

Getting Started

Summer School on Fire Dynamics Modeling 2017

Lukas Arnold

Contents:

1. Virtual Machine (VM)
2. FDS
3. Linux
4. Python

1. Virtual Machine (VM)

1.1 Basics

1.2 Guest System

1. Virtual Machine (VM)

1.1 Basics

1.2 Guest System

Virtual machines

Virtual machines (VM) emulate a computer and allow to run a guest operating system (OS) within a host system.

With VMs it is possible to have multiple instances of an OS running, which may for example have individual and sole tasks.

We have prepared a VM with a Linux OS, here Manjaro Linux, with all software needed for the summer school.

The image is available on USB drives (contact Ashish) and as a download link (see emails).

Virtual Box



- ▶ Virtual Box is a free and open source software to create, manage and run VMs.

→ <https://www.virtualbox.org>

- ▶ The provided VM image can be imported via File / Import Appliance.

VM notes and hints

- ▶ Switch mouse / keyboard focus between host and guest via the 'Host key', see right bottom of a running VM
- ▶ A shared clipboard can be setup Devices / Shared Clipboard
- ▶ Snapshots allow to switch (while off) between states of the VM, create an initial snapshot to be able to come back to the initial state

1. Virtual Machine (VM)

1.1 Basics

1.2 Guest System

Manjaro Linux



→ <https://manjaro.org>

- ▶ User name: `summer`
 - ▶ Password: `school`, only needed for new software installation
- Start VM

firesss_* scripts

The VM comes with a few handy scripts to ease your setup:

- ▶ `firesss_udpate_bin`: updates the school's scripts
- ▶ `firesss_help`: prints a short help message with a few usefull lines to `grep`
- ▶ `firesss_preparaessh`: sets up you ssh environment to login to JURECA
- ▶ `firesss_getmaterial`: downloads teaching material

School material

Using the VM, we have prepared the `firesss_getmaterial` command, which grabs the current version of the selected material and puts it into a new directory.

→ Demo on VM

Usage of `firesss_getmaterial`:

```
> firesss_getmaterial -h
Usage: firesss_getmaterial <lecture number: 00 ... 09>
  -v, --verbose      : forward all output to console
  -a, --all          : get all lectures
```

Software

- ▶ Browser: Firefox
- ▶ Text editors: mousepad, emacs, gedit
- ▶ Python: version 3.6, including all needed modules
- ▶ FDS+SVM: recent repository clone, aliases set to `fds` and `smv`
- ▶ `git` + `gcc`: to grab and compile FDS from the GitLab repository

2. FDS

2.1 Binary Installation

2.2 Installation from source

2. FDS

2.1 Binary Installation

2.2 Installation from source

Binary downloads at FDS-SMV webpage

Binary files for common OS are available here:

→ <https://pages.nist.gov/fds-smv/downloads.html>

2. FDS

2.1 Binary Installation

2.2 Installation from source

Get the source code

The source code of FDS is currently hosted by GitHub:

→ <https://github.com/firemodels/fds>

It can be either

- ▶ downloaded as an archive file
- ▶ or cloned via git

To access the GitHub repositories, there exists a GUI application 'GitHub Desktop':

→ <https://desktop.github.com>

Note: We can offer git support / first steps on an individual basis during the week. Just contact me.

Compilation and linkage

In order to execute the source code, it needs to be translated to the target CPU machine language. This is done by a compiler. The interaction with the operating system (OS) and third party libraries is put together with the linker.

A common choice is the GNU compiler collection (GCC), which covers many mainstream languages, including FORTRAN.

GCC is directly available on Linux and macOS systems, or can be added via the package management software.

Demo on the VM

```
cd
mkdir build_fds_demo
cd build_fds_demo

git clone https://github.com/firemodels/fds.git .
cd Build/gnu_linux_64
sh make_fds.sh

./fds_gnu_linux_64
```

Note: to make this executable globally available either the PATH variable must be extended accordingly or an alias should be set.

