Tartalomjegyzék

- 1 Octave
  - 1.1 First steps
    - 1.1.1 Getting the program
    - 1.1.2 Calculator
  - 1.2 Data types
    - 1.2.1 Number representation
    - 1.2.2 Matrices
    - 1.2.3 Ranges
    - 1.2.4 typeinfo
  - 1.3 Operations with matrices
    - 1.3.1 Transpose
    - 1.3.2 Addition
    - 1.3.3 Multiplication
    - 1.3.4 Every element or the whole
  - 1.4 Variables
  - 1.5 Indexes
  - 1.6 Vectorization
  - 1.7 Functions
  - 1.8 Tasks
    - 1.8.1 What is this?
    - 1.8.2 SOE
    - 1.8.3 More SOE
    - 1.8.4 Large matrix
    - 1.8.5 Function on matrices
    - 1.8.6 Part of a matrix
    - 1.8.7 Part of a matrix function
    - 1.8.8 Every second column
    - 1.8.9 Applying a function to elements
    - 1.8.10 Help yourself

◊ 1.8.11 Numerical differentiation

Octave

Octave is a numerical calculator, its the free (opensource) version of MatLab.

First steps

Getting the program

Use one of the following from home:

- install Octave, free for all platforms
- using `putty` log in to `leibniz`, start `octave` with the `octave` command
- run octave from the computer lab

Calculator

Octave can be used as an advanced calculator. Try the following (after entering octave):

```
2 + 3
```

press `Enter`:

```
> 2 + 3
ans =  5
```

Try these as well:

```
2 - 3
2 * 3
2 / 3
floor (2 / 3)
mod (2, 3)
2^3
sqrt (2)
log (2)
log (3)
log (8) / log (2)
exp (1)
pi
cos (pi / 2)
(180 / pi) * acos (0.5)
```

This is how you quit

```
exit
```

Data types

Every number is a floating point number, even if it happens to be an integer:

```
1000 / 9
```

```
ans = 111.11
```

We can however force it to use integers:

```
int32 (1000) / int32 (9)
ans = 111
```

In Octave every number is real as long as it does not turn out to be complex:

```
sqrt (2)
sqrt (-2)
```

Number representation

- `double`: double precision float, 64 bit (8 byte)
    - ♦ real: 8 byte
    - ♦ complex: 16 byte
- `single`: single precision float, 32 bit (4 byte)
    - ♦ real: 4 byte
    - ♦ complex: 8 byte
- `int32`: 32 bites two's complement integer (4 byte)
- `int8`: 8 bites two's complement integer -128..127 (1 byte)
- `uint32`: 32 bites unsigned integer (4 byte)
- `uint8`: 8 bites unsigned integer: 0..255 (1 byte)

Size does matter:

```
log(single(1.0001))
log(double(1.0001))
int32(100+100)
int8(100+100)
```

Matrices

In `octave` **every number is a matrix**

- numbers: `1x1`
- vectors:
    - ♦ row vector: `1xn`
    - ♦ column vector: `nx1`
- matrix: `nxm`

About.

Row vector:

```
[1 2 3 4]
[1, 2, 3, 4]
```

Column vector:

```
[1; 2; 3; 4]
```

This is not a column vector:

```
[[1], [2], [3], [4]]
```

Matrix:

```
[1 2; 3 4]
[1, 2; 3, 4]
```

Special matrices:

- `zeros`: all 0
- `ones`: all 1
- `eye`: 1 in diagonal, 0 elsewhere
- `diag`: square diagonal, with the speciafied diagonal

```
zeros (2, 3)
eye (2, 3)
ones (3, 1)
diag ([1, 2, 3, 4])
```

Also try:

```
size (5)
size ([1, 2, 3])
size ([1; 2; 3])
```

Ranges

Try the following:

```
1:10
```

We can specify a step as well:

```
1:0.1:2
1:2:10
```

Does not work for complex numbers, since there's no order on complex numbers!
The result will always be `double` but we can convert it:

```
int32 (1:0.5:10)
```

Decreasing range:

```
4:-1:1
```

Empty range:

```
4:1:1
4:1
```

Diagonal matrix:

```
> diag(1:4)
ans =
  1 0 0 0
  0 2 0 0
  0 0 3 0
```

```
  0 0 0 4
```

typeinfo

You can find out the type of an item with `typeinfo`.

```
typeinfo (1)
typeinfo (int32(1))
typeinfo (i)
typeinfo ([1 2 3 4])
typeinfo ([1 i -1 -i])
typeinfo (1:4)
typeinfo ([1:4])
typeinfo (1 >= 0)
typeinfo ("http://wiki.math.bme.husometext"http://wiki.math.bme.hu)
typeinfo ([1 + 2i 1 - 1i])
```

Operations with matrices

Since every number is a `1x1` matrix, this can be applied to numbers as well.

