This is the command line mode of Python. Type:

```
print(12 + 15) <Enter>
a = 12.5       <Enter>
b = 2.0        <Enter>
print(a * b)   <Enter>
print(a / b)   <Enter>
```

In this mode, only single statements can be entered. It is not possible to ask Python to execute a set of statements at once.

### 5.1.2 Executing Python scripts

Below is a Python script (or program).

```
print(12 + 15)
a = 12.5
b = 2.0
print(a * b)
print(a / b)
```

From the Python program, select File, New Window, and an ASCII editor pops up. Copy-paste the python script from the table above in the editor window. Use File -> Save As to save the script in a separate directory (e.g. *python_scripts*) on your harddisk, using the filename *intro.py*.

Run the script by selecting Run from your Python program. Now, you have executed your first python program. The python interpreter has executed all statements in the script file, printing the three outcomes of the calculations given in the script.

## 5.2 Variables, expressions and statements

### 5.2.1 Values and Types

Below is an extended version of the script used in the previous section.

```
print(12 + 15)
a = 12.5
b = 2.0
print(a * b)
print(a / b)
# print the type of b
print(type(b))
```

Create this script using copy-paste to the Python script editor, or just add the two additional lines to the script which you used in the previous exercise. Note that the note print the type of b is typed after a #. This so-called comment will be ignored by the Python interpreter. Save the script as `types.py` and execute it.

---

**Question 1:** What is the type of the variable b?

a) floating-point

b) integer

Correct answer: a

Feedback: The variable is defined by adding the decimal value, that is `2.0`. Typing `2` would have defined the variable as an integer.

---

Add the following lines to `types.py`:

```
message = "Hello World!"
c = 2
```

And add two statements that give you the type of the variables `message` and `c`. Save the script and execute it. Note that the type of `c` is different from `b`, since `b` contains a dot (`.`).

Also note that variable names such as `message`, `b`, and `a`, are case sensitive. Try this by adding the following statement to `types.py`, at the bottom, saving it, and executing:

```
print(C)
```

All statements in the script file are executed, apart from the last one. The interpreter gives you the line number with the error: it is the last line. You tried to print `C`, but it is not available, since lowercase `c` is defined, while uppercase `C` isn't.

Also add commands that give you the type of `5 + "PCRaster"`, `2/5`, `2.0/5`, `2/5.0`, and `2.0/5.0`.

### 5.2.2 Order of operations

The velocity of flowing water in a channel ($v$, m/s) is dependent upon the flow depth or hydraulic radius ($r$, m), the roughness of the surface and the slope ($s$, -). This relationship is commonly expressed by the Manning equation:

$v = (r^{2/3} s^{1/2}) / n$

where $n$ is the Manning coefficient of roughness (between 0.01 for smooth surfaces up to 0.8 for rough surfaces).

Create a Python script (save it as `manning.py`) printing the flow velocity as a function of the hydraulic radius, the Manning coefficient and slope. Use three statements assigning values to variables representing the hydraulic radius, roughness of the surface and slope. Use the following values for the variables: hydraulic radius: 3 m, slope 0.1, Manning's n 0.01. Use intuitive variable names (e.g. `hydraulic_radius`) and provide comments for each statement (the units used). Use one statement assigning the flow speed to another variable using the equation given above. Finally, print this variable. Execute the script.

---

**Question 2:** What is the flow speed in m/s for the input values given in the text?

a) 65.78

b) 315.34

c) 270.92

d) 376.2

Correct answer: a

Feedback: A correct script is given below (note the comments). The order of calculation is explicitly defined here using brackets. Look up your Python book which brackets can be removed, to make the program easier readible.

```
# hydraulic radius (m)
hydraulic_radius = 3.0
# slope (-)
slope = 0.1
# manning's n
n = 0.01
# flow speed
v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
# print flow speed
print(v)
```

---

### 5.2.3 Operations on strings and composition

Modify the `manning.py` script (see previous section) such that it prints:

```
With a hydraulic radius of ? m, the flow speed is ? m/s.
```

At the questions marks, the script should print the values of the input variable used for hydraulic radius and the resulting flow speed. Use the Python format string syntax. Save and execute the script to test it.

---

**Question 3:** How did you print *With a hydraulic radius*? By defining it as as:

a) Floating point

b) Integer

c) String

Correct answer: c

Feedback: None

```
# hydraulic radius (m)
hydraulic_radius = 3.0
# slope (-)
slope = 0.1
# manning's n
n = 0.01
# flow speed
v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
# print flow speed
print(v)
# print flow speed with sentence
print(f"With a hydraulic radius of {hydraulic_radius} (m) the flow speed is {v} m/s.")
```

## 5.3 Functions

### 5.3.1 Keyboard input, types of variables and type conversion

Copy-paste the script in the table below to your editor, save it as `name2.py`, and execute it. The function `input` reads keyboard input and assigns it to a variable (here, `name`).

```
# program for greeting people
name = input("What is your name?\n")
print("Hello, " + name + "! How are you?")
```

What is the difference between using + and `,` in a print statement? Try it!

Copy-paste the script in the table below to your editor, save it as `slope.py`, and execute it. The script uses a formatted string literal (f-string) for string formatting. It is recommended to use f-strings from now on to format output.

```
# program to enter the slope in degrees
slope = input("Enter the slope in degrees ")
print(f"The slope you entered is {slope} degrees")
```

Add a statement that prints the type of `slope`. Execute the program with different inputs (e.g., 20, 35.2, 0). What is the type of the return value (here: `slope`) of the function `raw_input`?

If you want to do calculations with `slope` (see next section), you need `slope` as a floating-point value. Convert the type of `slope` and assign the result to a new variable by adding the line

```
slopeFloat = float(slope)
```

Add another line to the script printing the type of `slopeFloat` and see whether it is converted.

### 5.3.2 Math functions

The Universal Soil Loss Equation contains the S factor ($S$) representing the effect of slope steepness ($s$, m/m) on soil erosion. The equation is:

$S = 0.065 + 0.045\ s + 0.0065\ s^2$

Open the Python script `slope.py` (see previous section) and save it as `s.py`. Now modify it, resulting in a script that

- asks for an input value of the slope steepness (in degrees),
- prints the S factor.

Note that you will need type conversion, and the `math` module to convert from degrees to radians, and to calculate the slope as a fraction (m/m, as used in the equation) from the slope in radians (entered by the user). The Python documentation gives a list of all functions in the math module. Let the program print intermediate values of variables, in addition to the final result. Be sure to check the answer created by the script before answering the next question!

---

**Question 4:** Run your script and enter an input value of 20 for the slope in degrees. What is $S$?

a) 0.032

b) 0.82

c) 0.082

d) 0.32

Correct answer: c

Feedback: None

```
import math

# program to calculate the s factor in the USLE

slope = input("Enter the slope in degrees ")
print(f"The slope you entered is {slope} degrees.")
slopeFloat = float(slope)

# slope in radians
slopeRadians = (slopeFloat / 360) * 2.0 * math.pi
print(f"This corresponds to a slope of {slopeRadians} in radians.")

# slope in m/m
slopeFraction = math.tan(slopeRadians)
print(f"This corresponds to a slope of {slopeFraction} (m/m)")

# slope factor
S = 0.065 + 0.045 * slopeFraction + 0.0065 * slopeFraction**2.0

print(f"The slope factor is {S}")
```

---

## 5.4 Conditionals and recursion

### 5.4.1 Boolean expressions: comparison operators

The program below uses a comparison operator to compare two floating-point values.

```
x = 12.4
y = 15.2
xIsGreaterThanY = x > y
print(xIsGreaterThanY)
```

Copy-paste the program to your script editor and save as `comp.py`. Execute it. Add a statement that prints the type of `x`, `y`, and `xIsGreaterThanY`.

---

**Question 5:** What is the type of `xIsGreaterThanY`?

a) Integer

b) Floating point

c) String

d) Boolean

Correct answer: d

Feedback: None

```
x = 12.4
y = 15.2

xIsGreaterThanY = x > y

print(xIsGreaterThanY)
print(type(xIsGreaterThanY))
```

---

Copy-paste the script below to your editor, save it as comp_eq.py, and execute it. What is the meaning of ==? Is the outcome what you expected?

```
x = 12
y = 12
xIsEqualToY = x == y
print(xIsEqualToY)
```

Run the script.

Change the first two lines in `comp_eq.py` to:

```
x = 12.0001
y = 12.0
```

As you know this has changed the type of `x` and `y` to floating point. Execute the script. Does it return the value for xIsEqualToY that you expected?

Now, assign to `x` a value of 12.000000000000000001 (note: this is 17 zeros after the dot). Execute the script. You will notice that Python considers 12.0 and 12.000000000000000001 as equal values. This is caused by the (un-)precision how floating-points are stored in the computer. This exercise learns you to be careful to use == on floating points, since you can never be sure about the outcome when the values which are compared are almost equal. It should actually not be used on ==.

As a final exercise with comparison operators, try out some others. A list of all operators is given in the Python Documentation, Library Reference (on MS Windows it should be in your Python menu). Alternatively, Python Documentation is also available at http://www.python.org. Be sure to have the Library Reference bookmarked since you will use it more often. If you search for something, a good entry to the Library Reference is its index (linked at the top-right corner of the Library Reference page).