Compilation with MPI support

1. Install OpenMPI via the software manager
2. Change to the build directory `mpi_gnu_linux_64`
3. Invoke `sh make_fds.sh`
4. Execute with the `mpirun` command

3. Linux

3.1 General

3.2 Command line

3. Linux

3.1 General

3.2 Command line

Basic idea of UNIX / Linux

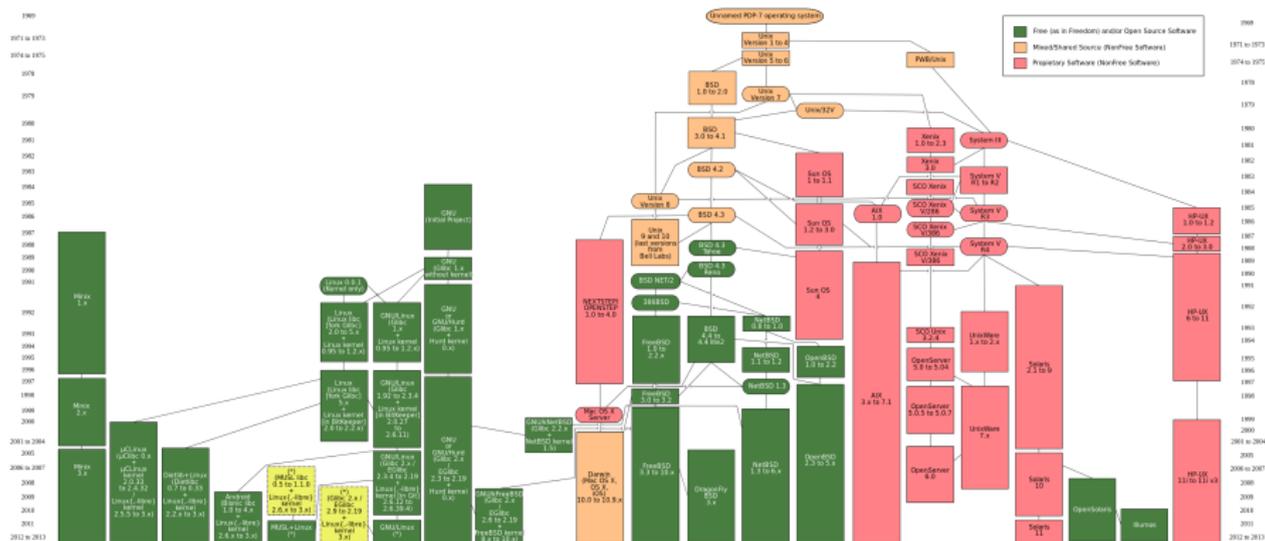
The UNIX based operating systems (OS) have the following philosophy:

1. write computer programs, that solve a single task, but very efficient
2. write computer programs, such that they can work together
3. write computer programs, that process simple text streams

Therefore the whole OS is a collection of a zoo of small specialised applications. The combination of all of them creates a complex and mighty system.

The base system of an UNIX-like OS can be used with text based terminals. However, most systems have a graphical system on top.

History of UNIX / Linux



3. Linux

3.1 General

3.2 Command line

Introduction Unix shell

A Unix shell, or just shell, is a user interface to the UNIX OS. It is already a highly abstracted layer to hide the kernel complexity.

The most popular shells are:

- ▶ C shell (csh)
- ▶ Bourne shell (sh)
- ▶ Bourne-Again shell (bash)

Note: In the following we will focus on the bash shell.

Note: "Build-in" documentation to all relevant commands: `man command`.

Note: To open a terminal in the file manager, use the context menu.

Environment variables

All shells provide variables to setup the execution environment. They are read out as **\$AVAR** and set as **AVAR**:

```
1 echo $USER
2 export OMP_NUM_THREADS=8
```

Some important environment variables are:

- ▶ **PATH**, list of directories
- ▶ **USER**, user name
- ▶ **PWD**, current directory
- ▶ **HOME**, user's home directory

The command **env** lists all set environment variables.

Command execution

A command or program is executed by typing its name:

```
evn
```

This must be either a build-in command, a program found in the PATH environment or a direct reference to an executable file.

To execute a custom program in the current directory (not in PATH):

```
./a.out
```

Commands may be executed in background by appending a `&` at the end:

```
./a.out &
```

The command `ps` prints all running processes in current terminal and `ps aux` shows all processes on the system.

Wildcards

Wildcards are used to provide a matching pattern for the shell. The shell will evaluate it and explicitly list the result.

Some common wildcards are:

- ▶ *****: match everything
- ▶ **?**: match any single character
- ▶ **[list]**: match any single character from list, here: l, i, s, t

Wildcards can be combined with constant strings and with each other.

List all files ending with `.pdf`:

```
ls *.pdf
```

Pipes and redirection

The output of programs is either to

- ▶ stdout: the normal program output
- ▶ stderr: error messages

In many cases it is useful to write the output into a file. This can be done via redirections

- ▶ `>`: creates a new output file, overwrites old one
- ▶ `>>`: appends the output

Print all environment variables into a file:

```
env > env.log
```

To forward the output of an command to be the input of an other, pipes `|` are used:

```
find . -mtime -15m -type f | grep -i info | wc -l
```

Command line

At the command line the user issues commands and executes programs.

