Octave with Spice

Or a Gentle Introduction to GNU Octave Towards Linear Programming

Tommi Sottinen

tommi.sottinen@uwasa.fi

www.uwasa.fi/~tsottine/spicy_or/octave_with_spice.pdf

November 7, 2022

Preface

These notes are an introduction on how to use GNU Octave for the six week 5 ECTS course ORMS1020 Operations Research in the University of Vaasa. I expect that it will take two weeks to go through the material presented here in the course. The remaining four weeks will then be from the companion notes "Linear Programming with Spice".

The first three chapters of these notes are the gentle introduction to Octave and the fourth chapter is the spice. Indeed, the first three chapters should give a student with minimal experience of programming the necessary background to understand chapter 4, where we show how Linear Programming problems (LP's) can be solved by using GNU Linear Programming Kit (GLPK) in GNU Octave.

I would like to thank Matti Laaksonen and Rudi Wietsma for carefully reading the manuscript and for pointing out several mistakes.

T.S.
Vaasa November 7, 2022

Contents

1 Getting GNU Octave **3**

Installing GNU Octave 3

Getting Help 5

2 GNU Octave as Calculator **7**

Simple Calculations and Variables 7

Matrices and Vectors 10

3 Working with m-Files **16**

Script m-Files 16

Function m-Files 19

4 Solving LP's with GLPK **21**

Product-Mix Problem 21

Solving Product-Mix Problem with GLPK 24

Chapter 1

Getting GNU Octave

These instructions refer to GNU Octave version 6.2.0 released in 2021-02-20. Instructions for newer versions (e.g. 6.3.0 released in 2021-07-11) should be pretty much the same. If you have already an older GNU Octave installed, it will probably work just fine. There is most likely no reason to update your GNU Octave for the purpose of these notes and the ORMS1020 course. On the other hand, there is absolutely no reason not to update your GNU Octave to the latest stable version.

Installing GNU Octave

GNU Octave is open source free alternative to the famous mathematical programming language Matlab. The home page of GNU Octave is https://www.gnu.org/software/octave/index, where you can download it and find its documentations.

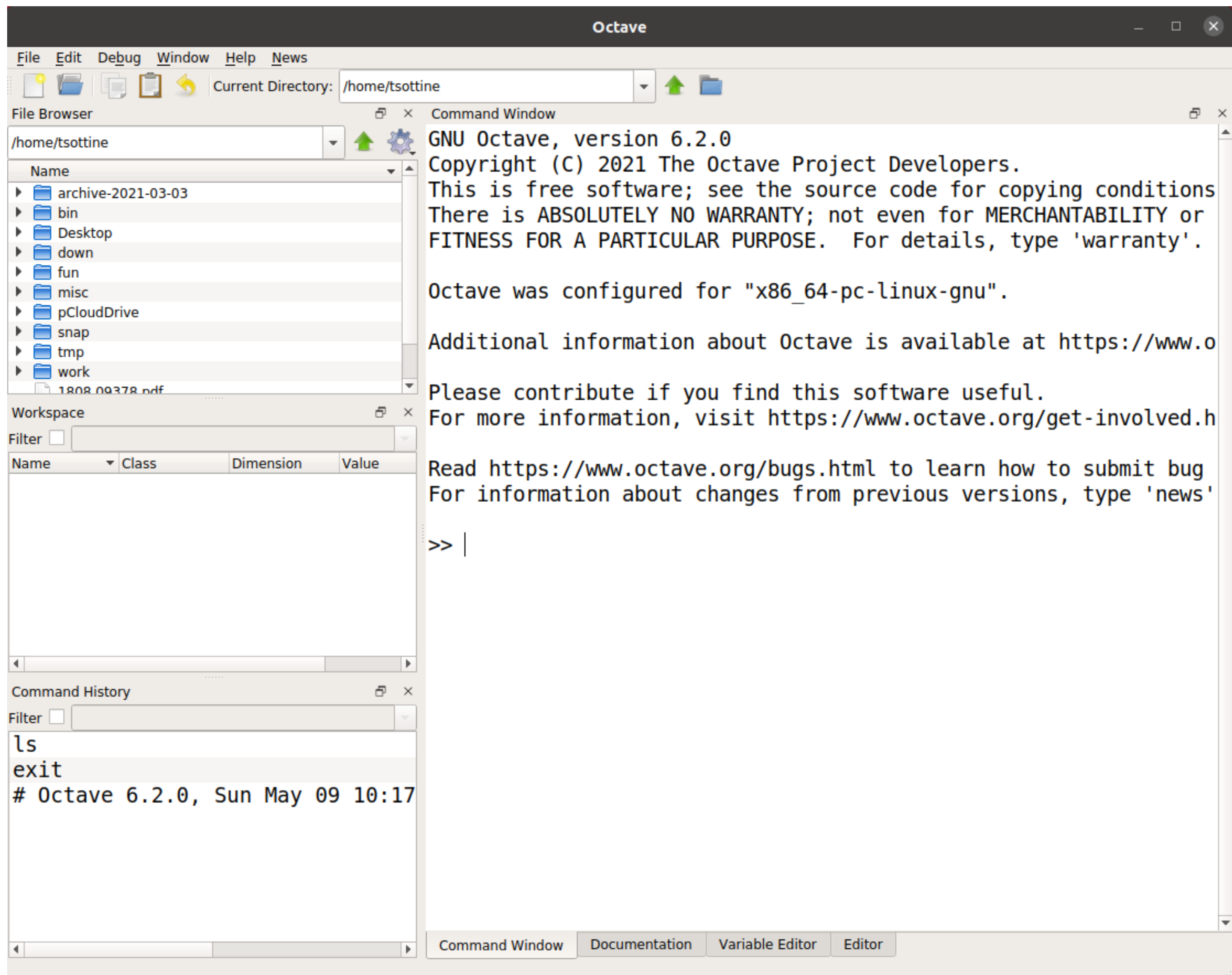
For Windows users there are installers for GNU Octave. A direct link to the installers is https://www.gnu.org/software/octave/download#ms-windows. It is recommended to use Windows-64 installer.

If you are a macOS user and familiar with Homebrew or MacPorts, you can find an installer that way. There are also instructions for macOS users in the GNU Octave wiki page https://wiki.octave.org/Octave_for_macOS.

Linux users find help on how to install GNU Octave from the GNU Octave wiki page https://wiki.octave.org/Octave_for_GNU/Linux.

Finally, since GNU Octave is open source, you can also compile it yourself. The source code can be found here: https://ftpmirror.gnu.org/octave.

After installation, depending on your system, you should have Octave GUI (Graphical User Interface) and maybe also Octave CLI (Command Line Interface) installed. I strongly recommend using the GUI version. Once you open the GUI version of GNU Octave, you should have something like the following window in your screen.



Note that depending on your system setup, you Octave GUI window may look slightly different.

The following is our first exercise. The exercises are littered all around the notes and not collected to separate sections. They also follow the general running numbering of subsections. This makes finding the exercises slightly inconvenient and the numbering may look a bit confusing. The reason for this inconvenience is that in this way you are encouraged to read the material and you will also know how the exercises are related to the material. The first Exercise 1.1 is simple. Its almost only point is to check that your installation of GNU Octave works.

1.1 Exercise (Hello, World!)

Type **help printf** and **help disp** in the GNU Octave Command Window. Then make GNU Octave say "Hello, World!" to you.

1.2 Remark (Octave Online)

If you failed to install GNU Octave to your computer, or just for some reason do not want to do it, you can use GNU Octave online in https://octave-online.net/. In Octave Online you

can use the command line (a.k.a. console) immediately, but using script files (a.k.a. m-files) requires signing in to the system. You can simply sign in by using your e-mail.

Octave Online seems convenient, but I strongly discourage using it. Besides the obvious problems with online systems, it seems that Octave Online is very slow compared to installing GNU Octave on your own computer. Also, it seems that Octave Online is not comfortable with big data. Finally, I have limited experience in using Octave Online, so I may not be able to help with the problems you will experience in using it.

Getting Help

The most obvious way of getting help on GNU Octave is to switch to the Documentation tab next to the Command Window tab at the bottom of your GNU Octave GUI window. This will open the GNU Octave Manual. You can also find this manual in https://octave.org/doc/v6.2.0/. (This is the manual for version 6.2.0. I leave it to you to guess where you can find the manual for version 6.3.0.)