Transpose

Use an **accent** (`'`):

```
> [1 2; 3 4]'
ans =
 1 3
 2 4
> _
```

Or

```
> (1:4)'
ans =
  1
  2
  3
  4
```

For complex matrices the accent means andjugate

```
in other words, the conjugate of the transpose:
> [1 2i; 3i 4]'
ans =
  1 - 0i   0 - 3i
  0 - 2i   4 - 0i
```

Conjugate: `(2 + 1i)'`

Addition

```
1 + (1:4)
eye (2, 2) + ones (2, 2)
[1; 2; 3; 4] - [4; 3; 2; 1]
```

Multiplication

Every multiplication is matrix multiplication:

```
> [1 2; 3 4]*[1 2; 3 4]
ans =
   7   10
  15   22
```

Power as well, so is the inverse:

```
[1 2; 3 4]^2
[1 2; 3 4]^-1
```

For multiplications the dimensions must be compatible:

```
ones (2, 3) * ones (3, 5)
```

Multiplying a row vector and a column vector results in a scalar, while the reverse is a dyiadic multiplication:

```
[1,2,3]*[1;2;3]
[1;2;3]*[1,2,3]
```

Every element or the whole

If power is a matrix operator, then what is this?

```
[1 2; 3 4]^0.5
```

And this?

```
sqrt([1 2; 3 4])
```

Some operators apply to every element of a matrix, while others apply to the matrix itself.

```
> (1:4)^2
error: for A^b, A must be a square matrix
```

This results in an error since multiplying a `1x4` matrix with itself is not possible. But:

```
> (1:4).^2
ans =
  1  4  9 16
```

Putting a dot in front of an operator makes it apply to every element of a matrix instead of apply to the matrix itself. For addition this does not matter.

```
> [1 2; 3 4] + [1 2; 3 4]
ans =
  2   4
  6   8
> [1 2; 3 4] .+ [1 2; 3 4]
ans =
  2   4
  6   8
```

But for multiplication it matters:

```
> [1 2; 3 4] * [1 2; 3 4]
ans =
   7   10
  15   22
> [1 2; 3 4] .* [1 2; 3 4]
ans =
   1    4
   9   16
```

Power:

```
> [1 2; 3 4]^-1
ans =
 -2.00000   1.00000
  1.50000  -0.50000
> [1 2; 3 4].^-1
ans =
  1.00000   0.50000
  0.33333   0.25000
```

Elementry functions apply to every element usually:

```
sin (0:0.1:2*pi)
exp ([0,-1;1,0])
```

While elementry operators apply to the matrix (`*, ^, /, \`)

Variables

We use *variables* to store values.

```
a = 2
b = 3
a + b
```

The variable `ans` always exists. It stores the last calculated value.
If a variable doesn't exist we can't use it in a calculation:

```
> a / q
error: `q' undefined
```

Using a semicolon (`;`) we can do silent calculations:

```
a = 2;
b = 3;
a + b
```

The `whos` commands shows the currently stored variables.

```
> whos
Variables in the current scope:
  Attr Name        Size                     Bytes  Class
  ==== ====        ====                     =====  =====
       a           1x1                          8  double
       ans         1x1                          8  double
       b           1x1                          8  double
Total is 3 elements using 24 bytes
```

Variables can be overwritten anytime:

```
> a = 2;
> a = [1 2; 3 4];
> whos
Variables in the current scope:
  Attr Name        Size                     Bytes  Class
  ==== ====        ====                     =====  =====
       a           2x2                         32  double
```

Indexes

Let `M` be a `3x3` matrix. The *j*th element of the *i*th row of this matrix is:

```
M = rand(3,3);
i = 1;
j = 3;
M(i,j)
```

Concatenating matrices:

```
[M M]
[M; M]
```

Partial series:

```
l = 0:0.1:1;
l (:)
l (1:11)
l (1:5)
l (5:end)
l (1:2:11)
```

Also:

```
l (1:2:11) = 0
```

Part of a matrix:

```
A = [1 2 3; 4 5 6; 7 8 9];
A (1:3, 1:2)
A (1:2, 1:3)
```

Skipping one line:

```
A ([1, 3], :)
```

Or selecting a specific part of a matrix:

```
S = ones (8, 8);
S (3:6, 3:6) = -1
```