Some features:

- ▶ use arrow keys to move in history
- ▶ `ctrl+a` / `ctrl+e`, move to begin / end of line
- ▶ `ctrl+k` / `ctrl+u`, delete from cursor to end / begin of line
- ▶ `alt+b` / `alt+f`, move backward / forward a word
- ▶ `ctrl+r`, search in history
- ▶ `ctrl+c`, terminate command
- ▶ `ctrl+z`, suspend command
- ▶ `bg` / `fg`, put suspended job to background / foreground

Changing directories

The current directory is changed via the `cd` command.

Invoking no arguments changes to the user's home directory, otherwise the target directory is specified.

There exist a couple of special directories (also handy for other commands):

- ▶ `~`: the user's home directory
- ▶ `~username`: the home directory of user `username`
- ▶ `.`: this directory
- ▶ `..`: above directory (root direction)
- ▶ `-`: last directory (`cd` command only)

Listing directory contents

The contents of a directory are displayed with `ls`. Without arguments it shows the current directory, otherwise the target directory.

Some common options:

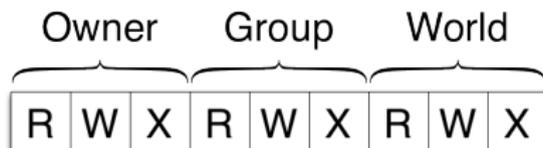
- ▶ `--color=always`, color output
- ▶ `-l`, long output, i.e. list permissions, data, size
- ▶ `-a`, list all files, i.e. also hidden (starting with a `.`)
- ▶ `-h`, show sizes human readable

List all files and sort w.r.t. the modification time:

```
ls -larth
```

Listing and changing access permissions

The access permissions can be shown with the `ls` command. The first character in the `ls` output is the type. The syntax of the following characters is as follows:



To change the permissions the command `chmod` is used. The first argument is a combination of permission target, permission modification and new permission:

- ▶ target, u: user, g: group, o: other
- ▶ modifier, +: add, -: remove
- ▶ permission, r: read, w: write, x: execute

To change the permissions in all subdirectories use the `-R`.

Listing files

There exist a couple of ways to list files:

- ▶ **cat**: prints file content to terminal
- ▶ **less**: print content and allows to scroll and search, quit with q
- ▶ **head**: prints the first lines of a file
- ▶ **tail**: prints the last lines, option **-f** follows the file change

Note: all of the above commands are highly configurable.

Copy and link files

A simple file copy is done with `cp` and may be executed recursively `-r`.

There exist two types of links that can be used:

- ▶ hard links `ln`
- ▶ soft links: `ln -s`

Find files

Searching files is done with the `find` command. It has a lot of options, some common scenarios are:

- ▶ find all files ending with `stat` in home directory:

```
find $HOME -name *stat
```

- ▶ find all files that have been modified in the last 24 hours (in home directory):

```
find $HOME -mtime 0
```

Find content in files

To search file's content, the `grep` command is used.

The first argument is the search string the second the files to be searched in.
Some common options:

- ▶ `-r`: recursive search
- ▶ `-i`: case insensitive search

The following example searches for the string `fire` in all files:

```
grep -i fire *
```

Pattern based search and replace

A common task is to search and replace the content of a file. This can be done with the `sed` command.

It takes regular expressions as arguments and processes a file.

To change the string `lukas` to `matthias` in a file at all occurrences:

```
sed 's/lukas/matthias/g' in.file
```

Creating tar-balls

To create a archive of files, the **tar** command is a common tool.

The major options are:

- ▶ **-c** / **-x**, create / extract mode
- ▶ **-f**, specify output / input file
- ▶ **-z**, compress mode
- ▶ **-t**, list contents

Create a tar-ball out of a directory

```
tar -cf dir.tar dir/
```

Extract a compressed tar-ball

```
tar -xzf files.tgz
```

Quick overview – vi

A common text based editor is **vi**. Here is a quick introduction to do simple editing:

1. open the file with `vi file.dat`
2. enter edit mode by pressing `i`
3. do the editing
4. press `esc` to end edit mode
5. press `:w` to save the file
6. press `:q` to quit
7. to quit without saving, press `:q!`

Quick overview – emacs

An other popular and also very powerful editor is **emacs**. A quick introduction:

1. open a file with `emacs file.dat`
2. start editing
3. save file by pressing `ctrl-x ctrl-s`
4. exit email `ctrl-x ctrl-c`
5. to abort a command `ctrl-g`

Connecting to remote servers via ssh

To connect to modern supercomputer, you have to use `ssh` with a public-private key encryption.