GNU Octave also has Wiki pages https://wiki.octave.org/GNU_Octave_Wiki with FAQ https://wiki.octave.org/FAQ. This is a good place to start if you have problems.

There is also a GNU Octave community https://octave.discourse.group/, where you can ask for help. This is a good place to ask if you have specific advanced problems or you think you have found a bug.

Finally, if you already know what you want to do, but do not know or remember the details, you can type `help <topic>` on the Command Window. The next Example 1.3 explains how to use **help**.

1.3 Example (Help tan)
Suppose you want to solve the equation

$$\tan x = 0.5. \tag{1.4}$$

Formally, a solution is

$$x = \tan^{-1} 0.5,$$

where $\tan^{-1}$ denotes an inverse of the function $\tan$. Unfortunately, the inverses of trigonometric functions have funny names. Also, some might interpret $\tan^{-1} y$ as $1/\tan y$. So, how do we solve (1.4) with GNU Octave? The best way to start is to see what GNU Octave has to say about the tan function. So, we type **help tan** in the Command Window:

```
1 >> help tan
2 'tan' is a built-in function from the file libinterp/corefcn/mappers.cc
3
4 -- tan (Z)
5 Compute the tangent for each element of X in radians.
6
7 See also: atan, tand, tanh.
```

```
8
9  Additional help for built-in functions and operators is
10 available in the online version of the manual.  Use the command
11 'doc <topic>' to search the manual index.
12
13 Help and information about Octave is also available on the WWW
14 at https://www.octave.org and via the help@octave.org
15 mailing list.
```

In the answer we spot (look at line 7) something vaguely familiar from our days in high school: atan. This looks like arcus tan or arctan, which is one name for the inverse of tan. So, let us ask help for that

```
1  >> help atan
2  'atan' is a built-in function from the file libinterp/corefcn/mappers.cc
3
4  -- atan (X)
5  Compute the inverse tangent in radians for each element of X.
6
7  See also: tan, atand.
8
9  Additional help for built-in functions and operators is
10 available in the online version of the manual.  Use the command
11 'doc <topic>' to search the manual index.
12
13 Help and information about Octave is also available on the WWW
14 at https://www.octave.org and via the help@octave.org
15 mailing list.
```

Looks correct. So, the solution for (1.4) is given by the following Command Window conversation.

```
1 >> atan(0.5)
2 ans = 0.4636
```

So, the answer to (1.4) is $x = 0.4636$.

Chapter 2

GNU Octave as Calculator

GNU Octave is a very convenient calculator. It can be used as a simple calculator (with variables), but its true power is in how it handles matrices and vectors.

Simple Calculations and Variables

Basic arithmetics with GNU Octave is pretty similar to any computing language: $+$ and $-$ signs are what they are, multiplication sign is $*$, division sign is $/$, and ˆ denote the exponentiation. Precedences follow some more or less natural rule, that may or may not be the same as you have learned in school. Thus, the use of parentheses is strongly encouraged.

2.1 Example (What is Precedence?)
Suppose we want to calculate

$$1.4 \times 10^{-2} + \frac{2}{7+3}. \tag{2.2}$$

This can be easily calculated by hand. Indeed, we obtain

$$\begin{aligned} 1.4 \times 10^{-2} + \frac{2}{7+3} &= \frac{1.4}{100} + \frac{2}{10} \\ &= \frac{1.4+20}{100} \\ &= 0.214. \end{aligned}$$

Let us then calculate the same with GNU Octave. Let us give a wrong answer first:

```
1 >> 1.4*10^-2   +   2 / 7+3
2 ans = 3.2997
```

This answer is wrong, because $2/7{+}3 \neq 2/(7{+}3)$. The human reader might read the command line 1 above differently because of all the extra spaces, but GNU Octave does not care about extra spaces. Also, while `10^-2` is technically correct, I strongly encourage to use parentheses there to avoid mistakes. Here is the correct way to calculate (2.2):

```
1 >> 1.4*10^(-2)   +   2/(7+3)
2 ans = 0.2140
```

2.3 Exercise (Viral Precedence Problem)
Calculate the following viral math problem

$$8 \div 2(2+2)$$

with GNU Octave.

2.4 Exercise (Rounding Errors)
Computers are fast, but they still make errors. One typical source of errors are rounding errors due to the way computers understand numbers. Make GNU Octave give a wrong aswer to you (due to rounding errors).

Hint: 1/3 is difficult for us humans using decimal system. Computers use binary system. For them 0.2 is difficult.

Let us then consider calculation with variables. Like most programming languages, GNU Octave can handle many different types of variables. GNU Octave uses weak typing meaning that we do not have to declare variables or their type like in Java, or C. If you don't know what this means, you are lucky. So, don't worry about it. In this section we only consider (real) numbers. In the next section we consider (real) vectors and (real) matrices. Later in the course we will also consider strings and a little bit struct (structure) type variables. Other variable types are not used in this course.

2.5 Example (Variables)
To assing a value $\frac{1}{3}$ to a variable x we can simply write

```
1 >> x = 1/3
2 x = 0.3333
```

Now we should see in the Workspace panel a line that says something like `x` under Name and `double` under Class. We can also ask what variables we have in our workspace by asking **who**

```
1 >> who
2 Variables visible from the current scope:
3
4 x
```

Since `x` is defined we can use it in calculations. For example, we can calculate $2^x = 2^{1/3}$ as

```
1 >> 2^x
2 ans = 1.2599
```

GNU Octave calculated the answer and assinged its value to the variable named **ans**. Indeed, if we now ask **who**, we obtain

```
1 >> who
2 Variables visible from the current scope:
3
4 ans   x
```

To see the values of the variables that are defined we can simply type them on the Command Window

```
1 >> ans
2 ans = 1.2599
3 >> x
4 x = 0.3333
```

2.6 Remark (Variable ans)

The variable **ans** that appeared in Example 2.5 above is used by GNU Octave to assing the value of the latest answer, unless we assign its value to a different variable. Example 2.7 below illustrates this point.

2.7 Example (clc, who, clear, and ans)

In the following discussion we start by clearing the visible Command Window by using the command **clc** (you can also press CTRL+L), and then clear all the variables in the workspace by using the command **clear**, and then assing some values to some variables.

```
1 >> clc
2 >> who
3 Variables visible from the current scope:
4
5 ans  x
6
7 >> clear
8 >> who
9 >> x
10 error: 'x' undefined near line 1, column 1
11 >> x = 1/3
12 x = 0.3333
13 >> y = 2^(1/3)
14 y = 1.2599
15 >> who
16 Variables visible from the current scope:
17
18 x  y
```

So, after line 1 the Command Window becomes empty, but the variables are not cleared as is shown by the **who** command in line 2. Lines 7 and 8 show that after the **clear** command the workspace is empty of variables. Indeed, asking for x in line 9 gives a more or less obvious error message. Now, in line 11 we define x and GNU Octave annoyingly echoes back what happened. In line 13 we define y to be the calculation we did earlier in Example 2.5, when we did not assing a variable to the solution. Since we now asked the answer to be stored in the variable y, GNU Octave does not use the auxiliary variable **ans** as in Example 2.5. This can be seen by asking **who** in line 15. We obtain the result that there are two variables, x and y, defined. The variable **ans** is not defined.

In Example 2.7 we noticed the annoying thing that GNU Octave echoes back (almost) everything we do. Quite often this is not what we want. Indeed, we usually want GNU Octave to perform the calculations quietly and only give the answer when we ask for it. We can tell GNU Octave to be quiet by ending our commands with semicolon. So, the semicolon works very differently in GNU Octave than in Java or C!

2.8 Example (Shut-Up Semicolon)
Compare this with Example 2.7:

```
1 >> clc
2 >> clear
3 >> who
4 >> x = 1/3;
5 >> y = 2^x;
6 >> who
7 Variables visible from the current scope:
8
9 x   y
10
11 >> x
12 x = 0.3333
13 >> y
14 y = 1.2599
```

2.9 Remark (Characters in Names)
Finally, a word of warning: GNU Octave, like most programming languages, is case sensitive. This means that variables x and X are not the same. Also, there are some restrictions on the variable names. As a general rule, do not start a name with a number and do not use non-English letters like ä or ø. Obviously, you should not use spaces or special characters (except the underscore _) in variable names.