### 5.4.2 Logical operators

Logical operators are Boolean expressions comparing two Boolean inputs, and returning a true (represented by a integer 1) or false (represented by an integer 0). The exercises below will learn you the meaning of these operators. Use Python scripts to answer the questions when needed.

---

**Question 6:** What is the value of c in: `c = 0 and 0`?

a) 0 (False)

b) 1 (True)

Correct answer: a

Feedback: The operator `and` only returns True if both inputs are True (that is 1).

---

---

**Question 7:** What is the value of c in: `c = not 1 and 0`?

a) 0 (False)

b) 1 (True)

Correct answer: a

Feedback: None

---

---

**Question 8:** What is the value of c in: `c = not (1 and 0)`?

a) 0 (False)

b) 1 (True)

Correct answer: b

Feedback: None

---

---

**Question 9:** What is the value of c in: `c = not(not 0 or not 0)`?

a) 0 (False)

b) 1 (True)

Correct answer: a

Feedback: None

---

### 5.4.3 Conditional execution

In a previous exercise, you have made a script to calculate the slope factor in the Universal Soil Loss Equation. You have saved it as `s.py`. If you do not have it anymore, it is given in the table below. The script does not take into account that the equation used for the slope factor is not valid for

- negative values of the slope,
- values of the slope above 45 degrees.

```
import math

# program to calculate the s factor in the USLE

slope = input("Enter the slope in degrees ")
print(f"The slope you entered is {slope} degrees.")
slopeFloat = float(slope)

# slope in radians
slopeRadians = (slopeFloat / 360) * 2.0 * math.pi
print(f"This corresponds to a slope of {slopeRadians} in radians.")

# slope in m/m
slopeFraction = math.tan(slopeRadians)
print(f"This corresponds to a slope of {slopeFraction} (m/m)")

# slope factor
S = 0.065 + 0.045 * slopeFraction + 0.0065 * slopeFraction**2.0

print(f"The slope factor is {S}")
```

Open the `s.py` script and save it as `condi.py`. If you execute it, you will find that it does not take these restrictions into account. It runs with negative values or values for the slope greater than 45 degrees without complaining. But the value it returns in these cases is non-sense.

Modify `condi.py` such that it prints appropriate error messages when the user enters a slope value below zero degrees or above 45 degrees. Otherwise, it should print the slope factor. Execute the script several times to check whether it works before answering the next question.

---

**Question 10:** Which keyword did you use?

a) If

b) ifor

c) if

d) ifthen

Correct answer: c

Feedback: There are several ways to solve this problem using conditionals. The script below uses a chained conditional.

```
import math

# program to calculate the s factor in the USLE
```

(continues on next page)

(continued from previous page)

```
slope = input("Enter the slope in degrees ")
print(f"The slope you entered is {slope} degrees.")
slopeFloat = float(slope)

if slopeFloat > 45:
    print(
        "The slope you entered is greater than 45 degrees. For "
        "slope values above 45 degrees, the slope factor equation "
        "is not valid and the slope factor cannot be calculated"
    )

elif slopeFloat < 0:
    print("You entered a negative slope. Please enter positive values only")

else:
    # slope in radians
    slopeRadians = (slopeFloat / 360.0) * 2.0 * math.pi
    print(f"This corresponds to a slope of {slopeRadians} in radians.")

    # slope in m/m
    slopeFraction = math.tan(slopeRadians)
    print(f"This corresponds to a slope of {slopeFraction} (m/m)")

    # slope factor
    S = 0.065 + 0.045 * slopeFraction + 0.0065 * slopeFraction**2.0

    print(f"The slope factor is {S}")
```

## 5.5 Fruitful functions

### 5.5.1 Adding new functions

In one of the previous sections, you made the program for calculating the slope factor (s.py). It should be similar to the version for s.py given in the table below.

```
import math

# program to calculate the s factor in the USLE

slope = input("Enter the slope in degrees ")
print(f"The slope you entered is {slope} degrees.")
slopeFloat = float(slope)

# slope in radians
slopeRadians = (slopeFloat / 360) * 2.0 * math.pi
print(f"This corresponds to a slope of {slopeRadians} in radians.")

# slope in m/m
slopeFraction = math.tan(slopeRadians)
```

(continues on next page)

(continued from previous page)

```
print(f"This corresponds to a slope of {slopeFraction} (m/m)")

# slope factor
S = 0.065 + 0.045 * slopeFraction + 0.0065 * slopeFraction**2.0

print(f"The slope factor is {S}")
```

In many cases it is convenient to define functions for often used calculations in a program. For instance, the statement

```
slopeRadians = (slopeFloat / 360.0) * 2.0 * math.pi
```

Can be replaced by,

```
slopeRadians = degreesToRadians(slopeFloat)
```

while providing a definition of the function `degreesToRadians`:

```
def degreesToRadians(angleInDegrees):
     angleInRadians = (angleInDegrees / 360.0) * 2.0 * math.pi
     return angleInRadians
```

The word `def` indicates a function definition. Note the `:` at the end of the first line of the function definition. Here, the function has an 'input', which is called the argument of the function. It is `angleInDegrees`. Everything below the first line of the function definition is called the body of the function, indented from the left margin by one or two spaces. The statements in the body define what the function does. The last statement, a return statement, defines the return value of the function. In our example, the return value of the function will be assigned to `slopeRadians`. The function body in this example contains only two statements, but you may use more (not too many as this will make the code hard to read). For instance, an alternative way of defining the function would be:

```
def degreesToRadians(angleInDegrees):
    angleInDegreesDivided = angleInDegrees / 360.0
    angleInRadians = angleInDegreesDivided * 2.0 * math.pi
    return angleInRadians
```

Now it will be clear why we use a return statement: `angleInRadians` needs to be returned. Not `angleInDegreesDivided`!

Open `s.py` (either your own version or the one in the table below) and save it as `s_function.py`. Make the changes suggested above. Put the function definition at the top of the script, one line below `import math`. Save the script and execute it to check it. It should generate the same result as the original one!

Now, let's do a test: save `s_function.py` as `test.py` and move the function definition to the bottom of the script. Save `test.py` and execute it. Why do you think it doesn't work?

The next step is to make a function calculating the slope factor. Open `s_function.py` and save it as `s_twofu.py`. Modify the script by defining a second function (define it below the `degreesToRadians` function definition) named `sFactor` having one argument, the slope in degrees given as a float, with the slope factor as the return value. You could use this as the first line of the function definition:

```
def sFactor(slopeInDegrees):
```

Note that most statements which were in the `s_function.py` script need to be moved to the body of the `sFactor` function. Calculate $S$ now by calling (at the bottom of the script) the function `sFactor`. Be sure the script runs before answering the next question.

**Question 11:** Where should the `sFactor` function definition be provided?

a) Certainly below the definition of the `degreesToRadians` function.

b) Certainly above the definition of the `degreesToRadians` function.

c) Certainly inside the definition of the `degreesToRadians` function.

d) Either above or below the definition of the `degreesToRadians` function.

Correct answer: d

Feedback: As long as all the function definitions needed to execute a line are above that line, it makes no difference in what order they are given. In general, function definitions are not given inside other function definitions.

```
import math

# program to enter the slope in degrees


def degreesToRadians(angleInDegrees):
    angleInRadians = (angleInDegrees / 360.0) * 2.0 * math.pi
    return angleInRadians


def sFactor(slopeInDegrees):
    # slope in radians
    slopeRadians = degreesToRadians(slopeInDegrees)
    print(f"This corresponds to a slope of {slopeRadians} in radians.")

    # slope in m/m
    slopeFraction = math.tan(slopeRadians)
    print(f"This corresponds to a slope of {slopeFraction} (m/m)")

    # slope factor
    S = 0.065 + 0.045 * slopeFraction + 0.0065 * slopeFraction**2.0

    return S


slope = input("Enter the slope in degrees ")
print(f"The slope you entered is {slope} degrees.")

# slope factor
S = sFactor(float(slope))

print(f"The slope factor is {S}")
```

---

Imagine field data are collected of the topographical slope to estimate soil erosion for individual arable fields. On each arable field, slope is measured two times, at randomly chosen locations on the arable field. It is assumed that the average slope factor of an individual field ($S_f$) can be estimated by:

$S_f = (S_1 + S_2) / 2.0$

with $S_1$, $S_2$, the slope factors calculated from the first and second slope measurements on the field, respectively. Open `s_twofu.py` and save it as `s_two_in.py`. Using the equation given above, modify the script such that it can be used to calculate the average slope factor (Sf) of a field:

- asks the user to enter the first slope of the field
- asks the user to enter the second slope of the field
- prints Sf

You will find that you can use the same function sFactor twice. Test the script before answering the next question.