The connection syntax is as follows

```
ssh user@server
```

The configuration files and public-private keys are stored in `~/.ssh`

To ease a frequently used connection, the config file may be customised, e.g.

```
1 Host jureca
2 Hostname jureca.fz-juelich.de
3 User train115
4 IdentityFile ~/.ssh/id_train115
```

Remote file copy (in both directions) is done with `scp`:

```
scp file.dat user@server:~
```

Access the internet

To download files or "access" websites the `wget` command may be used.

```
wget URI
```

Shell configuration files

To setup an environment at every shell startup, configuration files (`.bashrc`, `.bash_profile`, `.profile`) may be used. All the commands listed in the config files are executed.

Common examples:

- ▶ set aliases: `alias lt='ls -larth'`

- ▶ export variables:

```
export FZJSVN='https://svn.version.fz-juelich.de'
```

- ▶ extend path variables: `export PATH=$HOME/Software/bin:$PATH`

- ▶ read in other configuration files: `source ~train199/setup_firefoam.sh`

4. Python

4.1 General

4.2 Nutshell

4.3 Examples

4. Python

4.1 General

4.2 Nutshell

4.3 Examples

Read-only Python

This summer school is not a programming lecture. However, for scientific data visualization and analysis, as well as for model development, programming skills are handy.

Therefore we offer you the basics to be able to **read** Python scripts and understand how they work. You can do all exercises without writing code by yourself, but you can use this school as a chance to start doing so.

If you see potential for your work, there is plenty of material on Python programming in the world wide web.

Python

In contrast to computer programs, Python scripts are not executed, but interpreted by a Python interpreter.

It is widely spread because

- ▶ It is very simple to learn
- ▶ There exist a huge amount of modules to be easily used
- ▶ It is free
- ▶ It is OS independent, interpreter are available for most common OS, including HPC systems

To 'run' a Python script, pass it as an argument to the Python executable:

```
> python my_script.py
```

Although, there exist many useful IDEs, we keep it simple during the school and use basic text editing software.

4. Python

4.1 General

4.2 Nutshell

4.3 Examples

Syntax

The Python language is very simple and contains only few keywords, like:

`for`, `if`, `def`, `import`, `return`, `False`, `True`, `and`, `or`, `not`, ...

Execution blocks, so called scopes, are defined by indentation, in contrast to other languages, that use key-words or key-characters. Besides indentation, spacing has no meaning.

```
1 i = 5
2 b = i * 3
3 for it in range(10):
4     print(it)
5     if it < 5:
6         print("too small")
7         do_something(it)
8         b = b + it**2
9 print(b)
```

Comments use the # character, where all following characters in the line are not evaluated, but handled as a comment.

Common keywords and operators

- ▶ value assignment: =
- ▶ comparison operators: >, <=, ==
- ▶ arithmetic operators: e.g. +, *, **, %
- ▶ logical operators and values: and, or, not, True, False
- ▶ for, while: define loops
- ▶ if: flow control
- ▶ import, from: module loading
- ▶ def: function definition

Functions

Functions allow to encapsulate frequent and common tasks. They are called by their name, followed by `()`, that may contain function arguments. An optional return value may be passed by a function.

```
b = max(5,7)
```

Here, 5 and 7 are arguments to the function `max`, which assigns the return value to the variable `b`.

Variables

In Python, the types of variables do not have to be declared first, but they adopt to the types they are assigned to. Assignment is done via =, e.g.:

```
1 a = 5
2 b = "summer school"
3 a = [5, 8, "fire"]
```

Basic value types are:

- ▶ integer: 1,3,5,-78
- ▶ floats: 3.142, 42.1, 1e-7
- ▶ strings: "juelich", 'aachen'
- ▶ lists: [7, 3, 6.7, 'koeln']

The values of variables can be printed to the command line with the `print` function.

Accessing lists and arrays

Lists and (numpy) arrays are accessed via an index:

```
1 mylist = [7,9,1]
2 mylist[0] = 5
3 myarray = np.zeros(7)
4 myarray[2:5] = 1.0
5 b = myarray[3]
```

where the first entry has the index 0!