Matrices and Vectors

GNU Octave, being a Matlab clone, is matrix oriented. This means that all the basic arithmetics is designed to work for matrices.

A matrix is defined in GNU Octave by presenting its elements inside brackets row-by-row. Elements in the row are separated by a comma or simply by a space. The rows are separated by a semicolon (this is the second use of semicolon in GNU Octave). If A is a matrix, then A(i,j) is the element in its ith row and jth column.

2.10 Example (Matrices and Their Elements)
Let us consider the matrix

$$\mathbf{A} = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \end{bmatrix}.$$

To introduce this matrix to GNU Octave we can assing

```
>> A = [11 12 13 14; 21 22 23 24]
A =

11    12    13    14
21    22    23    24
```

Now, for $\mathbf{A} = [A_{ij}]$ we have $A_{13} = A_{1,3} = 13$ and $A_{31} = A_{3,1}$ does not exist. The following conversation is how GNU Octave sees this:

```
>> A(1,3)
ans = 13
>> A(3,1)
error: A(3,_): out of bound 2 (dimensions are 2x4)
```

A vector is simply a matrix that has either one row or one column. If it has only one column, it is a column vector. If it has only one row, it is a row vector. If not otherwise stated, we will assume that all vectors are column vectors. Also note that vectors that have only one row and one column are called numbers.

Matrix algebra works with GNU Octave with the obvious and convenient way: if $\mathbf{A}$ and $\mathbf{B}$ are matrices (with suitable dimensions), then the transpose $\mathbf{A}' = \mathbf{A}^\top$ of $\mathbf{A}$ is in GNU Octave `A'`. Sums like $\mathbf{A}+\mathbf{B}$ and $\mathbf{A}-\mathbf{B}$ are obvious, the product (when defined) $\mathbf{AB}$ is `A*B`. Matrix power $\mathbf{A}^n$ is obviously `A^n`, the inverse matrix $\mathbf{A}^{-1}$ can be written as `A^(-1)` or `inv(A)`. Finally, the matrix equation (a.k.a. a system of linear equations) $\mathbf{Ax} = \mathbf{b}$ can be solved by the left-division \ as `x=A\b`. It is left for the reader to figure out what the right-division / does.

The n-by-n identity matrix $\mathbf{I} = \mathbf{I}_n$ can be constructed with GNU Octave by using the (terrible) pun `eye(n)` as the following console command shows:

```
>> eye(4)
ans =

Diagonal Matrix

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

In this course we do not use matrix algebra much as such. We will however use block matrices a lot. This means that we will cut matrices apart and paste them together. This can become very complicated very quickly. So, instead of giving the full exposure, we will just consider some cases now. More complicated constructions will be given later.

Let $\mathbf{A}$ be some matrix. If we want to refer to its row i, a typical mathematical notation is $\mathbf{A}_{i\bullet}$. One way to understand this is that the row i is fixed, but the columns "run free". With GNU Octave colon takes the place of the bullet: $\mathbf{A}_{i\bullet}$ becomes `A(i,:)` In the same way, $\mathbf{A}_{\bullet j}$ and `A(:,j)` refer to the fixed column j of the matrix $\mathbf{A}$.

2.11 Example (Matrix Rows and Columns via Colon)
The following console conversation should explain what is going on with the bullet/colon notation.

```
>> A = [1 0 0 2 0; 3 6 0 0 7]
A =

1     0     0     2     0
3     6     0     0     7

>> A(1,:)
ans =

1     0     0     2     0

>> A(2,:)
ans =

3     6     0     0     7

>> A(:,1)
ans =

1
3

>> A(:,2)
ans =

0
6

>> A(:,6)
error: A(_,6): out of bound 5 (dimensions are 2x5)
```

We can also take out a block of matrix. In GNU Octave this means that we use A(v1,v2), where v1 and v2 are vectors that indicate which rows and columns we take.

2.12 Example (Matrix Blocks)
The following discussion should explain how the cutting block construct A(v1,v2) works.

```
>> A = [11 12 13 14; 21 22 23 24; 31 32 33 34]
A =

11     12     13     14
21     22     23     24
31     32     33     34

>> v1 = [1 3]
v1 =

```

```
11 1     3
12
13 >> v2 = [3  4]
14 v2 =
15
16 3     4
17
18 >> A(v1,v2)
19 ans =
20
21 13     14
22 33     34
23
24 >> A(v1,v2) = [0  0;  0  0]
25 A =
26
27 11     12      0      0
28 21     22     23     24
29 31     32      0      0
```

So here we first, in line 1, defined the matrix A. In line 8 we defined the (row or column, it does not matter) vector v1 to correspond rows 1 and 3. In line 13 we defined the (row or column, it does not matter) vector v2 to correspond columns 3 and 4. In line 19 we asked what is the submatrix of the matrix A, where only rows 1 and 3 and columns 3 and 4 are considered. Finally, in line 25 we asked GNU Octave to replace the 2-by-2 submatrix A(v1,v2) with a 2-by-2 zero matrix.

Submatrices, that is cutting matrices, as considered above can be tricky. Block matrices, that is pasting matrices, is arguably much simpler. Again the following console discussion of Example 2.13 should explain how to paste matrices.

2.13 Example (Block Matrix)

Let

$$\mathbf{A} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 760 \\ -98 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 2 \\ 4 \\ 0 \\ 4 \\ 0 \end{bmatrix}, \quad \text{and} \quad \mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

We want to construct the block matrix

$$\mathbf{T} = \begin{bmatrix} & \mathbf{c}' & & 1 \\ \mathbf{A} & & \mathbf{I} & \mathbf{b} \end{bmatrix} = \begin{bmatrix} 2 & 4 & 0 & 4 & 0 & 1 \\ 11 & 12 & 13 & 1 & 0 & 760 \\ 21 & 22 & 23 & 0 & 1 & -98 \end{bmatrix}.$$

This is done with the following console discussion

```
1 >> A = [11  12  13;  21  22  23]
2 A =
```

```
3
4 11    12    13
5 21    22    23
6
7 >> b = [760 -98]'
8 b =
9
10 760
11 -98
12
13 >> I = eye(2)
14 I =
15
16 Diagonal Matrix
17
18 1    0
19 0    1
20
21 >> c = [2 4 0 4 0]'
22 c =
23
24 2
25 4
26 0
27 4
28 0
29
30 >> T = [c' 1; A I b]
31 T =
32
33   2     4     0     4     0     1
34  11    12    13     1     0   760
35  21    22    23     0     1   -98
```

2.14 Exercise (Matrix Block)

Let

$$\mathbf{A} = \begin{bmatrix} 11 & 12 & 13 & 14 & 15 & 16 \\ 21 & 22 & 23 & 24 & 25 & 26 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{bmatrix}.$$

What is `A([2 3], [1 6])`?

2.15 Exercise (Marix Block with Colon Notation)

In GNU Octave there is a construct `n:m` that builds a vector of successive indices in the following way: `n:m=[n n+1 n+2 ... m]`. So, for example, `2:5 = [2 3 4 5]`.

Let

$$\mathbf{A} = \begin{bmatrix} 11 & 12 & 13 & 14 & 15 & 16 \\ 21 & 22 & 23 & 24 & 25 & 26 \\ 31 & 32 & 33 & 34 & 35 & 36 \\ 41 & 42 & 43 & 44 & 45 & 46 \end{bmatrix}.$$

What is `A(2:4, [1 6])`?

Chapter 3

Working with m-Files

So far we have used the Command Window when discussing with GNU Octave. This becomes very quickly very clumsy. In this chapter we introduce the so-called m-files that makes this discussion much more fluent.

There are two types of m-files: script files and function files.

Script m-Files

A script type m-file is simply a list of console commands. The advantage of using script m-files instead of the console are twofold:

(i) You can edit and debug the file and run all its command at once without the need executes many commands in a row.
(ii) You can format the file so that, for example, matrices can be adjusted by rows and columns.

Suppose we want to calculate the net present value of a cash-flow under a given interest rate. As all students with loans should know, the formula for this is

$$\text{npv} = \sum_{t=1}^{T} \frac{c_t}{(1+r)^t}, \tag{3.1}$$

where c_t is the cash received at the end of period t, T is the number of periods, and r is the (fixed) rate of interest during the periods.

3.2 Problem (Net Present Value)
Suppose we want to calculate the net present value of a cash-flow gives us 1 000 EUR, 5 000 EUR and 2 500 EUR respectively at the end of the following 3 years. We assume that the annual interest rate is fixed 0.5%. Thus the cash-flow is the vector $\mathbf{c} = [1000\ 5000\ 2500]'$ and the interest rate is $r = 0.005$. Calculating this with Console Window by using the formula (3.1) would be quite painful. So, we should do something else. We give two solutions by using script m-files in this section. The first one is quick and dirty and the second one is elegant but tedious. In both solutions we create a text file (m-files are technically text files having the

extension .m in their names) that is in the Current Directory of our GNU session. You can set the Current Directory in the field near the top of your Octave GUI window. The script file is executed by typing its name (without the .m extension) in the Command Window.

3.3 Solution (Net Present Value, Quick and Dirty)

Below is the code of a quick and dirty solution to the npv-problem 3.2 (indeed, it's a one-liner). The code is the contents of the m-file npv_qd.m, which can be downloaded from www.uwasa.fi/~tsottine/spicy_or/npv_qd.m