---

**Question 12:** On a field, the slope is measured two times, resulting in a slope of 11.0 and 13.0 degrees. What is the average slope factor?

a) 0.013

b) 0.091

c) 0.075

d) 0.021

Correct answer: c

Feedback: None

```
import math

# program to enter two slopes in degrees and calculate average slope
# factor


def degreesToRadians(angleInDegrees):
    angleInRadians = (angleInDegrees / 360.0) * 2.0 * math.pi
    return angleInRadians


def sFactor(slopeInDegrees):
    # slope in radians
    slopeRadians = degreesToRadians(slopeInDegrees)

    # slope in m/m
    slopeFraction = math.tan(slopeRadians)

    # slope factor
    S = 0.065 + 0.045 * slopeFraction + 0.0065 * slopeFraction**2.0

    return S


slopeOne = input("Enter the first slope in degrees ")
print(f"The first slope you entered is {slopeOne} degrees.")

slopeTwo = input("Enter the second slope in degrees ")
print(f"The second slope you entered is {slopeTwo} degrees.")
```

(continues on next page)

(continued from previous page)

```
# average slope factor of the field
Sf = (sFactor(float(slopeOne)) + sFactor(float(slopeTwo))) / 2.0

print(f"The average slope factor of the field is {Sf}")
```

## 5.6 Iteration

### 5.6.1 The while statement, creating a table (part 1)

In a previous section, you created a program (`manning.py`) calculating the flow velocity of surface water as a function of the slope, the hydraulic radius, and the Manning roughness coefficient. If you have deleted it, it is given in the table below.

```
# hydraulic radius (m)
hydraulic_radius = 3.0
# slope (-)
slope = 0.1
# manning's n
n = 0.01
# flow speed
v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
# print flow speed
print(v)
# print flow speed with sentence
print(f"With a hydraulic radius of {hydraulic_radius} (m) the flow speed is {v}
↪ m/s.")
```

The program returns the flow velocity for a single set of inputs only. Here we will go a step further. Assume a table is needed that prints the flow velocity for different values of the slope, as shown in the table below (note that the velocity values are not the correct values).

```
hydraulic radius is: 2.9 (m)
mannings n is: 0.01 (-)

slope (-) velocity (m/s)
0.01      17.435990575118256
0.02      17.259433675623676
0.03      18.12296889834665
0.04      21.17198115023651
0.05      32.87265731575946
0.06      45.91280032308756
0.07      49.803973725916286
```

To create a program that generates the table above, it is wise first to rewrite the program `manning.py` by wrapping the Manning's equation in a function with three arguments: the slope, the hydraulic radius, and the Manning's n. The function should return the flow velocity. To do this, open `manning.py` (or copy-paste it from the table above) and save it as `man_func.py`. Wrap the code in a function. When running `man_func.py`, it should create the same output as `manning.py`! Test this before answering the question below.

**Question 13:** If the function is defined as `def manning(slopeIn, hydraulic_radiusIn,nIn)` the names of the variables at the top of the script (e.g., `slope`, in `slope = 3.0`) also need to be changed to these same names (e.g., `slopeIn`), and when calling the function, the same names have to be used (e.g. `slopeIn`).

a) Yes

b) No

Correct answer: b

Feedback: No, the names used in the function can be different from the names of the variables that are passed to the function. Both versions of the script below will work.

```
slope = 3.0
hydraulic_radius = 2.9
n = 0.01


def manning(slope, hydraulic_radius, n):
    # flow speed
    v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
    return v


velocity = manning(slope, hydraulic_radius, n)

# print flow speed
print(
    f"With a hydraulic radius of {hydraulic_radius} (m) the flow speed is {velocity} m/s.
↪"
)
```

```
slope = 3.0
hydraulic_radius = 2.9
n = 0.01


def manning(slopeIn, hydraulic_radiusIn, nIn):
    # flow speed
    v = (((hydraulic_radiusIn) ** (2.0 / 3.0)) * (slopeIn ** (1.0 / 2.0))) / nIn
    return v


velocity = manning(slope, hydraulic_radius, n)

# print flow speed
print(
    f"With a hydraulic radius of {hydraulic_radius} (m) the flow speed is {velocity} m/s.
↪"
)
```

---

The next step is to modify the program such that it prints just the first column. Open `man_func.py` and save it as `man_tab1.py`. Keep all lines in the program, except the two lines that calculate and print the flow velocity. Remove

these, or de-activate them by changing them into comments:

```
# velocity = manning(slope, hydraulic_radius, n)
#
# # print flow speed
# print(f"With a hydraulic radius of {hydraulic_radius} (m) the flow speed is {velocity}␣
↪m/s.")
```

This allows you to copy-paste this part of the program later, in the case you need it again. Now, add a `while` statement such that the program prints the table below (i.e. the header and just the first column of what we need as output). You will also need some print statements that print the header (i.e. the lines above the actual table). To print a white line, use the character `\n` in a string.

```
hydraulic radius is: 2.9 (m)
mannings n is: 0.01 (-)
slope (-)
0.01
0.02
0.03
0.04
0.05
0.06
0.07
```

---

**Question 14:** Can a variable that is used inside a `while` statement be used below the `while` statement, that is outside the statement?

a) Yes

b) No

c) Depends on the name of the variable

Correct answer: a

Feedback: Yes, unlike variables inside functions, which are local, that is, they can only be used inside the function, variables in a while statement are global, and they can be accessed also outside the while loop. Try it out yourself by adding a line (second script below).

```
hydraulic_radius = 2.9
n = 0.01


def manning(slope, hydraulic_radius, n):
    # flow speed
    v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
    return v


print(f"hydraulic radius is: {hydraulic_radius} (m)")
print(f"mannings n is: {n} (-)\n")

print("slope (-)")
```

(continues on next page)

(continued from previous page)

```
x = 0.01

while x < 0.08:
    print(x)
    x = x + 0.01
```

```
hydraulic_radius = 2.9
n = 0.01


def manning(slope, hydraulic_radius, n):
    # flow speed
    v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
    return v


print(f"hydraulic radius is: {hydraulic_radius} (m)")
print(f"mannings n is: {n} (-)\n")

print("slope (-)")

x = 0.01

while x < 0.08:
    print(x)
    x = x + 0.01

print(f"x can be printed below the loop: {x}")
```

---

Finally, extend the program to calculate the flow speed for each slope value (using the manning function created at the start of this exercise), and prints it as a second column (as shown in the table at the start of this section). You can get a nicely formatted table using a tab or multiple tabs between the column. A tab is printed with the string character '\t'. Save it as `man_tab2.py`.

---

**Question 15:** What is the velocity (m/s) at a slope of 0.03 and 0.05, respectively?

a) 24.62, 25.23

b) 22.23, 22.34

c) 35.22, 45.47

d) 20.33, 35.22

Correct answer: c

Feedback: None

```
hydraulic_radius = 2.9
n = 0.01
```

(continues on next page)

(continued from previous page)

```
def manning(slope, hydraulic_radius, n):
    # flow speed
    v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
    return v


print(f"hydraulic radius is: {hydraulic_radius} (m)")
print(f"mannings n is: {n} (-)\n")
```

### 5.6.2 The while statement, creating a table (part 2)

In the previous section, you made a table of flow velocity values for different slope values. Actually, the flow velocity is a function of two variables, slope and hydraulic radius. So, a table like the one below is needed, giving the velocity for different combinations of slope and hydraulic radius. In this exercise, you will modify the program such that it generates the table below. Again the values for velocity are not correct for the settings required.

```
mannings n is: 0.01 (-)

slope (-) hydr. radius (m)  velocity (m/s)
0.11      3.0               68.98857573551179
0.11      4.0               83.57370775943878
0.11      5.0               96.97869717995717
0.12      3.0               72.05621731056017
0.12      4.0               87.28989087773772
0.12      5.0               101.29094569634633
0.13      3.0               74.99848881276804
0.13      4.0               90.85419896864778
0.13      5.0               105.42695885492729
0.14      3.0               77.92961001631138
0.14      4.0               94.28385806182264
0.14      5.0               109.40672569242204
0.15      3.0               80.5613000539548
0.15      4.0               97.59306487558015
0.15      5.0               113.24672004113509
0.16      3.0               83.20335292207616
0.16      4.0               100.79368399158986
0.16      5.0               116.96070952851464
```

Create the program for generating the table in two steps. Open `man_tab2.py` and save it as `man_tab3.py`. Modify `man_tab3.py` such that it generates the table below. If you have problems solving this, first answer the first question below the table.

```
mannings n is: 0.01 (-)

slope (-) hydr. radius (m)
0.11      3.0
0.11      4.0
0.11      5.0
0.12      3.0
0.12      4.0
```

(continues on next page)

(continued from previous page)

```
0.12      5.0
0.13      3.0
0.13      4.0
0.13      5.0
0.14      3.0
0.14      4.0
0.14      5.0
0.15      3.0
0.15      4.0
0.15      5.0
0.16      3.0
0.16      4.0
0.16      5.0
```

---

**Question 16:** How do you think you can generate the table given above?

a) Using two while loops below each other.

b) Using two while loops, the second one nested in the body of the first one.

c) Using a while loop, with an if statement inside the while loop.

Correct answer: b

Feedback: None

```
n = 0.01


def manning(slope, hydraulic_radius, n):
    # flow speed
    v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
    return v


print(f"mannings n is: {n} (-)\n")

print("slope (-)\thydr. radius (m)")
x = 0.1
while x < 0.16:
    x = x + 0.01
    y = 2.0
    while y < 4.1:
        y = y + 1
        print(f"{x}\t\t{y}")
```

---

Now, save `man_tab3.py` as `man_tab4.py` and change it resulting in an output including flow velocities that corresponds to the table given at the top of this section.