While lists can be addressed only with a single index, arrays accept ranges of indices.

if-statements

To control the execution flow, if-statements are used. Here the general syntax is as follows:

```
1 if condition:
2     true-block
3 else:
4     false-block
```

Or as an example:

```
1 if 5 > 8:
2     print("that's wrong")
3 else:
4     print("true")
```

If not needed, the else block can be omitted.

for-loops

Looping over a set (e.g. a list) of values is done with a the `for` statement:

```
1 for i in set:  
2     for-block
```

Or, to print the squares of the numbers 0 to 9:

```
1 for i in range(10):  
2     print(i**2)
```

where the `range(n)` function returns a list starting by 0 to $n-1$.

4. Python

4.1 General

4.2 Nutshell

4.3 Examples

Example 01 – Variables and arithmetic operations

01_variables.py

```
1 # simple assignment
2 a = 70
3 b = 70*4.5
4 c = b ** 0.5
5
6 # print unformatted
7 print(a, b, c)
8
9 # formatted print
10 print("values of a: {}, b: {}, c: {}".format(a,b,c))
11
12 # create empty list, via [], append values to it
13 alist = []
14 alist.append(a)
15 alist.append(4.5)
16 alist.append(10)
17
18 # read and assign list's elements
19 alist[2] = alist[0] * 3
20
21 # print the list
22 print(alist)
```

Example 01 – Variables and arithmetic operations – results

```
1 (70, 315.0, 17.74823934929885)
2 values of a: 70, b: 315.0, c: 17.7482393493
3 [70, 4.5, 210]
```

Example 02 – Flow control

02_flow_control.py

```
1 n = 30
2 ix = range(n+1)
3 print("ix: ", ix)
4
5 my_sum = 0
6 for i in ix:
7     x_new = (2.0 * i / n) - 1.0
8     print("i: {:3d} -> x_new: {:.4f}".format(i, x_new))
9
10    if x_new >= 0:
11        my_sum += x_new ** 2
12    else:
13        my_sum += x_new
14
15 print("my_sum: ", my_sum)
```

Example 02 – Flow control – results

```
1 ('ix: ', [0, 1, 2, 3, 4, 5, 6, 7, 8, ..., 26, 27, 28, 29, 30])
2 i: 0 -> x_new: -1.0000
3 i: 1 -> x_new: -0.9333
4 i: 2 -> x_new: -0.8667
5 i: 3 -> x_new: -0.8000
6 i: 4 -> x_new: -0.7333
7 i: 5 -> x_new: -0.6667
8 i: 6 -> x_new: -0.6000
9 i: 7 -> x_new: -0.5333
10 i: 8 -> x_new: -0.4667
11 i: 9 -> x_new: -0.4000
12 i: 10 -> x_new: -0.3333
13
14 [...]
15
16 i: 27 -> x_new: +0.8000
17 i: 28 -> x_new: +0.8667
18 i: 29 -> x_new: +0.9333
19 i: 30 -> x_new: +1.0000
20 ('my_sum: ', -2.4888888888888888)
```

Example 03 – Arrays

03_arrays.py

```
1 import numpy as np
2
3 n = 10
4 L = 3.0
5
6 x = np.linspace(-L, L, n)
7 y = x**2
8
9 print("x: {}".format(x))
10 print("y: {}\n".format(y))
11
12 print("x[2:4]: {}".format(x[2:4]))
13 print("x[-2:]: {}\n".format(x[-2:]))
14
15 d = y[1:] - y[:-1]
16 print("d: {}".format(d))
17 print("len(y): {}, len(d): {}\n".format(len(y), len(d)))
18
19 print("|d| > 1: {}".format(d[np.abs(d) > 1]))
20 print("|d| > 1: {}\n".format(np.where(np.abs(d) > 1)[0]))
21
22 z = np.zeros((3,3))
23 z[1, 2] = 2.0
24 print("z: \n{}".format(z))
```