```
1 npv = 1000/((1+0.005)^1) + 5000/((1+0.005)^2) + 2500/((1+0.005)^3)
```

As you can see, this is very quick and very dirty. To get the solution, make sure that the file `npv_qd.m` is in your Current Directory. You should be able to see it listed in the File Browser panel (top left panel typically). To execute the file simply state its name without the .m extension. Here is the related Console Window discussion

```
1 >> npv_qd
2 npv = 8408.3
```

I strongly discourage using quick and dirty m files like in Example 3.3. One of the dirtiest and stupidest thing we did was that we plugged in the numbers in the formula. You should absolutely never ever do this! What we should have done is to give the numbers as variables and then implement the formula for GNU Octave in a general form. Of course, if you are in a hurry and do not intend to use the m file later, you can be quick and dirty. However, if you intend to use the m file later, I encourage you to do something like the more elegant solution in Solution 3.4 below.

3.4 Solution (Net Present Value, Tedious and Elegant)

Below is the lengthy code of an elegant but tedious solution for Problem 3.2. The code is the contents of the m-file npv_elegant.m, which can be downloaded here: www.uwasa.fi/~tsottine/spicy_or/npv_elegant.m

```
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %%
3 %% Net Present Value (Problem 3.2 from Octave with Spice;
4 %% Solution 3.4 - Tedious and Elegant)
5 %%
6 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
7
8 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
9 %% Data
10 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
11
12 cf = [1000 5000 2500]';      %% Cash-flow.
13 r  = 0.005;                  %% Fixed rate of interest.
14
```

```
15 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
16 %% Calculations
17 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
18
19 T = length(cf);                  %% Number of periods.
20
21 %% Calculate the npv sum iteratively
22 npv = 0;
23 for t=1:T
24     npv = npv + cf(t)/( (1+r)^t );
25 end
26
27 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
28 %% Output Net Present Value (npv)
29 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
30 npv
```

Compared to the quick and dirty solution of Example 3.3 this one is certainly tedious. But it is also elegant.

Let us analyze the code above on a block level:

The code is divided into 4 blocks: The 1st block, lines 1–7 just describes what this code is supposed to do. The 2nd block, lines 8–14, set the data. The key point here is that this code will also work with different data, i.e., if you want to change the cash-flow or the interest rate, you can do it here and the rest will work just fine. The 3rd block, lines 15–26 will do the actual calculations, so that's where the beef is. The final 4th block, lines 27–30, is for the output.

A line-by-line analysis of the code is:

The lines 1–11 do absolutety nothing. They are there for the readers' convenience. Indeed, the percentage symbol % at the begining of the lines 1–6 and 8–10 means that these lines are comments. GNU Octave will simply ignore these lines. GNU Octave will also ignore the empty lines 7 and 11. Also, the use of double comment signs %% is not necessary; one % would be enough. This is a matter of style. Indeed, a double comment sign %% should mean a permanent comment, while a single comment sign % should mean something that is used for debugging or testing purposes.

Line 12 defines the (column) vector `cf` to contain the cash-flow. There is a comment sign %% at the end of the line, which GNU Octave will ignore. This comment is here for the readers' convenience. Line 13 defines the interest rate in the same way.

Lines 14–18 do absolutely nothing.

Line 19 gets the number of periods from the vector `cf`. The idea here is that we infer the number of periods `T` from the length of the vector `cf`, and do not set it by hand. Indeed, this way the m-file will work just fine if we change the parameters in line 12–13.

Lines 20–21 do absolutely nothing.

Lines 22–25 are **the beef** of the code. They calculate the sum in the formula (3.1) in a for-loop. Type `help for` in GNU Octave to unerstand what is going on!

Again, lines 26–29, do absolutely nothing.

Finally, in line 30 we call the variable `npv` without semicolon. Thus GNU Octave will echo the value of the variable `npv` to the console, showing us the result.

Now, calling the script m-file npv_elegant.m in the console (make su you have the m-file in your Current Directory) solves the Net Present Value problem 3.2:

```
>> npv_elegant
npv = 8408.3
```

3.5 Exercise
Calculate the Net Present Value (npv) of having 100 000 EUR after 10 years when the annual interest rate is 50%.

Function m-Files

Suppose you want to calculate the net present value (3.1) for many different values of cash-flows and/or interest rates. Then it makes sense to have the npv as a function, and not to make separate elegant script m-files for all the different cf and r values. To do this, you should make a function m-file for npv.

A function m-file is an m-file that starts with the keyword **function**. So, GNU Octave will know not to try to parse this file as a script m-file. Instead, GNU Octave will only parse this file, if it will be called as a function with given parameters.

Below is the code for the function m-file for npv, that can be downloaded from www.uwasa.fi/~tsottine/spicy_or/npv.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Function v = npv(cf,r)  returns the Net Present Value (npv) of the cash flow
%% cf.   The cash flow  cf is received at the end of each period.   The rate of
%% return over the period is r.   The parameter r is scalar.   The cash flow cf
%% is a (column) vector.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function v = npv(cf,r)
    T = length(cf);                     %% The number of periods.
    pv = zeros(T,1);                    %% Initialize present values (pv) to zero.
    for t=1:T
        pv(t) = cf(t) / (1+r)^t;        %% Set the pv's.
    end
    v = sum(pv);                        %% npv is the sum of pv's.
end
```

Trying to call the function as it were a script m-file will lead to the following unsatisfying discussion with GNU Octave console:

```
>> npv
error: 'cf' undefined near line 9, column 9
error: called from
npv at line 9 column 7
```

A more fruitful conversation with GNU Octave console and npv-function could be

```
1 >> r = 0.05
2 r =  0.050000
3 >> cf = [0 0 0 1]
4 cf =
5
6 0   0   0   1
7
8 >> npv(cf,r)
9 ans =  0.8227
```

So, what happened here?

In the first discussion GNU Octave just tried to understand the m-file `npv.m` as a script type m-file. It skipped the lines starting with % signs and empty lines. Then it accepted the keyword **function** and all its wrappings. But at line 9 column 9, GNU Octave should consider length of the variable cf. But the variable cf has no value assigned to it. Thus, GNU Octave, lacking abstract thinking, just quit.

The second discussion is quite different. There the variables r and cf are given values. Then the function npv is called with variables that have values assigned to them. Thus GNU Octave just passes the values of r and cf to the code written in the function m-file npv, and everything works just fine.

3.6 Exercise (Variable Interest Rate Net Present Value)
Modify the function npv.m so that it assumes the interest rate be non-constant, i.e. the interest rate for period t ir r_t.

Chapter 4

Solving LP's with GLPK

A Linear Program (LP) is a linear optimization problem, where all the "things" are linear. This means that that the objective function you want to optimize (maximize or minimize) is a linear function of your decision variables, and so are also your constraints.