---

**Question 17:** What is the velocity (m/s) at a slope of 0.14 and a hydraulic radius of 3.0 m?

a) 89.12

b) 46.83

c) 77.92

d) 77.82

Correct answer: d

Feedback: None

```
n = 0.01


def manning(slope, hydraulic_radius, n):
    # flow speed
    v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
    return v


print(f"mannings n is: {n} (-)\n")

print("slope (-)\thydr. radius (m)\tvelocity (m/s)")
x = 0.1
while x < 0.16:
    x = x + 0.01
    y = 2.0
    while y < 4.1:
        y = y + 1
        velocity = manning(x, y, n)
        print(f"{x}\t\t{y}\t\t\t{velocity}")
```

---

## 5.7 Strings

### 5.7.1 Length of strings, string slices

Make a program `strlast.py` that asks the user to enter her family name, and prints the last letter of the family name.

---

**Question 18:** To select letters from a string, one has to use

a) Round brackets (`()`)

b) Curley brackets (`{}`)

c) Angle brackets (<>)

d) Square brackets (`[]`)

Correct answer: d

Feedback: A correct program is below. Note that an alternative for getting the last item in a list is `string[-1]`.

```
# program that prints the last letter of your family name

string = input("Enter your family name: ")

length = len(string)

lastLetter = string[length - 1]

print(f"The last letter of your family name is a {lastLetter}")
```

---

Write a program `strrange.py` that asks the user to enter her family name, and prints the 3rd, 4th and 5th letter of the name.

---

**Question 19:** What index (on the string containing the family name) could be used for this?

a) 2:5

b) 3:5

c) 4:6

d) -2:-5

Correct answer: a

Feedback: None

```
# program that prints the 3rd up to (and including) the 5th letter of your first name

string = input("Enter your family name: ")

print(f"The 3rd, 4th and 5th letter of your family name are {string[2:5]}")
```

---

### 5.7.2 String traversal (part 1)

In this exercise, you will create a program that prints your family name backwards. As a hint, first make a program with a *while* statement (save it as `strback1.py`) that prints the following:

```
10 9 8 7 6 5 4 3 2 1 0
```

---

**Question 20:** By default, a print statement prints a newline after printing the value. How can this be changed to a whitespace (instead of a newline)? By adding to the print statement:

a) `end='\t'`, for instance `print(i, end='\t')`

b) `end=' '`, for instance `print(i, end=' ')`

c) `end=`, for instance `print(i, end=)`

Correct answer: b

Feedback: None

---

With the knowledge gained from the exercise above, write a program that asks for your family name, and prints it backwards. Save it as `strback2.py`.

---

**Question 21:** You will need the number of letters in the family name. How can this be retrieved?

a) Using `len(..)`, which is in the string module.

b) Using `length(..)`.

c) Using another loop.

d) Using `len(..)`.

Correct answer: d

Feedback: Below one of the possible scripts is given.

```
# program that prints your name backwards

string = input("Enter your family name: ")

stringBack = ""

i = len(string) - 1
while i >= 0:
    letter = string[i]
    stringBack = stringBack + letter
    i = i - 1

print(f"Your name backwards is {stringBack}")
print(stringBack)
```

---

### 5.7.3 String traversal (part 2), string methods

Run this program:

```
# program that prints the result of a division
a = 2.0/8.0
print(f"The division directly printed: {a}")
```

It shows that decimals in floating points are printed using a dot (.). Although this is actually standard in English, some programs read and write decimals using a comma, i.e. 0,25 instead of 0.25.

As an exercise, modify the program given above, such that it prints the result of a division (i.e., the variable a) using a decimal comma. Convert a to a string, and use a string traversal encoded in a while statement (like you did in the previous exercise) to replace the dot with a comma. Save the program as `strreplace1.py.`.

---

**Question 22:** Almost certainly you need an `if` statement inside the `while` loop. How did you use it?

a) To stop the loop at the end of the string.

b) To select letters that are equal to a `.`.

c) To rerun the loop in case a `.` is not found.

Correct answer: b

Feedback: None

```
# program that prints the result of a division
# using a comma instead of a .
a = 2.0 / 8.0
print(f"The division directly printed: {a}")

aString = str(a)

outString = ""
i = 0
while i < len(aString):
    letter = aString[i]
    if letter == ".":
        outLetter = ","
    else:
        outLetter = letter
    outString = outString + outLetter
    i = i + 1

print(outString)
```

---

The previous exercise showed that it is quite a lot of work to manipulate strings. For this reason, Python comes with built-in methods on strings. These methods are called using dot notation, e.g. `str.find('apple','a')`. Have a look at the description of the built-in `str` methods in the Python documentation. Now, rewrite `strreplace1.py` using the appropriate function from `str`. Save the program as `strreplace2.py`.

---

**Question 23:** Which function did you use?

a) `str.find(..)`

b) `str.replace(..)`

c) `str.partition(..)`

d) `str.split(..)`

Correct answer: b

Feedback: None

```
# program that prints the result of a division
# using a comma instead of a .
a = 2.0 / 8.0
print(f"The division directly printed: {a}")

aString = str(a)
```

(continues on next page)

(continued from previous page)

```
outString = str.replace(aString, ".", ",")

print(outString)
```

## 5.8 Lists

### 5.8.1 Accessing elements, lists and for loops

The built-in function `range` can be used to create a long list of integers as in the following program:

```
a = list(range(1, 500))
print(a)
```

Create a program (save it as `li_intro.py`) that:

- creates a list (using the `range(1, 500)` statement),
- uses a while loop traversing the list, printing each element, and its square root,
- uses the same while loop to calculate the sum of the square roots of the elements, and prints the sum.

It is recommended to use a shorter list first (when developing the program), e.g. `range(1, 5)`. But note that the question below refers to the list created with `range(1,500)`.

**Question 24:** What is the sum of the square roots of the elements?

a) 9201

b) 8723

c) 6310

d) 7442

Correct answer: d

Feedback: None

```
import math

a = list(range(1, 500))
print(a)

sum = 0.0
i = 0
while i < len(a):
    v = a[i]
    vSqrt = math.sqrt(v)
    print(v, vSqrt)
    sum = sum + vSqrt
    i = i + 1
print(f"The sum of the square roots of the values is {sum}")
```

### 5.8.2 List deletion

Create a program (name it `li_delete1.py`) that creates a list with 5 elements of type floating-point. Now, extend the program such that all elements smaller than a threshold value are removed from the list.

This can be solved by traversing the list using a while statement, but this may cause problems. As you will know, deleting an element in a list can be done using `del`. This results in removal of the element and consequently, the list becomes one element shorter. This will cause a problem: it is not possible to go through a list using a `for` loop and delete elements from the same list simultaneously because in that way the `for` loop will not reach all elements. A similar kind of problem happens when you try to add lists elements while traversing the same list. The solution is to create an empty list, and copy the elements that you want to keep to that clone list. Approach the problem in a stepwise manner as follows:

1. use a `while` loop to print only the list elements smaller than or equal to the threshold value, test the program,
2. above the `while` loop, create an empty list (e.g. `newList`),
3. in the `while` loop, append the list elements smaller than or equal to the threshold value to this empty list. For each element, append it to list. This can be done with `newList.append(a)`, where `a` is the value that needs to be appended.

---

**Question 25:** How can you create an emtpy list?

a) Using `[]`

b) Using `a.emptyList()`

c) Using `range(0)`

Correct answer: a

Feedback: None

```
a = [100.9, 34.0, 34.2, 7.8, 9.0]
print(a)

# empty list
aNew = []

i = 0
while i < len(a):
    if a[i] >= 10:
        aNew.append(a[i])
    i = i + 1

print(aNew)
```

---

### 5.8.3 Matrices and gridded maps

Write a program `matrix.py` that prints the nested list:

```
matrix = [[1,2,3],[4,5,6],[7,8,9]]
```

formatted like this:

```
1 2 3
4 5 6
7 8 9
```

---

**Question 26:** An element from this nested list `matrix` can be accessed through:

a) `matrix[1,2]`

b) `matrix{1}{2}`

c) `matrix[1][2]`

Correct answer: c

Feedback: None

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

i = 0
while i < len(matrix):
    j = 0
    while j < 3:
        print(matrix[i][j], end=" ")
        j = j + 1
    i = i + 1
    print()
```

---

### 5.8.4 Strings and lists

Write a program that asks the user to enter two numbers separated by a comma, printing the product (the number resulting from multiplication) of the two numbers. Save it as `multi.py`.

---

**Question 27:** You can use a string method (from `str`). Which one?

a) `split`

b) `replace`

c) `lower`

d) `count`

Correct answer: a

Feedback: None

```
numbers = input("Enter two numbers separated by a comma ")

# split the string using ',' as a separator, resulting in a list
numbersList = str.split(numbers, ",")

firstNumber = float(numbersList[0])
```

(continues on next page)

(continued from previous page)

```
secondNumber = float(numbersList[1])
product = firstNumber * secondNumber
print(f"The product of {firstNumber} and {secondNumber} is {product}")
```

---

## 5.9 Files

### 5.9.1 Writing variables to a text file

Create a program (use `f_write1.py` for the filename) that writes:

```
first row
second row
```

to a new file.

---

**Question 28:** What would be a good last statement in your program?

a) `outFile.close`

b) `outFile.write("second row")`

c) `outFile.close()`

d) `write(outFile, "second row")`

Correct answer: c

Feedback: None

```
outFile = open("test.txt", "w")
outFile.write("first row\n")
outFile.write("second row")
outFile.close()
```

---

In a previous section, you made the program `man_tab4.py`. If you have lost it, it is below. Run it again to see what it did.