Or a given index:

```
S = ones (8, 8);
S ([1 2 8], [2 6]) = 8
```

Flattening a matrix:

```
A = eye (3, 3);
A (:)
```

Vectorization

In Octave we usually work with multiple numbers not just one. For example let us calculate the norm of every row in the matrix `X`:

```
X = rand (10, 3);
sqrt (sum (X.^2, 2))
ans =
  0.99105
  0.86977
  1.29362
  0.91697
  1.26149
  0.84024
  1.45410
  1.19791
  1.01153
  1.07420
```

Let us look at the functions:

- `X.^2`: calculates the square of the items
- `sum( , 2)`: sums up the values into a column vector (the meaning of the 2 is to sum up along the second dimension)
- `sqrt`: square root of every item

Calculate the function value `2x^2-3x+1` where `x` is a row vector:

```
x=0:0.1:1;
fx=2.*x.^2 - 3.*x + 1
```

Functions

Try the following (multiple lines)!

```
> function fx = f (x)
>   fx = 1 / (x^2 + 1);
> endfunction
```

Test it:

```
> f(3)
ans = 0.10000
```

Defining functions:

```
function <<result>> = <<function name>> (<<variables>>)
  ...
endfunction
```

Any command can be inside the function, but at the end the `<<result>>` variable has to contain the result of the function. We usually use silent calculation (`;`) inside the function.

Another function:

```
function R = remove_last (x)
  R = x (1:end-1);
endfunction
```

Example:

```
> remove_last (1:5)
ans =
  1   2   3   4
```

Tasks

What is this?

What does the following do?

```
A=[1 2 3; 4 5 6; 7 8 9];
B=[9 8 7; 6 5 4; 3 2 1];
trace (A*B')
A(:)' * B(:)
```

What is `trace(A*B')`?

SOE

Calculate the solution of the following system of equations:

```
 x + 2y = 3
4x + 5y = 6
```

Solution 1:

```
A = [1 2; 4 5]
b = [3; 6]
x = A^-1 * b
```

Solution 2:

```
x = A \ b
```

More SOE

Solve the following systems of equations:

```
 x + 5y = 1
2x + 4y = 2
```

```
 x + 5y = 1
2x + 4y = 2
5x - 6y = -1
```

```
 x + 2y + 5z = 1
5x + 4y + 6z = 2
```

Large matrix

- Create the following matrix using the least characters you can:

```
1 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
2 2 3 2 2 2 2 2 2 2
2 2 2 4 2 2 2 2 2 2
2 2 2 2 5 2 2 2 2 2
2 2 2 2 2 6 2 2 2 2
2 2 2 2 2 2 7 2 2 2
2 2 2 2 2 2 2 8 2 2
2 2 2 2 2 2 2 2 9 2
2 2 2 2 2 2 2 2 2 10
```

- Same as before:

```
 0  1  0  0  0  0  0  0  0
-1  0  1  0  0  0  0  0  0
 0 -1  0  1  0  0  0  0  0
 0  0 -1  0  1  0  0  0  0
 0  0  0 -1  0  1  0  0  0
 0  0  0  0 -1  0  1  0  0
 0  0  0  0  0 -1  0  1  0
 0  0  0  0  0  0 -1  0  1
 0  0  0  0  0  0  0 -1  0
```

- Chessboard

```
  1  -1   1  -1   1
 -1   1  -1   1  -1
  1  -1   1  -1   1
 -1   1  -1   1  -1
  1  -1   1  -1   1
```

Function on matrices

Write a function that applies the function $2\sin^2 x + 1$ to every element of a given matrix.

Part of a matrix

Write a function that separates a 5x5 matrix into a 2x5 matrix with the 2nd and 4th row, and a 3x5 with the 1st, 3rd, 5th.

Part of a matrix function

Combine the previous two so that the function applies the $2\sin^2 x + 1$ function to both new matrices.

Every second column

Write a function that outputs the matrix with every second column skipped in the input matrix. (Help, `size` can help with the dimension of the matrix.)

Applying a function to elements

Write a function that applies the function $2\sin^2 x + 1$ to every second column of a matrix.

Help yourself

The octave documentation, stackoverflow or Google.

Numerical differentiation

Differentiate the function `f(x)=2x^2-3x+1` numerically! Given a row vector `x`, and `fx` with the corresponding function values.

```
x = 0:0.1:1
fx = 2.*x.^2 - 3.*x + 1
```

The numerical differentiation:

```
df = (fx(2:end) - fx(1:end-1)) ./ 0.1
```

With non-linear steps:

```
df = (fx(2:end) - fx(1:end-1)) ./ (x(2:end) - x(1:end-1))
```