Example 03 – Arrays – results

```

1  x: [-3.      -2.33333333 -1.66666667 -1.      -0.33333333  0.33333333
2     1.         1.66666667  2.33333333  3.         ]
3  y: [ 9.         5.44444444  2.77777778  1.         0.11111111  0.11111111
4     1.         2.77777778  5.44444444  9.         ]
5
6  x[2:4]: [-1.66666667 -1.         ]
7  x[-2:]: [ 2.33333333  3.         ]
8
9  d: [ -3.55555556e+00 -2.66666667e+00 -1.77777778e+00 -8.88888889e-01
10     -3.05311332e-16  8.88888889e-01  1.77777778e+00  2.66666667e+00
11     3.55555556e+00]
12 len(y): 10, len(d): 9
13
14 |d| > 1: [-3.55555556 -2.66666667 -1.77777778  1.77777778  2.66666667
15          3.55555556]
16
17 |d| > 1: [0 1 2 6 7 8]
18 z:
19 [[ 0.  0.  0.]
20  [ 0.  0.  2.]
21  [ 0.  0.  0.]]

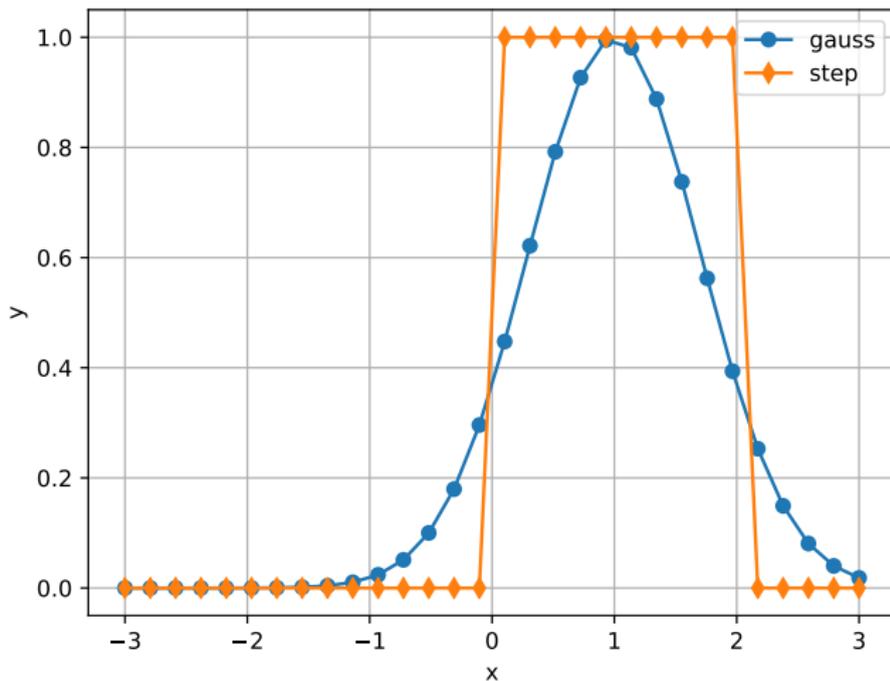
```

Example 04 – Plotting data

04_plotting.py

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 L = 3
5 n = 30
6 x0 = 1.0
7
8 x = np.linspace(-L, L, n)
9 y1 = np.exp(-(x-x0)**2)
10 y2 = np.zeros_like(x)
11 y2[ np.abs(x-x0) < 1.0] = 1.0
12
13 plt.plot(x,y1, label="gauss", marker='o')
14 plt.plot(x,y2, label="step", marker='d')
15
16 plt.xlabel("x")
17 plt.ylabel("y")
18 plt.legend(loc='best')
19 plt.grid()
20 plt.savefig("04_plotting.pdf")
21 plt.show()
```

Example 04 – Plotting data – results



Example 05 – Read and plot FDS device data

1. Change to directory `05_plot_fds_device/data_fds_couch`.
2. Run FDS with the `couch.fds` input file.
3. Go up one directory and run the Python script `print_devc.py`.
4. Have a look at the script.
5. Now, with graphical and file output: run `plot_devc.py`.
6. Check the created image files.
7. Do the same with the script `plot_hrr.py`

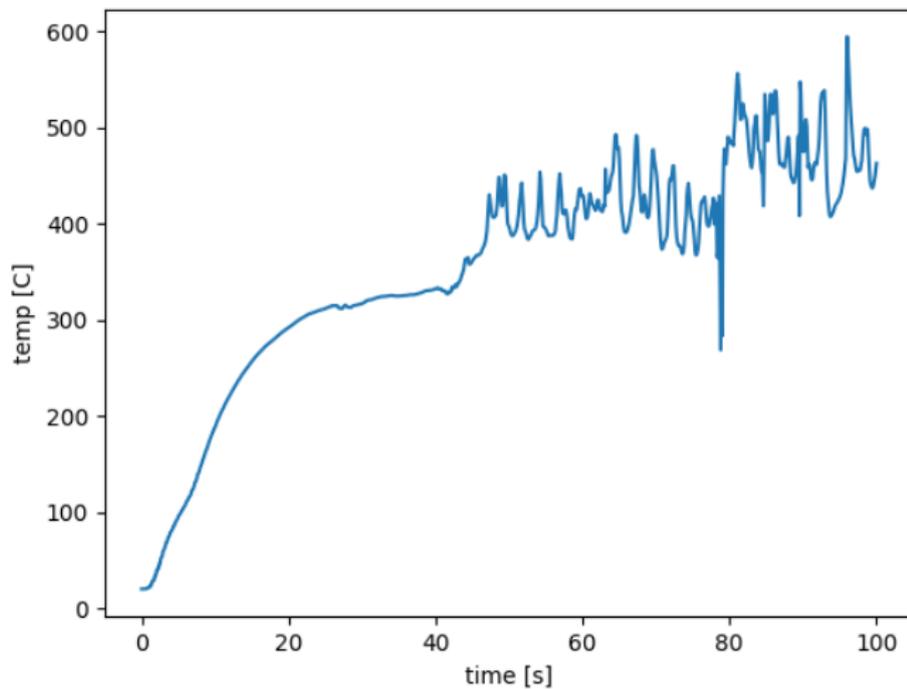
Example 05 – Read and plot FDS device data – results