```
n = 0.01


def manning(slope, hydraulic_radius, n):
    # flow speed
    v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
    return v


print(f"mannings n is: {n} (-)\n")

print("slope (-)\thydr. radius (m)\tvelocity (m/s)")
x = 0.1
```

(continues on next page)

(continued from previous page)

```
while x < 0.16:
    x = x + 0.01
    y = 2.0
    while y < 4.1:
        y = y + 1
        velocity = manning(x, y, n)
        print(f"{x}\t\t{y}\t\t\t{velocity}")
```

As you see, it prints a lot to screen. Now, let's try to save this in a file instead. Let's start with just the first line. Save `man_tab4.py` as `f_write2.py`. Add statements closing and opening a file, and modify the line:

```
print(f"mannings n is: {n} (-)\n")
```

in such a way that it writes to the file instead of printing to the screen. If you don't manage, answering the question below might help you.

---

**Question 29:** You can only write a variable of type string to a file. How could you convert a variable to a string?

a) `string(n)`

b) `str(n)`

c) `n.string()`

Correct answer: b

Feedback: You can convert a floating-point to a string using str, in this case str(n). Alternatively, you could use string formatting, but we won't do that here.

```
# program that saves the flow velocity for different slope values
# in a file

n = 0.01


def manning(slope, hydraulic_radius, n):
    # flow speed
    v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
    return v


outFile = open("test.txt", "w")
outFile.write(f"mannings n is: {n} (-)\n")

print("slope (-)\thydr. radius (m)\tvelocity (m/s)")
x = 0.1
while x < 0.16:
    x = x + 0.01
    y = 2.0
    while y < 4.1:
        y = y + 1
        velocity = manning(x, y, n)
        print(x, "\t\t", y, "\t\t\t", velocity)
```

(continues on next page)

(continued from previous page)

```
outFile.close()
```

---

Open `f_write2.py` and save it as `f_write3.py`. Modify `f_write3.py` until it writes all output of `man_tab4.py` to a file, formatted in the same way!

---

**Question 30:** Where did you type the statement(s) that write each line of the table to the file?

a) Inside the main loop (below the first `while` statement).

b) Inside the second loop nested inside the main loop (below the second `while` statement).

c) Below the two loops (outside the loops).

Correct answer: b

Feedback: None

```
# program that saves the flow velocity for different slope values
# in a file

n = 0.01


def manning(slope, hydraulic_radius, n):
    # flow speed
    v = (((hydraulic_radius) ** (2.0 / 3.0)) * (slope ** (1.0 / 2.0))) / n
    return v


outFile = open("test.txt", "w")
outFile.write(f"mannings n is: {n} (-)\n")
outFile.write("slope (-)\thydr. radius (m)\tvelocity (m/s)\n")

x = 0.1
while x < 0.16:
    x = x + 0.01
    y = 2.0
    while y < 4.1:
        y = y + 1
        velocity = manning(x, y, n)
        outFile.write(f"{x}\t\t{y}\t\t\t{velocity}\n")

outFile.close()
```

---

### 5.9.2 Reading from a text file

Reading data from a file can be done in many different ways. Here you will use the read and readlines methods. The ASCII file `data.col` in the table below contains observations of the pH in the soil. The first and second columns contain the x and y coordinates of the observations, respectively. The third column contains the pH values. A filename

with suffix `.col` was used here, since the data are formatted in columns, but any other name would be possible. This is quite a small file, that could be edited manually, but in many cases, data files are much larger, and a program is needed for modifying the contents or the format of the file. Here, you will write such a program.

```
12.3 134.2 -9
32.5 124.5 4.5
25.6 145.2 8.9
90.3 131.2 7.3
19.0 130.4   6.4
0.0 090.2 5.4
12 080.1 8.1
14.5  190.4 6.9
3.5 137.9 6.4
4.9 112.4 8.0
13.5 123.2 7.5
24.5 112.5 7.1
343.2 234.1 7.4
0.1 142.1 7.3
11.3 134.1 3.4
31.5 114.5 4.5
15.6 145.1 -9
90.3 131.1 7.3
19.0 130.4 -9
0.0  090.1 5.4
11 080.1 8.1
14.5 190.4 6.9
3.5 137.9 6.4
4.9 111.4 8.0
13.5 113.1 7.5
24.5  111.5 7.1
343.1 134.1 7.4
0.1 141.1 -9
```

Normally, you would have the file on your harddisk. Here, you need to create it first. Copy-paste it to your editor (also used for writing programs) and save it as `data.col`. Now, write a program that reads the contents of the file and prints it to screen. Use the `read` method. Save it as `f_read1.py`.

---

**Question 31:** What is the data type of the variable that is returned by the `read` method?

a) Table

b) Floating point

c) Integer

d) String

Correct answer: d

Feedback: The `read` method assigns all the contents of the file to a variable of type string. You can check this by using the `type` function as in the script below.

```
inFile = open("data.col")

# read the input file
```

(continues on next page)

(continued from previous page)

```
contentsInFile = inFile.read()
print(contentsInFile)

print(type(contentsInFile))

# close the file
inFile.close()
```

---

Now, let's look at the `readlines` method. Create the program `f_read2.py` given in the table below. Execute it.

```
inFile = open("data.col", "r")

a = inFile.readlines()
print(a)

inFile.close()
```

---

**Question 32:** What is the type of `a` and `a[0]`, respectively?

a) A list and a string.

b) A string and a list.

c) Both lists.

d) Both strings

Correct answer: a

Feedback: The type of `a` is a list as `readlines` returns a list where each line in the file is stored as a list item. The statement `a[0]` returns the contents of the first list item and this is a string, the last character `\n` is the newline character.

```
inFile = open("data.col", "r")

a = inFile.readlines()
print(a)

inFile.close()

print(type(a))
print(type(a[0]))
```

---

Create a program `f_read3.py` using the `readlines` method that prints separately each line of the `file data.col` to screen. You need to loop over the 'lines'.

---

**Question** What statement can be used here to create the loop?

a) `while` is possible, but `for` is not

b) both `while` and `for` is possible

c) `if` is the only statement that can be used here

d) `for` is possible, but `while` is not

*Correct answers:* b. or d.

*Feedback:* The 'classic' way of doing this is using a while loop. An alternative way is using a for loop. This is considered better, since it results in a shorter program that does the same.

```
inFile = open("data.col", "r")

a = inFile.readlines()

i = 0
while i < len(a):
    print(a[i], end="")
    i = i + 1

inFile.close()
```

```
inFile = open("data.col", "r")

a = inFile.readlines()
for aLine in a:
    print(aLine, end="")

inFile.close()
```

**Question 33:** What statement can be used here to create the loop?

a) `while` is possible, but `for` is not

b) both `while` and `for` is possible

c) `if` is the only statement that can be used here

d) `for` is possible, but `while` is not

Correct answer: b. or d.

Feedback: The 'classic' way of doing this is using a while loop. An alternative way is using a for loop. This is considered better, since it results in a shorter program that does the same.

```
inFile = open("data.col", "r")

a = inFile.readlines()

i = 0
while i < len(a):
    print(a[i], end="")
    i = i + 1

inFile.close()
```

```
inFile = open("data.col", "r")
```

(continues on next page)

(continued from previous page)

```
a = inFile.readlines()
for aLine in a:
    print(aLine, end="")

inFile.close()
```

---

Now, assume another existing program that you just bought needs to read the contents of `data.col`, but:

- the other program cannot read missing values (indicated by `-9` in `data.col`)
- the other program can only import the file when columns are separated by a comma (, `) instead of whitespace characters (as in `data.col`).

Open `f_read3.py` and save it as `f_read4.py`. Modify it, such that it does not print the lines containing a missing value for the pH.

---

**Question 34:** Which method can be used here?

a) `str.split`

b) `str.mv`

c) `str.capitalize`

d) `str.find`

Correct answer: a

Feedback: The script is given below. Note that in the for loop, `aLine` is a string. This string is separated in three elements (the 1st, 2nd and 3rd column of the file) using `string.split`. This method is very convenient, since it removes all white space inbetween the values on a line, and moreover, newline characters. It results in a list with list items that contain only the value itself and nothing more. You can check this by printing aLineList. The if statement checks whether the pH element is equal to -9. Note that the quotes around -9 are really needed here. The statement `if not(pH == -9):` doesn't work, since pH is of type string, so it needs to be compared to a variable of type string too, i.e. "-9". If you type just -9 without quotes, it doesn't work, since that would represent an integer.