In this chapter we consider an LP called the **Product-Mix Problem** or **Product-Selection Problem**. We first introduce the problem. Then we model it mathematically, and finally we solve it by using the GNU Octave function glpk (GNU Linear Programming Kit).

Product-Mix Problem

4.1 Problem (Muad'Dib Bakery)

Muad'Dib Bakery produces three types of Fremen cakes: Atreides, Corrino and Harkonnen. Each cake is made out of nutrient powders that are: fat, sugar, protein, water, and spice Melange. The nutrient contents and selling prices (in solaris) of the cakes are:

Atreides Cake Fat 70 g, sugar 500 g, protein 280 g, water 600 ml, spice Melange 8 mg. Price: 120 solaris.

Corrino Cake Fat 50 g, sugar 300 g, protein 180 g, water 800 ml, spice Melange 35 mg. Price: 170 solaris.

Harkonnen Cake Fat 150 g, sugar 500 g, protein 50 g, water 500 ml, spice Melange 10 mg. Price: 100 solaris.

Muad'Dib Bakery has the following amount of nutrient powders at its disposal daily:

Fat 100 kg
Sugar 400 kg
Protein 150 kg
Water 300 l
Spice Melange 10 g

Muad'Dib Bakery wants to maximize daily revenues. What should Muad'Dib Bakery do?

To model mathematically optimization problems like 4.1 one may follow the following three-step procedure:

4.2 Algorithm (Optimization modeling)

Step 1 Find the **decision variables**, i.e. find out what are the variables whose values you can choose.

Step 2 Find the **objective function**, i.e. find out how your objective to be minimized or maximized depends on the decision variables.

Step 3 Find the **constraints**, i.e. find out the (in)equalities that the decision variables must satisfy. (Don't forget the possible sign constraints!)

Let us then start the **mathematical modeling phase** following Algorithm 4.2 above.

Muad'Dib Bakery produces three different types of cakes. So, the **decision variables** are:

$$\begin{aligned} x_1 &= \text{number of Atreides cakes produced daily,} \\ x_2 &= \text{number of Corrino cakes produced daily,} \\ x_3 &= \text{number of Harkonnen cakes produced daily.} \end{aligned}$$

Once the decision variables $\mathbf{x} = [x_1\ x_2\ x_3]'$ are known, the **objective function** $z = z(\mathbf{x})$ of this problem is simply the revenue

$$z = 120x_1 + 170x_2 + 100x_3$$

Note that the profit z depends linearly on the number of produced cakes x_1, x_2, x_3. So, this is a linear problem so far (the constraints must turn out to be linear, too).

It may seem at first glance that the profit can be maximized by simply increasing x_1, x_2 and x_2. Well, if life were that easy, let's all start manufacturing Fremen cakes and become infinitely rich! Unfortunately, there are **constraints** that limit the decisions (or else the model is very likely to be wrong). Indeed, we need nutrient powders, and these are scarce resources.

Let us consider the constraint induced by the limited supply of **fat powder**. If we decide to produce x_1 Atreides cakes, x_2 Corrino cakes, and x_3 Harkonnen cakes, then the amount (in grams) of fat powder we have used is $70x_1 + 50x_2 + 150x_3$ grams. Since we only have 100 kg of fat powder, we have the constraint (note the unit change 1 kg = 1 000 g)

$$70x_1 + 50x_2 + 150x_3 \leq 100\ 000.$$

So, this is the constraint induced by the scarcity of fat powder. Note that this is a linear constraint. This is good. Indeed, if it were not a linear constraint we wouldn't be able to solve it with GNU Octave's function glpk.

Similarly, for **sugar powder** we have the linear constraint

$$500x_1 + 300x_2 + 500x_3 \leq 400\ 000,$$

since there is only 400 kg = 400 000 g of sugar powder available daily, and the requirement per cake for Atreides, Corrino, and Harkonnen type cakes are 500 g, 300 g and 500 g, respectively.

The daily supply for **protein powder** is 150 kg = 150 000 g. Consequently, we obtain the linear constraint

$$280x_1 + 180x_2 + 50x_3 \;\leq\; 150\ 000,$$

since Atreides cake needs 280 g of protein, Corrino cake need 180 g of protein, and Harkonnen cake needs 50 g of protein.

For **water powder** we obtain the linear constraint

$$600x_1 + 800x_2 + 500x_3 \;\leq\; 300\ 000,$$

since 300 l = 300 000 ml, and the requirements per cake types are 600 ml, 800 ml, and 500 ml.

Finally, for **spice Melange** the linear constraint is

$$8x_1 + 35x_2 + 10x_3 \;\leq\; 10\ 000,$$

for reasons that should be obvious by now.

The mathematical model for Muad'Dib Bakery is now ready. Indeed, putting all that we have found out together in a compact form, we have

$$
\begin{array}{llrcrcrcrl}
\max z & = & 120x_1 & + & 170x_2 & + & 100x_3 & & & \text{(revenue)} \\
\text{s.t.} & & 70x_1 & + & 50x_2 & + & 150x_3 & \leq & 100\ 000 & \text{(fat)} \\
 & & 500x_1 & + & 300x_2 & + & 500x_3 & \leq & 400\ 000 & \text{(sugar)} \\
 & & 280x_1 & + & 180x_2 & + & 50x_3 & \leq & 150\ 000 & \text{(protein)} \\
 & & 600x_1 & + & 800x_2 & + & 500x_3 & \leq & 300\ 000 & \text{(water)} \\
 & & 8x_1 & + & 35x_2 & + & 10x_3 & \leq & 10\ 000 & \text{(spice)} \\
 & & & & & & x_1, x_2, x_3 & \geq & 0 & \text{(sign constraints)}
\end{array}
\tag{4.3}
$$

Note the last **sign constraints**. They ensure that the values of the decision variables will always be positive (Muad'Dib Bakery sells cakes; it does not buy them). The problem does not state this explicitly, but it's still important (and obvious).

4.4 Remark (Muad'Dib LP with Matrices)

The compact form (4.3) can be written even more compactly by using matrices. Indeed, let $\mathbf{x} = [x_1\ x_2\ x_3]'$ be the column vector containing the three decision variables. Let $\mathbf{c} = [120\ 170\ 100]'$ be the column vector containing the prices associated with the decision variables. Then, the **objective function** z can identified with the vector $\mathbf{c}$ as

$$z \;=\; \mathbf{c}'\mathbf{x}.$$

Let then

$$\mathbf{A} \;=\; \begin{bmatrix} 70 & 50 & 150 \\ 500 & 300 & 500 \\ 280 & 180 & 50 \\ 600 & 800 & 500 \\ 8 & 35 & 10 \end{bmatrix}.$$

This means that $\mathbf{A}$ is the **technology matrix** that corresponds to how the different products (cakes) are produced from the resources (nutrient powders). The final piece of data is the

column vector $\mathbf{b} = [10000\ 400000\ 150000\ 300000\ 10000]'$ that tells the number of **available resources**. With $\mathbf{c}$, $\mathbf{A}$, and $\mathbf{b}$ the Muad'Dib Bakery LP (4.3) can be written very compactly as

$$
\begin{array}{llrcl}
\max & z = \mathbf{c}'\mathbf{x} & & & \\
\text{s.t.} & & \mathbf{A}\mathbf{x} & \leq & \mathbf{b}\ . \\
 & & \mathbf{x} & \geq & \mathbf{0}
\end{array}
\tag{4.5}
$$

4.6 Exercise (Limited Atreides, Part I)
Suppose that there is only limited demand for Atreides cakes: only 100 Atreides cakes can be sold daily. Model the Muad'Dib Bakery problem so that it takes into account this limitation.

4.7 Exercise (Harkonnen Promise, Part I)
Consider the Muad'Dib Bakery problem 4.1. Assume that the bakery has promised to bake at least one Harkonnen cake each day. Model this new problem mathematically, i.e., find the LP model similar to (4.3) for this problem.

Hint: You will have a new constraint with a lower bound instead of an upper bound.

Now that the mathematical modeling phase is finished, there is the **implementation phase**, i.e. we have to tell the problem to a computer so that it can solve it. In the next section we show how to do this with GNU Octave and its GLPK solver.