```

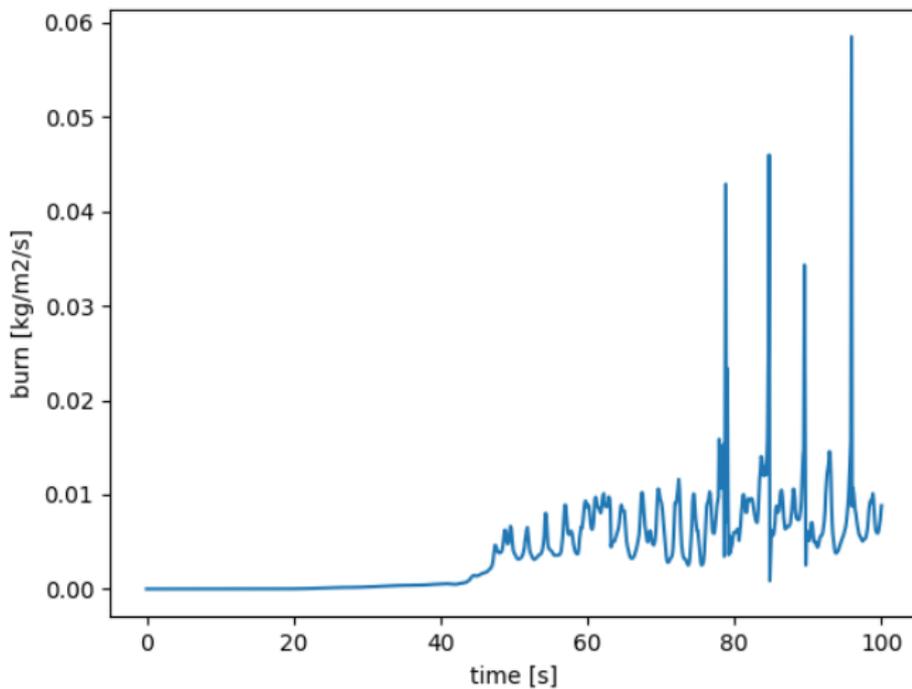
1  -- found quantities:
2  - quantity Time with units s
3  - quantity temp with units C
4  - quantity burn with units kg/m2/s
5  - quantity rad with units kW/m2
6  - quantity gauge with units kW/m2
7  - quantity con with units kW/m2
8  - quantity gas with units C
9  - quantity hrrpuv with units kW/m3
10 - quantity qr with units kW/m3
11 - quantity U with units kW/m2
12 -- first 10 time stamp values:
13 [ 0.          0.10206207  0.20412415  0.30618622  0.40824829  0.51031036
14  0.61237244  0.71443451  0.81649658  0.91855865]
15 -- first 10 values of device temp in C:
16 [ 20.          20.          20.105471  20.105471  20.154548  20.154548