```
inFile = open("data.col", "r")

a = inFile.readlines()
for aLine in a:
    # each line splitted in three separate strings stored as list items
    aLineList = str.split(aLine)
    # the third list item, i.e. pH
    pH = aLineList[2]
    # print only if pH is not a missing value
    if not (pH == "-9"):
        print(aLine, end="")

inFile.close()
```

---

Next, we need to replace the whitespace inbetween the columns with comma's, such that the output looks like this:

```
32.5,124.5,4.5
25.6,145.2,8.9
90.3,131.2,7.3
19.0,130.4,6.4
0.0,090.2,5.4
12,080.1,8.1
14.5,190.4,6.9
3.5,137.9,6.4
4.9,112.4,8.0
13.5,123.2,7.5
24.5,112.5,7.1
343.2,234.1,7.4
0.1,142.1,7.3
11.3,134.1,3.4
31.5,114.5,4.5
90.3,131.1,7.3
0.0,090.1,5.4
11,080.1,8.1
14.5,190.4,6.9
3.5,137.9,6.4
4.9,111.4,8.0
13.5,113.1,7.5
24.5,111.5,7.1
343.1,134.1,7.4
```

Open `f_read4.py` and save as `f_read5.py`. Modify it, printing comma's instead of whitespace between columns. You only need to change the print statement! As follows:

```
inFile = open("data.col", "r")

a = inFile.readlines()
for aLine in a:
    # each line splitted in three separate strings stored as list items
    aLineList = str.split(aLine)
    # the third list item, i.e. pH
    pH = aLineList[2]
    # print only if pH is not a missing value
    if not (pH == "-9"):
        print(aLineList[0] + "," + aLineList[1] + "," + aLineList[2])

inFile.close()
```

CHAPTER

SIX

# CALIBRATION

Download this website as pdf.

Download this website as epub (for e-readers).

To subscribe to our courses visit http://www.pcraster.eu

## 6.1 Introduction

### 6.1.1 Study area

Download the dataset `here`.

In this lab you will calibrate parameters of a hydrological model for the Dorfertal catchment near Kals (Austria). Use `aguila` to display the digital elevation map (`dem.map`, elevation above sea level in metres) the local drain direction map (`ldd.map`) and the location of the streamflow measurements (`sample_location.map`). The cell size used is 100 m. DEM data are from Open Data Oesterreich.

As you can see on the photo below, even in summer the peaks are covered with snow.

Timeseries data as input to the model are plotted in `observed_timeseries.pdf`. Temperature and precipitation are taken from CFSR for the time span from 1 jan 1990 up to 31 dec 1993. Streamflow is taken from GRDC.

### 6.1.2 The hydrological model

Except interception of precipitation (i.e., rainfall plus snowfall) by the vegetation, the model represents all main components of a hydrological system, where each component is modelled using simple equations. The figure below shows the model structure. The model has two storages, the snow storage and the subsurface storage. The latter represents all the water that is stored below the surface. The input of water is precipitation, the outputs from the system are streamflow and loss of water to the atmosphere. A number of fluxes is defined each defined by a particular equation. These equations use so called parameters, which are mostly assumed to be fixed. They are shown in red in the schematic. It is often hard to directly measure parameters and this is why they are 'calibrated'. We will come back to this later.

An essential component in the model is the air temperature. As it is not measured for all pixels in the area, we use a time series of temperature observed at a certain reference elevation. As temperature mostly decreases with elevation, temperature is then estimated for each pixel using the digital elevation model (surface level of each pixel). For detailed model equations refer to the text below. Most important in this exercise is the snowmelt rate parameter $m$ which is the one that will be calibrated.

Fig. 1: Dorfertal catchment with main stream, photo taken in summer.

The equations below are evaluated for each grid cell, for all timesteps (time step is 1 day):

The air temperature ($t$, degrees Celsius) above the surface is:

$$t = t_{obs} - l(e - e_{obs})$$

With, $t_{obs}$, the observed temperature (degrees Celcius); $l$, the temperature lapse rate (degrees / m elevation); $e$, the elevation of the land surface; $e_{obs}$, the reference elevation of the observed temperature.

Snowfall ($s$, m equivalent water depth/day) is

$$s = \begin{cases} p & \text{for } t < 0.0 \\ 0 & \text{for } t \geq 0.0 \end{cases}$$

With, $p$, the observed precipitation (m/day).

Rainfall ($r$, m/day) is

$$r = p - s$$

The rate of change in the snow depth ($s_d$, m equivalent water depth) is

$$\frac{ds_d}{dt} = s - mt - u$$

With, $m$, meltrate parameter (degree day factor, m/(day degree)); $u$, sublimation (m/day). The sublimation is equal to the loss of water to the atmosphere parameter ($l_{atm}$, m/day); if the snow depth $s_d$ becomes zero, sublimation becomes zero and loss of water to the atmosphere is taken from the subsurface water storage (see below).

The water flux available for infiltration ($a$) equals the sum of snowmelt and rainfall:

$$a = m(\max(t, 0)) + s$$

With, $m$, the snow melt rate parameter (m/(day degree)). The actual infiltration ($i$, m/day) is:

$$i = \min(i_{cap}, a)$$

With, $i_{cap}$ the infiltration capacity parameter (m/day).

Water in the subsurface is represented by $g$ (m), this is the amount of water stored below the surface, loosely defined as groundwater. Upward seepage from the subsurface ($g_{seep}$, m/day, loss of groundwater to the surface water) is:

$$g_{seep} = p_{gw} g$$

With, $p_{gw}$, seepage parameter (/day). The rate of change in the groundwater is:

$$\frac{dg}{dt} = i - g_{seep} - u$$

With, $u$ the loss to the atmosphere which is zero in case of snow cover (in which case loss to the atmosphere is taken from the snow cover).

The amount of surface runoff (m/day) generated is:

$$r_g = a - i + g_{seep}$$

The streamflow is calculated for each day as the sum of $r_g$ in all the upstream pixels of a cell.

Inspect the script `runoff.py`. It is a standard PCRaster Python script except for the added capability to pass a parameter value to the actual model; the `def __init__` statement takes an argument `meltRateParameter` which is passed when calling the model at the bottom of the script. At the bottom of the script is also some code that will be used for the calibration, which will be explained later on. The model simulates the same time span as given by `observed_timeseries.pdf`, i.e. three years.

Execute the model. Inspect the timeseries of maps created by the model; snow depth, `snow` (`snow0000.001`, `snow0000.002`, etc maps); subsurface storage, `sub`; and streamflow discharge, `dis`. For instance:

```
aguila dem.map sample_location.map --timesteps=[1,1461,1] snow dis
```

Do not forget to right-mouse click on the legend to plot the timeseries - if you click on the map you can then plot the timeseries for any location.

---

**Question 1:** Does the snowdepth at timestep 1 have a realistic value?

a) Yes, it is midsummer and snowdepth is thus close to zero.

b) Yes, it is midwinter but there was hardly any snowfall resulting in zero snowdepths.

c) No, it is assumed snowdepth is zero at the start as there is no information on the starting value.

d) No, it is assumed snowdepth is at a maximum at the start as there is no information on the starting value.

Correct answer: c

Feedback: The depth of the snow is unknown at the start, which is 1 january, i.e. midwinter. Of course there will be a considerable snowpack then. So simulation results for the first year at least should be considered a rough estimation. In the calibration later on we will use the first year to 'spin up' the model and from year two on data are used for the calibration.

---

---

**Question 2:** What is the mechanism determining larger amounts of streamflow in spring?

a) Snowmelt. The snowdepth is decreasing in spring causing high streamflow. In addition, peaks in streamflow occur due to direct rain.

b) Rainfall. The rainfall is seasonal with highest rain in winter and spring. In winter it falls as snow but in spring it falls as rain causing high streamflow.

c) Loss of water to the atmosphere. This changes over time, with highest losses in summer and winter, causing higher streamflow in spring (and autumn).

d) Wrong! Streamflow is highest in summer when precipitation falls as rain, which drains to the stream.

Correct answer: a

Feedback: Like most Alpine catchments the streamflow is mainly determined by the occurence of snowmelt. In spring, snowmelt is highest due to increasing temperatures. In summer and autumn most of the snow has disappeared and only rainfall contributes to streamflow (and flow from the groundwater). In winter there is hardly discharge as precipitation falls as snow, but there will be some baseflow from the groundwater.

---

### 6.1.3 Comparison with observed streamflow

Let's now compared modelled streamflow with observed (measured) streamflow. Here we do so by importing the observed streamflow (at the measurement location) to the model and reporting it as a map. This is somewhat cumbersome but has the advantage that we can evaluate results with `aguila`.

Add the following to the bottom of the `dynamic` section:

```
# read observed discharge
dis_obs = pcr.timeinputscalar("streamflow.txt", pcr.ifthen(self.sampleLocation, pcr.
↪boolean(1)))
```

And add additional statements that:

- Calculate the difference between modelled and observed streamflow (use the `-` operator, modelled minus observed).
- Write this difference map to disk (`report`), use `dMm0` as output file name.
- Write the observed discharge to disk (`dis_obs`), use `d0` as output file name.

Run the model and display `d0`, `dMm0`, and `dis`. Right-click on the legend to plot the timeseries and then click on the measurement location to retrieve the values at the this location in the timeseries.

---

**Question 3:** What is the main difference between modelled and observed streamflow?

a) The streamflow is overestimated in midwinter.

b) The streamflow is often too high in autumn.

c) The streamflow is often too low in spring and too high in summer.

d) The streamflow is often too high in spring and too low in summer.

Correct answer: c

Feedback: The streamflow is too high in summer and too low in spring. In other words, the high streamflows due to melting of the snow come too late in the season. The bottom part of the dynamic section should look like this:

```
        # read observed discharge
        dis_obs = pcr.timeinputscalar(
            "streamflow.txt", pcr.ifthen(self.sampleLocation, pcr.boolean(1))
        )
        # modelled minus observed discharge
        disMm0 = dischargeMetrePerSecond - dis_obs
        # write to disk
        self.report(disMm0, "dMm0")
        self.report(dis_obs, "d0")
```

---

## 6.2 Sensitivity Analysis

### 6.2.1 Melt rate parameter

Calibration involves the adjustment of one or more parameter values such that model outputs become similar (or almost similar) to observations (measurements) of these outputs. Here we have only one measurement: the streamflow and how it changes over time. In the following we will calibrate the melt rate parameter using the streamflow measurements.

But first we need to do a sensitivity analysis. Before calibrating parameters, it is wise to perform a sensitivity analysis. In a sensitivity analysis we explore how model outputs change when a certain parameter is changed. This is informative for calibration as it tells us whether the parameter that we calibrate anyway has effect on the model output that we use for calibration (here: streamflow).

Sensitivity analysis can be done in an exact manner by calculating for instance a sensitivity index. Here we follow a more exploratory way.

Open `runoff_sens_melt_rate.py`. It is exactly the same script as `runoff.py`, but the reports to disk have been removed. We will use this script to create a model scenario with an increased meltrate and compare its output with the default model (`runoff.py`).

In `runoff_sens_melt_rate.py` change three things:

1) Change the `meltRateParameter`, defined at the bottom part of the script to 0.004.

2) When we run this script, we want to store outputs in another folder, `melt_rate_high`, it has already been created. And we want to compare the scenario with the original run. To do so, add the following lines at the bottom of the dynamic section:

```
# sensitivity analysis
# snow cover
# read the snow cover from the default run
snow_default = self.readmap("snow")
# calculate the difference with the default run
snowDiff = self.snow - snow_default
# report in the melt_rate_low folder
self.report(snowDiff, "melt_rate_high/snowD")
```

3) Item 2) above compared the snow cover of the scenario with the original run. Do the same for streamflow.

Now, first run the default model `runoff.py`. When finished, run `runoff_sens_melt_rate.py`.

When finished, you can plot results stored in the scenario directory, combined with the difference between modelled and observed streamflow from the previous section (dMm0):

```
aguila --timesteps=[1,1461,1] melt_rate_high/disD melt_rate_high/snowD dMm0
```

---

**Question 4:** What is the effect of an increased melt rate parameter?

a) The snow accumulation in autumn is higher causing higher streamflows throughout the year.

b) The snow accumulation in autumn is less causing lower streamflows throughout the year.

c) The snow disappears more slowly in spring causing more streamflow in summer.

d) The snow disappears more quickly in spring causing more streamflow in spring.

Correct answer: d

Feedback: A higher meltrate causes more rapid melt of snow in spring resulting in more streamflow in that season. With lower meltrates, the snow melts later in the year and in addition more snow is lost due to sublimation of snow in summer. The bottom part of the dynamic section should look like this:

```
        # sensitivity analysis
        # snow cover
        # read the snow cover from the default run
        snow_default = self.readmap("snow")
        # calculate the difference with the default run
        snowDiff = self.snow - snow_default
        # report in the melt_rate_low folder
        self.report(snowDiff, "melt_rate_high/snowD")

        # discharge
        # read the discharge from the default run
        dis_default = self.readmap("dis")
        # calculate the difference with the default run
        disDiff = dischargeMetrePerSecond - dis_default
        # report in the melt_rate_low folder
        self.report(disDiff, "melt_rate_high/disD")


nrOfTimeSteps = 1461

# read the observed streamflow from disk and store in numpy array
streamFlowObservedFile = open("streamflow.txt", "r")
streamFlowObserved = numpy.zeros(nrOfTimeSteps)
streamFlowObservedFileContent = streamFlowObservedFile.readlines()
for i in range(0, nrOfTimeSteps):
    splitted = str.split(streamFlowObservedFileContent[i])
    dischargeModelled = splitted[1]
    streamFlowObserved[i] = float(dischargeModelled)
streamFlowObservedFile.close()


# run the model with a particular value of the meltrate parameter
# for brute force calibration put the code below in a loop
```

(continues on next page)

(continued from previous page)

```
# increased melt rate
# meltRateParameter = 0.003
meltRateParameter = 0.004
```

---

---

**Question 5:** In the next section we will calibrate the melt rate parameter. Do you think it should be higher than the value currently used or lower?

a) Higher, streamflow was too high in summer and this will reduce.

b) Lower, streamflow was too low in summer and this will reduce.

c) Higher, streamflow was throughout the year too high and this will reduce.

d) Lower, streamflow was throughout the year too low and this will reduce.

Correct answer: a

Feedback: The streamflow was too low in spring and too high in summer. With higher melt rates, the melt will start earlier in the year resulting in an increase in streamflow in spring. This will lead to a better fit between observed and modelled streamflow. In the next section you will determine how much exactly streamflow has to be increased.

---

## 6.3 Calibration of a single parameter

### 6.3.1 Calibration using brute force

We will now calibrate the melt rate parameter with the brute force technique. Brute force means that we run the model multiple times, in a loop, each time with a parameter value taken from a predefined list of parameter values. The parameter values that gives the best fit with observed discharge is the 'optimum' value, i.e. the calibrated value.

But what is 'best fit'? This can be defined with a goal function or objective function. Here we use the 'mean squared error':

$$SS = \frac{\sum_{i=1}^{T} \left(\hat{Q}_i - Q_i\right)^2}{T}$$

With, $SS$, the value of the goal function; $\hat{Q}_i$, modelled streamflow at timestep $i$; $Q_i$, measured streamflow at timestep $i$; $T$, the number of timesteps.

Open `runoff_calibration_one_par.py`. Again it is the same script as `runoff.py` but the reports to disk have been removed. Have a look at the lower part now. There, the observed streamflow is first read from disk and stored as an array named `streamFlowObserved`. This represents the $Q_i$ values in the equation above. The lowest part then calls the model (and passing the melt rate parameter value) which returns the modelled streamflow `streamFlowModelled`. At the bottom then the goal function above is evaluated for all timesteps. Note that it neglects the first year, which is the spin up time period required as snow cover is not yet realistic in this year.

If you run the model once, it prints the parameter value used and the objective function value.

Now modify the script such that it runs the model for a series of snow melt rate values, for instance from 0.0 up to 0.02, with a step size of 0.001. Note that you can first run it with a larger step and later on 'zoom in' to the correct value by narrowing the search range and using smaller steps. You will need to write a loop in Python, where you call the model each time with a different melt rate value. Store results in a text file and plot the response line, which is a plot

of the parameter value (on the x-axis) vs. the goal function value (on the y-axis). The lowest point on this curve is the calibrated melt rate.

---

**Question 6:** What is the calibrated value of the melt rate parameter?

a) 0.0089

b) 0.0090

c) 0.0060

d) 0.0061

Correct answer: b

Feedback: The answer script is provided below. It writes a text file that you can plot with plotting program (e.g. spreadsheet) or with for instance matplotlib in Python. The matplotlib script is also provided below.

```
import numpy
import pcraster as pcr
import pcraster.framework as pcrfw


# helper function 1 to read values from a map
def getCellValue(Map, Row, Column):
    Value, Valid = pcr.cellvalue(Map, Row, Column)
    if Valid:
        return Value
    else:
        raise RuntimeError("missing value in input of getCellValue")


# helper function 2 to read values from a map
def getCellValueAtBooleanLocation(location, map):
    # map can be any type, return value always float
    valueMap = pcr.mapmaximum(pcr.ifthen(location, pcr.scalar(map)))
    value = getCellValue(valueMap, 1, 1)
    return value


class MyFirstModel(pcrfw.DynamicModel):
    def __init__(self, meltRateParameter):
        pcrfw.DynamicModel.__init__(self)
        pcr.setclone("clone.map")

        # assign 'external' input to the model variable
        self.meltRateParameter = meltRateParameter

    def initial(self):
        self.clone = self.readmap("clone")

        dem = self.readmap("dem")

        # elevation (m) of the observed meteorology, this is taken from the
```

(continues on next page)

(continued from previous page)

```
        # reanalysis input data set
        elevationMeteoStation = 1180.0
        elevationAboveMeteoStation = dem - elevationMeteoStation
        temperatureLapseRate = 0.005
        self.temperatureCorrection = elevationAboveMeteoStation * temperatureLapseRate

        # potential loss of water to the atmosphere (m/day)
        self.atmosphericLoss = 0.002

        # infiltration capacity, m/day
        self.infiltrationCapacity = pcr.scalar(0.0018 * 24.0)

        # proportion of subsurface water that seeps out to surface water per day
        self.seepageProportion = 0.06

        # amount of water in the subsurface water (m), initial value
        self.subsurfaceWater = 0.0

        # amount of upward seepage from the subsurface water (m/day), initial value
        self.upwardSeepage = 0.0

        # snow thickness (m), initial value
        self.snow = 0.0

        # flow network
        self.ldd = self.readmap("ldd")

        # location where streamflow is measured (and reported by this model)
        self.sampleLocation = self.readmap("sample_location")