Solving Product-Mix Problem with GLPK

There are many LP (Linear Program) and MILP (Mixed Integer Linear Program) solvers in the world. In GNU Octave, one has GLPK (GNU Linear Programming Kit). Here you have its help (version 6.2.0):

```
1 >> help glpk
2 'glpk' is a function from the file /snap/octave/78/share/octave/6.2.0/m/
    optimization/glpk.m
3
4 -- [XOPT, FMIN, ERRNUM, EXTRA] = glpk (C, A, B, LB, UB, CTYPE,
5 VARTYPE, SENSE, PARAM)
6 Solve a linear program using the GNU GLPK library.
7
8 Given three arguments, 'glpk' solves the following standard LP:
9
10 min C'*x
11
12 subject to
13
```

```
14 A*x  = b
15 x >= 0
16
17 but may also solve problems of the form
18
19 [ min | max ] C'*x
20
21 subject to
22
23 A*x [ "=" | "<=" | ">=" ] b
24 x >= LB
25 x <= UB
26
27 Input arguments:
28
29 C
30 A column array containing the objective function coefficients.
31
32 A
33 A matrix containing the constraints coefficients.
34
35 B
36 A column array containing the right-hand side value for each
37 constraint in the constraint matrix.
38
39 LB
40 An array containing the lower bound on each of the variables.
41 If LB is not supplied, the default lower bound for the
42 variables is zero.
43
44 UB
45 An array containing the upper bound on each of the variables.
46 If UB is not supplied, the default upper bound is assumed to
47 be infinite.
48
49 CTYPE
50 An array of characters containing the sense of each constraint
51 in the constraint matrix.  Each element of the array may be
52 one of the following values
53
54 "F"
55 A free (unbounded) constraint (the constraint is
56 ignored).
57
58 "U"
59 An inequality constraint with an upper bound ('A(i,:)*x
60 <= b(i)').
61
62 "S"
63 An equality constraint ('A(i,:)*x = b(i)').
64
65 "L"
66 An inequality with a lower bound ('A(i,:)*x >= b(i)').
67
68 "D"
69 An inequality constraint with both upper and lower bounds
70 ('A(i,:)*x >= -b(i)') _and_ ('A(i,:)*x <= b(i)').
```

```
71
72 VARTYPE
73 A column array containing the types of the variables.
74
75 "C"
76 A continuous variable.
77
78 "I"
79 An integer variable.
80
81 SENSE
82 If SENSE is 1, the problem is a minimization.  If SENSE is -1,
83 the problem is a maximization.  The default value is 1.
84
85 PARAM
86 A structure containing the following parameters used to define
87 the behavior of solver.  Missing elements in the structure
88 take on default values, so you only need to set the elements
89 that you wish to change from the default.
90
91 Integer parameters:
92
93 'msglev (default: 1)'
94 Level of messages output by solver routines:
95
96 0 ('GLP_MSG_OFF')
97 No output.
98
99 1 ('GLP_MSG_ERR')
100 Error and warning messages only.
101
102 2 ('GLP_MSG_ON')
103 Normal output.
104
105 3 ('GLP_MSG_ALL')
106 Full output (includes informational messages).
107
108 'scale (default: 16)'
109 Scaling option.  The values can be combined with the
110 bitwise OR operator and may be the following:
111
112 1 ('GLP_SF_GM')
113 Geometric mean scaling.
114
115 16 ('GLP_SF_EQ')
116 Equilibration scaling.
117
118 32 ('GLP_SF_2N')
119 Round scale factors to power of two.
120
121 64 ('GLP_SF_SKIP')
122 Skip if problem is well scaled.
123
124 Alternatively, a value of 128 ('GLP_SF_AUTO') may be also
125 specified, in which case the routine chooses the scaling
126 options automatically.
127
```

```
128 'dual (default: 1)'
129 Simplex method option:
130
131 1 ('GLP_PRIMAL')
132 Use two-phase primal simplex.
133
134 2 ('GLP_DUALP')
135 Use two-phase dual simplex, and if it fails, switch
136 to the primal simplex.
137
138 3 ('GLP_DUAL')
139 Use two-phase dual simplex.
140
141 'price (default: 34)'
142 Pricing option (for both primal and dual simplex):
143
144 17 ('GLP_PT_STD')
145 Textbook pricing.
146
147 34 ('GLP_PT_PSE')
148 Steepest edge pricing.
149
150 'itlim (default: intmax)'
151 Simplex iterations limit. It is decreased by one each
152 time when one simplex iteration has been performed, and
153 reaching zero value signals the solver to stop the
154 search.
155
156 'outfrq (default: 200)'
157 Output frequency, in iterations. This parameter
158 specifies how frequently the solver sends information
159 about the solution to the standard output.
160
161 'branch (default: 4)'
162 Branching technique option (for MIP only):
163
164 1 ('GLP_BR_FFV')
165 First fractional variable.
166
167 2 ('GLP_BR_LFV')
168 Last fractional variable.
169
170 3 ('GLP_BR_MFV')
171 Most fractional variable.
172
173 4 ('GLP_BR_DTH')
174 Heuristic by Driebeck and Tomlin.
175
176 5 ('GLP_BR_PCH')
177 Hybrid pseudocost heuristic.
178
179 'btrack (default: 4)'
180 Backtracking technique option (for MIP only):
181
182 1 ('GLP_BT_DFS')
183 Depth first search.
184
```

```
185 2 ('GLP_BT_BFS')
186 Breadth first search.
187
188 3 ('GLP_BT_BLB')
189 Best local bound.
190
191 4 ('GLP_BT_BPH')
192 Best projection heuristic.
193
194 'presol (default: 1)'
195 If this flag is set, the simplex solver uses the built-in
196 LP presolver. Otherwise the LP presolver is not used.
197
198 'lpsolver (default: 1)'
199 Select which solver to use. If the problem is a MIP
200 problem this flag will be ignored.
201
202 1
203 Revised simplex method.
204
205 2
206 Interior point method.
207
208 'rtest (default: 34)'
209 Ratio test technique:
210
211 17 ('GLP_RT_STD')
212 Standard ("textbook").
213
214 34 ('GLP_RT_HAR')
215 Harris' two-pass ratio test.
216
217 'tmlim (default: intmax)'
218 Searching time limit, in milliseconds.
219
220 'outdly (default: 0)'
221 Output delay, in seconds. This parameter specifies how
222 long the solver should delay sending information about
223 the solution to the standard output.
224
225 'save (default: 0)'
226 If this parameter is nonzero, save a copy of the problem
227 in CPLEX LP format to the file '"outpb.lp"'. There is
228 currently no way to change the name of the output file.
229
230 Real parameters:
231
232 'tolbnd (default: 1e-7)'
233 Relative tolerance used to check if the current basic
234 solution is primal feasible. It is not recommended that
235 you change this parameter unless you have a detailed
236 understanding of its purpose.
237
238 'toldj (default: 1e-7)'
239 Absolute tolerance used to check if the current basic
240 solution is dual feasible. It is not recommended that
241 you change this parameter unless you have a detailed
```

```
242 understanding of its purpose.
243
244 'tolpiv (default: 1e-10)'
245 Relative tolerance used to choose eligible pivotal
246 elements of the simplex table. It is not recommended
247 that you change this parameter unless you have a detailed
248 understanding of its purpose.
249
250 'objll (default: -DBL_MAX)'
251 Lower limit of the objective function. If the objective
252 function reaches this limit and continues decreasing, the
253 solver stops the search. This parameter is used in the
254 dual simplex method only.
255
256 'objul (default: +DBL_MAX)'
257 Upper limit of the objective function. If the objective
258 function reaches this limit and continues increasing, the
259 solver stops the search. This parameter is used in the
260 dual simplex only.
261
262 'tolint (default: 1e-5)'
263 Relative tolerance used to check if the current basic
264 solution is integer feasible. It is not recommended that
265 you change this parameter unless you have a detailed
266 understanding of its purpose.
267
268 'tolobj (default: 1e-7)'
269 Relative tolerance used to check if the value of the
270 objective function is not better than in the best known
271 integer feasible solution. It is not recommended that
272 you change this parameter unless you have a detailed
273 understanding of its purpose.
274
275 Output values:
276
277 XOPT
278 The optimizer (the value of the decision variables at the
279 optimum).
280
281 FOPT
282 The optimum value of the objective function.
283
284 ERRNUM
285 Error code.
286
287 0
288 No error.
289
290 1 ('GLP_EBADB')
291 Invalid basis.
292
293 2 ('GLP_ESING')
294 Singular matrix.
295
296 3 ('GLP_ECOND')
297 Ill-conditioned matrix.
298
```

```
299 4 ('GLP_EBOUND')
300 Invalid bounds.
301
302 5 ('GLP_EFAIL')
303 Solver failed.
304
305 6 ('GLP_EOBJLL')
306 Objective function lower limit reached.
307
308 7 ('GLP_EOBJUL')
309 Objective function upper limit reached.
310
311 8 ('GLP_EITLIM')
312 Iterations limit exhausted.
313
314 9 ('GLP_ETMLIM')
315 Time limit exhausted.
316
317 10 ('GLP_ENOPFS')
318 No primal feasible solution.
319
320 11 ('GLP_ENODFS')
321 No dual feasible solution.
322
323 12 ('GLP_EROOT')
324 Root LP optimum not provided.
325
326 13 ('GLP_ESTOP')
327 Search terminated by application.
328
329 14 ('GLP_EMIPGAP')
330 Relative MIP gap tolerance reached.
331
332 15 ('GLP_ENOFEAS')
333 No primal/dual feasible solution.
334
335 16 ('GLP_ENOCVG')
336 No convergence.
337
338 17 ('GLP_EINSTAB')
339 Numerical instability.
340
341 18 ('GLP_EDATA')
342 Invalid data.
343
344 19 ('GLP_ERANGE')
345 Result out of range.
346
347 EXTRA
348 A data structure containing the following fields:
349
350 'lambda'
351 Dual variables.
352
353 'redcosts'
354 Reduced Costs.
355
```

```
356 'time'
357 Time (in seconds) used for solving LP/MIP problem.
358
359 'status'
360 Status of the optimization.
361
362 1 ('GLP_UNDEF')
363 Solution status is undefined.
364
365 2 ('GLP_FEAS')
366 Solution is feasible.
367
368 3 ('GLP_INFEAS')
369 Solution is infeasible.
370
371 4 ('GLP_NOFEAS')
372 Problem has no feasible solution.
373
374 5 ('GLP_OPT')
375 Solution is optimal.
376
377 6 ('GLP_UNBND')
378 Problem has no unbounded solution.
379
380 Example:
381
382 c = [10, 6, 4]';
383 A = [ 1, 1, 1;
384 10, 4, 5;
385 2, 2, 6];
386 b = [100, 600, 300]';
387 lb = [0, 0, 0]';
388 ub = [];
389 ctype = "UUU";
390 vartype = "CCC";
391 s = -1;
392
393 param.msglev = 1;
394 param.itlim = 100;
395
396 [xmin, fmin, status, extra] = ...
397 glpk (c, A, b, lb, ub, ctype, vartype, s, param);
398
399 Additional help for built-in functions and operators is
400 available in the online version of the manual. Use the command
401 'doc <topic>' to search the manual index.
402
403 Help and information about Octave is also available on the WWW
404 at https://www.octave.org and via the help@octave.org
405 mailing list.
```