17  20.406798  20.406798  21.338868  21.338868]
18 -- first 10 values of device burn in kg/m2/s:
19 [ 0.00000000e+00  0.00000000e+00  2.89327670e-20  2.89327670e-20
20  2.89372510e-20  2.89372510e-20  2.89462290e-20  2.89462290e-20
21  2.89646120e-20  2.89646120e-20]

```

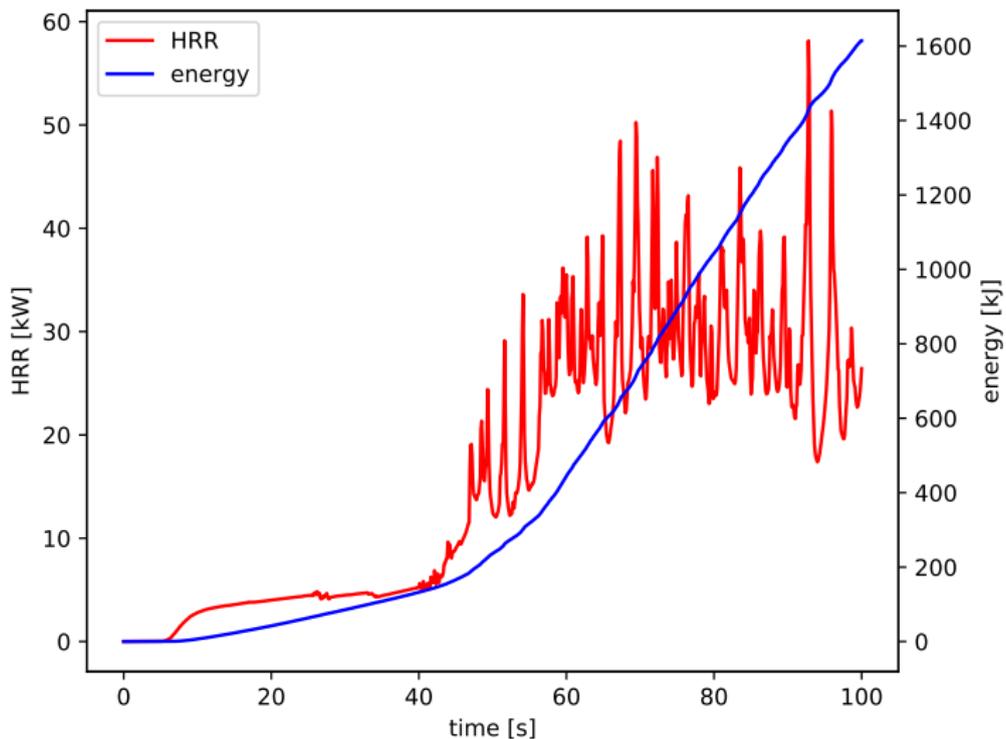
Example 05 – Read and plot FDS device data – results



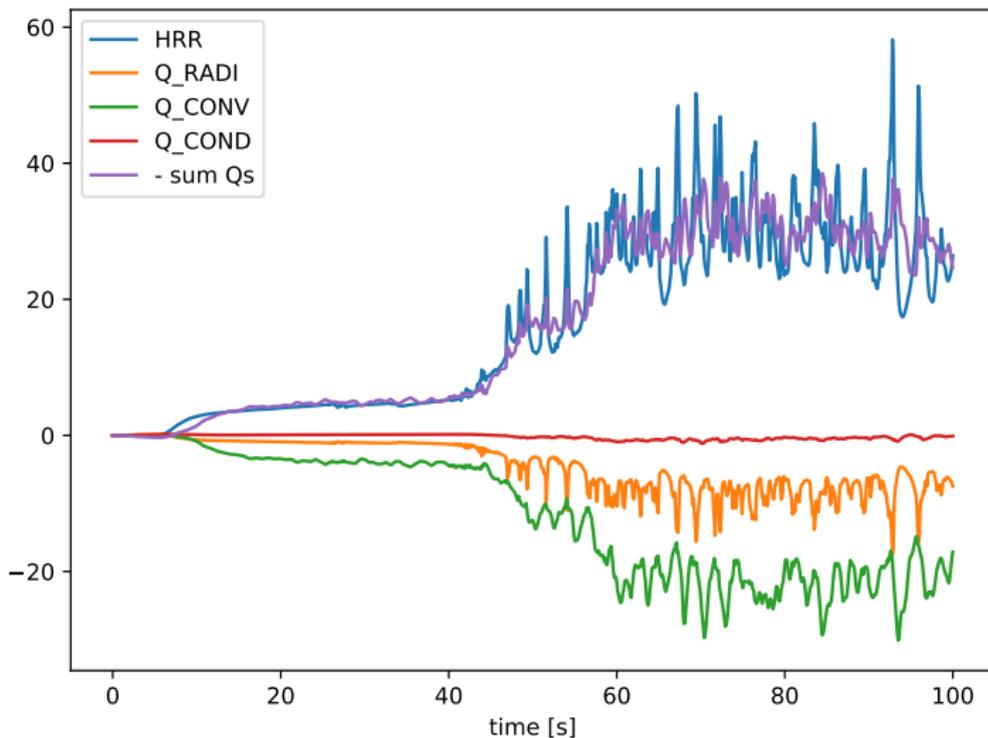
Example 05 – Read and plot FDS device data – results



Example 05 – Read and plot FDS device data – results



Example 05 – Read and plot FDS device data – results



Example 05 – Read and plot FDS device data – results