        # initialize streamflow timeseries for directly writing to disk
        self.runoffTss = pcrfw.TimeoutputTimeseries(
            "streamflow_modelled", self, self.sampleLocation, noHeader=True
        )
        # initialize streamflow timeseries as numpy array for directly writing to disk
        self.simulation = numpy.zeros(self.nrTimeSteps())

    def dynamic(self):
        precipitation = pcr.timeinputscalar("precipitation.txt", self.clone) / 1000.0
        temperatureObserved = pcr.timeinputscalar("temperature.txt", self.clone)
        temperature = temperatureObserved - self.temperatureCorrection

        freezing = temperature < 0.0
        snowFall = pcr.ifthenelse(freezing, precipitation, 0.0)
        rainFall = pcr.ifthenelse(pcr.pcrnot(freezing), precipitation, 0.0)

        self.snow = self.snow + snowFall

        potentialMelt = pcr.ifthenelse(
            pcr.pcrnot(freezing), temperature * self.meltRateParameter, 0
        )
        actualMelt = pcr.min(self.snow, potentialMelt)
```

(continues on next page)

(continued from previous page)

```
        self.snow = self.snow - actualMelt

        # sublimate first from atmospheric loss
        self.sublimation = pcr.min(self.snow, self.atmosphericLoss)
        self.snow = self.snow - self.sublimation

        # potential evapotranspiration from subsurface water (m/day)
        self.potential_evapotranspiration = pcr.max(
            self.atmosphericLoss - self.sublimation, 0.0
        )

        # actual evapotranspiration from subsurface water (m/day)
        self.evapotranspiration = pcr.min(
            self.subsurfaceWater, self.potential_evapotranspiration
        )

        # subtract actual evapotranspiration from subsurface water
        self.subsurfaceWater = pcr.max(
            self.subsurfaceWater - self.evapotranspiration, 0
        )

        # available water on surface (m/day) and infiltration
        availableWater = actualMelt + rainFall
        infiltration = pcr.min(self.infiltrationCapacity, availableWater)
        self.runoffGenerated = availableWater - infiltration

        # streamflow in m water depth per day
        discharge = pcr.accuflux(self.ldd, self.runoffGenerated + self.upwardSeepage)

        # upward seepage (m/day) from subsurface water
        self.upwardSeepage = self.seepageProportion * self.subsurfaceWater

        # update subsurface water
        self.subsurfaceWater = pcr.max(
            self.subsurfaceWater + infiltration - self.upwardSeepage, 0
        )

        # convert streamflow from m/day to m3 per second
        dischargeMetrePerSecond = (discharge * pcr.cellarea()) / (24 * 60 * 60)
        # sample the discharge to be stored as timeseries file
        self.runoffTss.sample(dischargeMetrePerSecond)

        # read streamflow at the observation location
        runoffAtOutflowPoint = getCellValueAtBooleanLocation(
            self.sampleLocation, dischargeMetrePerSecond
        )
        # insert it in place in the output numpy array
        self.simulation[self.currentTimeStep() - 1] = runoffAtOutflowPoint


nrOfTimeSteps = 1461
```

(continues on next page)

(continued from previous page)

```
# read the observed streamflow from disk and store in numpy array
streamFlowObservedFile = open("streamflow.txt", "r")
streamFlowObserved = numpy.zeros(nrOfTimeSteps)
streamFlowObservedFileContent = streamFlowObservedFile.readlines()
for i in range(0, nrOfTimeSteps):
    splitted = str.split(streamFlowObservedFileContent[i])
    dischargeModelled = splitted[1]
    streamFlowObserved[i] = float(dischargeModelled)
streamFlowObservedFile.close()


# function to create series of parameter values for which model is run
def createParValues(lower, upper, nrSteps):
    step = (upper - lower) / (nrSteps - 1)
    return numpy.arange(lower, upper + step, step)


# create melt rate parameters for which model is run
meltRateParameters = createParValues(0.0, 0.02, 21)
print("model will be run for values of melt rate: ", meltRateParameters)

f = open("calibration_one_par_results.txt", "w")

# calibrate melt rate
for meltRateParameter in meltRateParameters:
    myModel = MyFirstModel(meltRateParameter)
    dynamicModel = pcrfw.DynamicFramework(myModel, nrOfTimeSteps)
    dynamicModel.setQuiet()
    dynamicModel.run()

    # obtain modelled streamflow
    streamFlowModelled = myModel.simulation
    # calculate objective function, i.e. mean of sum of squares and ignore first year
    SS = numpy.mean((streamFlowModelled[365:] - streamFlowObserved[365:]) ** 2.0)
    print("meltrate: ", meltRateParameter, ", mean squared error: ", SS)
    f.write(str(meltRateParameter) + " " + str(SS) + "\n")

f.close()
```

```
import matplotlib.pyplot as plt

# read the calibration results file
file = open("calibration_one_par_results.txt", "r")
lines = file.readlines()

parValues = []
goalValues = []
for line in lines:
    splitted = str.split(line)
    parValues.append(float(splitted[0]))
    goalValues.append(float(splitted[1]))
```

(continues on next page)

(continued from previous page)

```
file.close()

f = plt.figure()
plt.plot(parValues, goalValues)
plt.xlabel("parameter value")
plt.ylabel("goal function value")
f.savefig("plot_calibration_one_par_results.pdf")
```

---

### 6.3.2 Effect of observations used in calibration

In the previous example you used the last three years for calibration. Let's see whether using a subset of the data for calibration has an effect of the calibrated value.

Rerun the calibration but now calibrate on the discharge of the last year alone (instead of using the last 3 years).

---

**Question 7:** What is the calibrated value of the melt rate parameter, when calibrating against discharge for the last year only?

a) 0.011

b) 0.012

c) 0.014

d) 0.018

Correct answer: c

Feedback: The melt rate calibrated for the last year is somewhat higher. There are many possible explanation for this: measurements errors, process-descriptions in the model that in some years are more representative for processes occurring in the system modelled compared to other years, gradual changes over years in drivers of the system, and maybe more. In general it is always better to use a longer time span for calibration as this results in a calibration that is representative of the average system behaviour.

```
    # calculate objective function, i.e. mean of sum of squares and ignore first year
    SS = numpy.mean((streamFlowModelled[1095:] - streamFlowObserved[1095:]) ** 2.0)
    print("meltrate: ", meltRateParameter, ", mean squared error: ", SS)
```

---

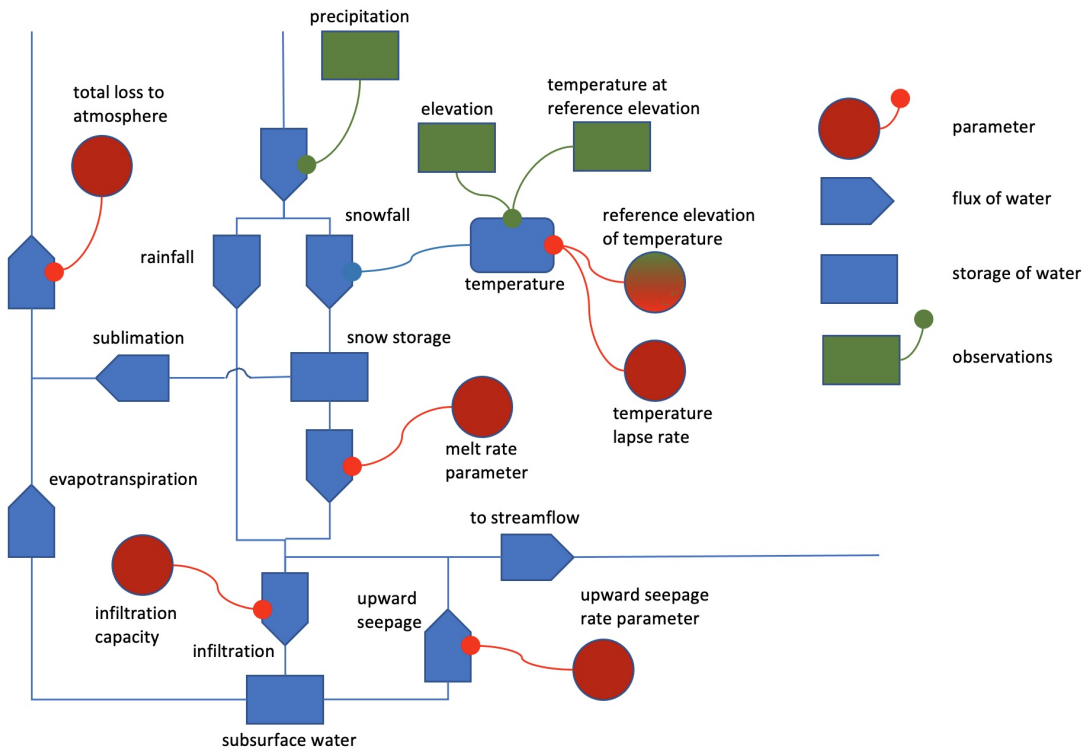


Fig. 2: Schematic of the model. Reference elevation of temperature is in principle given by the dataset, but has some uncertainty and could be adjusted in calibration, which is why it is shown as a parameter here.