Quite a lot, eh!

If you look at the lines 4–5 of the help listing above, you'll find that there are three main ingredients in the (Muad'Dib Bakery) LP problem:

- The **objective function**, typically denoted by c.

- The **technology matrix**, typically denoted by A.
- The constraint (upper) bounds, or available **resources**, typically denoted by b.

Additionally, we must tell the function glpk what are the lower bounds lb, the upper bounds ub, the types of the constraints ctype, and the sense (max or min) of the problem sense. Finally there is the input parameter param, but this is optional. For now, we do not care about it. The parameters c, A and b are given in Remark 4.4. As for lb, the only lower bounds we have are the sign constraints. From lines 39–42 we see that the default lower bound are the sign constraints. So, we use the default. This means that we give the nothing for the lower bounds. In GNU Octave the nothing is the **empty matrix** []. As for the upper bounds ub, we do not have any. Indeed, the upper bounds b are not for the decision variables, but for the technology matrix. So, we give the nothing [] for the upper bounds also. Then we have the ctype, that is the type of bounds associated with the technology matrix A. They are all upper bounds, i.e. of type $\leq$. There are 5 of them. Thus, we will have ctype="UUUUU". Then we have to tell the types of our 3 decision variables x_1, x_2 and x_3. We will assume that they are continuous. Indeed, we will almost always assume this. So, vartype="CCC". Finally, we must tell glpk whether we are maximizing or minimizing. We are maximizing. So according to lines 81–83 of the help listing we set sense=−1.

Having set the parameters c, A and b, we can solve Problem 4.1 by calling the function glpk as

```
1 [x_max, z_max] = glpk(c,A,b, [],[],"UUUUU","CCC",-1).
```

Then the optimal decision will be in the vector x_max and the optimal revenue will be in the scalar z_max.

4.8 Solution (Muad'Dib Bakery)

Here is the script m-file that solves the Muad'Dib Bakery problems 4.1 (you can downdoad it from www.uwasa.fi/~tsottine/spicy_or/muaddib.m)

```
1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %%
3  %% Muad'Dib Bakery. (Problem 4.1 from Octave with Spice)
4  %%
5  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6
7  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
8  %% Data
9  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
10
11 %% Prices for Atreides, Corrino, Harkonnen (Solari)
12 c = [120 170 100]';
13
14 %% Nutrient powder composition of the cakes Atreides, Corrino, Harkonnen
15 A = [ 70   50  150;               %% Fat (g)
16      500  300  500;               %% Sugar (g)
17      280  180   50;               %% Protein (g)
18      600  800  500;               %% Water (ml)
19        8   35   10 ];             %% Spice Melange (mg)
20
21 %% Available nutrients (note unit scale)
22 b = 1000*[100 400 150 300 10]';
```

```
23
24 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
25 %% Solution with GLPK
26 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
27
28 [x_max, z_max] = glpk(c,A,b, [], [], "UUUUU", "CCC", -1);
29
30 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
31 %% Output: x_max for the optimal decision and z_max for the optimal value.
32 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
33 x_max
34 z_max
35
36 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
37 %% Test result:
38 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
39 %%>> muaddib
40 %%x_max =
41 %%
42 %%    171.2329
43 %%    246.5753
44 %%           0
45 %%
46 %%z_max = 6.2466e+04
```

Let us explain a little bit what happens in the m-file muaddib.m.

Lines 1–5 are comment lines that do absolutely nothing. They are there for the human reader so that they see what this m-file is all about. You might also want to type **help** muaddib in the GNU Octave console, just for fun.

Empty lines, like line 6, do absolute nothing.

Lines 7–10 do absolutely nothing, except they tell the human reader that next we have a block where the data of the problem is given.

Line 11 is a comment line that tells that next the prices of the cakes are given.

In line 12 the prices of the cakes are stored to the vector c. The apostrophe makes the vertor a column vector and the semicolon in the end suppresses the output, i.e., GNU Octave will not echo to the console what just happened.

Line 14 is a comment line that tells the human reader that next the technology matrix is given.

The technology matrix A is given in lines 15–19. Each line ends with a comment for the human readers' convenience. Note the semicolon in line 19, which tell the GNU Octave to shut up instead of shouting what just happens in the console.

You should be able to guess by now what the line 21 does.

The resource vector b in given in line 22. Note the change of scale: each element in the vector is multiplied by 1 000.

Lines 24–26 tell the human reader that the data section is finished and next the solution is calculated.

The solution is calculated in the one-liner 28. The result is stored in the variables x_max and z_max. Note the semicolon in the end of line 28. This means that the solution is not echoed to the console.

Lines 30-32 tells the human reader that next we will ask GNU Octave to print the solutions. Indeed, so far we have told GNU OCtave to keep quiet. If we just type a variable name, its value will be printed in the console.

Finally, the commented section 36–46 is there just for test purposes. Indeed, you sould run the m-file muaddib.m and check that you get the same result as given there.

Let us then run the m-file muaddib.m in the GNU Octave console (make sure that the m-file is in you current directory). We get the solution:

```
1 >> muaddib
2 x_max =
3
4 171.2329
5 246.5753
6 0
7
8 z_max = 6.2466e+04
```

So, Muad'Dib Bakery should produce 171.239 Atreides cakes, 246.5753 Corrino cakes, and no Harkonnen cakes at all. With this choise the daily revenue is maximized and it is 62 466 solaris.

4.9 Exercise (LP with No Story)
Consider the following LP that has no story:

$$
\begin{array}{lcrcrcrcrcr}
\max z & = & 12x_1 & + & 4x_2 & + & 8x_3 & & & & \\
\text{s.t.} & & 7x_1 & + & 6x_2 & + & 3x_3 & - & 2x_4 & \leq & 20 \\
 & & & & 5x_2 & + & 4x_3 & + & 3x_4 & \leq & 16 \\
 & & 3x_1 & + & x_2 & & & & & = & 9 \\
 & & 2x_1 & + & x_2 & & & - & x_4 & \geq & 2 \\
 & & & - & x_2 & + & x_3 & & & \geq & 1 \\
 & & & & & & & x_1, x_2, & x_3, x_4 & \geq & 0
\end{array} .
$$

Solve it with glpk.

4.10 Exercise (Limited Atreides, Part II)
Solve the problem of Exercise 4.6 with glpk.

4.11 Exercise (Harkonnen Promise, Part II)
Solve the problem of Exercise 4.7 with glpk.

4.12 Remark (LP vs. IP vs. MILP)
The optimal solution 4.8 of Muad'Dib Bakery problem 4.1 turned out to be **fractional**: The number of cakes produced are not whole numbers (or integers). It was not clearly stated that the cakes should be whole. If we insist that the all the cake types produced must be whole cakes, then instead of an LP (Linear Program) we have an IP (Integer Progam). If we only insist that some of the cake types (Atreides, say) must be whole cakes, then we are dealing with a MILP (Mixed Integer Linear Program).

4.13 Exercise (Muad'Dib Bakery with Integrity)
Solve the Muad'Dib Bakery problem 4.1 when only whole cakes can be sold.

Hint: Look at the glpk's 7th input parameter vartype.

4.14 Remark (Muad'Dib Kwisatz Haderach)
Let us take a sneak peek into what we are going to learn during the Operations Research course. At this point there is no need to fully understand what is going on here. Everything will be explained in full detail later.

So far we have asked glpk only to tell us x and z the optimal decision and the optimal value (max or min), i.e, we have used glpk in the form

```
1 [x, z] = glpk(c,A,b, lb, ub, ctype, vartype, sense)
```

We have omitted the optional input parameter param, and we shall usually continue to do so. Here, we use it a little bit, just to a have a small flavor of it. The parameter param (just p later) is actually a **tuning paramer**. It is of **struct** type which means it has different fields that can be set to different values. To go through all the possible tuning possibilities would take too long, and would probably be almost pointless. We will consider only to fields here:

- msglev controls how much glpk will tell us what it is doing. Value 0 means blissful quietness. Value 3 means annoying babbling.
- presol set whether glpk uses a so-called presolver (by default it does) or not. The idea of the presolver is to try some easy tricks before going to a "full" solver.

In the example below we will set p.msglev=3 so that glpk will tell us "everything" it is doing. We will also set p.presol=0, i.e., we will tell glpk not to use a presolver.

Let us consider the optional output parameters e and xtra. That is, let us consider calling glpk as

```
1 [x, z, e, xtra] = glpk(c,A,b, lb, ub, ctype, vartype, sense, p)
```

The third output parameter e is of no interest to us, except in the case if something went wrong. The fourth output parameter xtra, on the other hand, is very interesting. The output parameter xtra is of **struct** type. This means that it is a structural variable that contains different fields. Its fields are:

- lambda for the **dual variables** a.k.a. **shadow prices** a.k.a. **marginal prices**.
- redcosts for the **reduced costs** a.k.a. **opportunity costs**.
- time for the time it took to calculate the solution.
- status for the status of the solution. (This is a funny field, since it seems to be redundant given the return value e. The author is truly confused why this even exitsts.)

Here is a solution to the Muad'Dib Bakery Problem 4.1 where we have tuned glpk so that it babbles a lot and does not use a presolver. Also, we asked for full output x, z, e, xtra. You can donwload the solution m-file muaddib_kh here: www.uwasa.fi/~tsottine/spicy_or/muaddib_kh.m

```
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %%
3 %% Muad'Dib Kwisatz Haderach. (Remark 4.14 from Octave with Spice)
4 %%
5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6
7 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
8 %% Data
9 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
10
11 %% Prices for Atreides, Corrino, Harkonnen (Solari)
12 c = [120 170 100]';
13
14 %% Nutrient powder composition of the cakes Atreides, Corrino, Harkonnen
15 A = [ 70  50 150;                %% Fat (g)
16      500 300 500;                %% Sugar (g)
17      280 180  50;                %% Protein (g)
18      600 800 500;                %% Water (ml)
19        8  35  10 ];              %% Spice Melange (mg)
20
21 %% Available nutrients (note unit scale)
22 b = 1000*[100 400 150 300 10]';
23
24 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
25 %% Solution with GLPK with outputs
26 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
27
28 p.msglev = 3;
29 p.presol = 0;
30 [x, z, e, xtra] = glpk(c,A,b, [], [], "UUUUU", "CCC", -1, p)
```

Running the m-file muaddib_kh.m will give hte following output:

```
1 >> muaddib_kh
2 Scaling...
3 A: min|aij| =  8.000e+00  max|aij| =  8.000e+02  ratio =  1.000e+02
4 EQ: min|aij| =  1.786e-01  max|aij| =  1.000e+00  ratio =  5.600e+00
5 Constructing initial basis...
6 Size of triangular part is 5
7 GLPK Simplex Optimizer, v4.65
8 5 rows, 3 columns, 15 non-zeros
9 *     0: obj =  -0.000000000e+00 inf =   0.000e+00 (3)
10 *     2: obj =   6.246575342e+04 inf =   0.000e+00 (0)
11 OPTIMAL LP SOLUTION FOUND
12 x =
```

```
13
14 171.2329
15 246.5753
16 0
17
18 z = 6.2466e+04
19 e = 0
20 xtra =
21
22 scalar structure containing the fields:
23
24 lambda =
25
26 0
27 0
28 0
29 0.1945
30 0.4110
31
32 redcosts =
33
34 0
35 0
36 -1.3699
37
38 time = 0
39 status = 5
```

Here the output lines 2–11 are there because we asked glpk to babble. If we would have let glpk to use the presolver these lines would have been slightly different. In thelines 12–18 we have the usual result. The interesting new information is in lines 20–36. There we see two vectors:

- xtra.lambda
- xtra.redcosts

The shadow prices $\boldsymbol{\lambda}$=xtra.lambda are related to the constraints or to the resources $\mathbf{b}$. The shadow price associated with a resource tells you how much more profit you would get by increasing the amount of that resource by one unit. So, since the shadow prices for fat, sugar and protein are all zero, it means that Muad'Dib Bakery does not gain anything for additional fat, sugar or proteig powder. Shadow prices for water and spice Melange are positive. For sexampe spice Melange's shadow price $\lambda_5 = 0.4110$. This means that if someone is willing to sell spice Melange to Muad'Dib Bakery with price less than 0.4110 sol/mg, Muad'Dib Bakery should buy the powder. If the price is more than 0.4110 sol/mg, it does not make sense for Muad'Dib bakery to buy more spice Melange.

The reduced costs $\mathbf{u}$=xtra.redcosts are related to the decision variables, or to the objective coefficients $\mathbf{c}$. The reduced costs and the decision variables are complementary in the sense that if $x_k > 0$ then the associated reduced cost $u_k = 0$, and vice versa. One way of interpreting the reduced cost is to say that it is the amount of how much profit gained from decision variable must increase so that it will become optimal to produce it. So, for Muad'Dib the reduced costs for Atreides and Corrino are both 0, since it is optimal to make them. It is not optimal to make Harkonnen cakes. The reduced cost $u_3 = -1.3699$ tells us that the price of Harkonnen cake must be increase to 100+1.37 = 101.37 before it is optimal to make them.