

DAVIS

Data Visualization and Integration Shell



Project for the course
10904-01 - Operating Systems by
Lars Schneider & Mirco Franco



Department of Computer Science
University Basel
Switzerland
Spring semester 2024

Contents

1	Introduction	2
2	Background	2
3	Functionalities and Built-ins	3
3.1	ls	3
3.2	clear	5
3.3	hist	5
3.4	wordle	7
3.5	echo	8
3.6	plot	9
3.6.1	Flags	10
3.6.2	Arguments	12
3.7	latex	15
3.8	help	15
4	Methodology & Implementation	16
4.1	Command parser & executor	16
4.2	History	17
4.3	Plotting and Latex	18
4.3.1	plot	18
4.3.2	latex	18
5	Problems & Solutions	19
5.1	Keystrokes	19
5.2	Robustness and Security	19
5.3	Gnuplot	19
6	Conclusion	20
6.1	Lessons learned	20
7	References	20
8	Appendix	21
8.1	Declaration of Independent Authorship	21

1 Introduction

For our project we wanted to make use of the utilities of a command line interface ("CLI" for short). Command line interfaces are powerful tools and offer many functionalities. This is exactly why we decided to implement our own shell to make use of these utilities. A shell is a command-line interpreter that provides a command-line interface for users to interact with their operating system. We set our goal to create a practical and user-friendly shell, as in our opinion, user-friendliness is rarely given. To do so, we implemented some basic shell commands (see 3). In addition we wanted our CLI to partly cover the topic of data visualization. The idea was to create a graph using gnuplot. Gnuplot is a C-program allowing graphical representations of data. The shell should then be able to integrate the visualized data into a \LaTeX template. Thus the ever so creative name: **DA**tA **V**isualization and **I**ntegration **S**hell.

2 Background

A **shell** allows users to execute commands, run scripts, and perform various tasks through textual commands. Common Unix-based operating systems such as Linux, macOS and many others come with a default shell installed. The default shell can vary between systems, but popular choices include the Bourne Shell (sh), the Bourne Again Shell (bash) or the Z Shell (zsh). You can try to run the command `echo $SHELL` or `echo $shell`, if you are using an Unix-based operating system, to find out which shell instance you are currently running. Shells are essential components of an operating system, providing users with a powerful and flexible interface to manage and control their systems. They play a crucial role in system administration, software development, and everyday computing tasks on Unix-based operating systems.

Gnuplot is a program for representing measured data or mathematical functions. The program is freely usable and allows many functionalities for plotting a graph. \LaTeX is a software for creating documents. Other than most word processors, like Microsoft Word or Pages for MacOS, is \LaTeX code-based. This means that you write code into the .tex file and a \LaTeX -editor (e.g. Overleaf) compiles the code. The resulting document can then be saved as a PDF-file.

3 Functionalities and Built-ins

DAVIS offers a wide spectrum of commands. All commands that are provided by the system can also be used with our shell, as DAVIS just executes the command using the system calls. System calls are interfaces that allow user programs to interact with the operating system. To that we added self-implemented versions of some of these system provided functionalities, as well as a few completely new commands. The self-implemented commands are prioritized over system commands. In conclusion, our shell offers all functionalities of a provided shell and even more. In this section we are going to cover all our built-in commands and explain their usage.

3.1 ls

The first command we are going to talk about is `ls`. It lists the contents, such as directories and files, inside your specified working directory. In 1 you will see the output using the standard usage, which displays the contents visible to the user in the current working directory. Therefore it is equivalent to `ls .`, where `"."` stands for the current directory you are in. We decided to highlight executable files with the color green (DAVIS.o), symbolic links yellow (test.txt) and directories blue (commands), as it improves readability.

```
[DAVIS] ls  
davis.c  test.txt  DAVIS.o  commands  test  test_parsing.h  Makefile  test_parsing.c  davis.h  
[DAVIS]
```

Figure 1: Output of default `ls` usage.

As we mentioned, the previous example was the default usage of `ls`. We added so-called "flags" to further modify your search query. In operating systems such as MacOS, Linux or Windows there exists the possibility to create invisible files or directories. In order to display them, you can make use of the flag `-a`. If you know compare 2 to 1 you will notice two additional entries inside the resulting query, namely `".."` and `"."`. In UNIX-like operating systems, `.` represents the current directory, and `..` represents the parent directory. These special directories are commonly used in command-line operations and file system navigation. They provide a convenient way to reference directories relative to the current location or to navigate up the directory tree. Another option we show in the following image is the specification of an alternative directory you want to select (instead of `.`). This path has to be accessible and valid in order for the command to run successfully.

```
[DAVIS] ls -a ../Documents  
project-plan.txt  idea_presentation.txt  ..  examplePlot3  .  diary.txt  report  
[DAVIS]
```

Figure 2: Output of `ls -a` with specified directory.

Moreover, we added the known flag `-l`, which lists the entries in a list with additional information about each entry. In our case that would be the filename, size in bytes, what kind of entry it is and the permissions (left to right). You can see the output of this command on the next page 3.

You might be a bit confused while looking at this output. Do not worry we were too. To make sure everyone understands each number's purpose and is able to identify the properties correctly, we added our own flag `-r`. This command mostly shows it's effect in combination with the flag `-l`. The `-r` flag is our "readable" flag. It adds a title to the table indicating the purpose of each colon. Furthermore it calculates the size of each entry to the most adequate metric. We believe writing the permissions out instead of using numbers is another very helpful addition.

```
[DAVIS] ls -l
davis.c      16534   8      33204
test.txt     8        10     41471
DAVIS.o      51920   8      33277
commands     4096    4      16893
test         677     8      33204
test_parsing.h 482     8      33204
Makefile     182     8      33204
test_parsing.c 7882    8      33204
davis.h      1444    8      33204
[DAVIS] █
```

Figure 3: Output of `ls -l`.

```
[DAVIS] ls -lr
Name          Size    Type      Permissions
davis.c       16 KB   File      -rw-rw-r--
test.txt      8 B     Link      -rwxrwxrwx
DAVIS.o       50 KB   File      -rwxrwxr-x
commands      4 KB    Directory drwxrwxr-x
test          677 B   File      -rw-rw-r--
test_parsing.h 482 B   File      -rw-rw-r--
Makefile      182 B   File      -rw-rw-r--
test_parsing.c 7 KB    File      -rw-rw-r--
davis.h       1 KB    File      -rw-rw-r--
[DAVIS] █
```

Figure 4: Output of `ls -l` combined with the readable flag.

The flags can be combined randomly giving us some more possibilities we will omit.

3.2 clear

After using a command line interface for several commands it sometimes can get messy and the output hard to grasp. Because of this we offered the solution to wipe the terminal entirely. This is being done with the simple command `clear`, which uses the standard library function "`system`" that allows you to execute shell commands from within a C program.

```
[DAVIS] cat ./commands/clear.c
#include "clear.h"

int clear()
{
    system("clear");
    return 0;
}
[DAVIS] clear
```

Figure 5: Before ...

```
[DAVIS]
```

Figure 6: ... and after `clear`.

3.3 hist

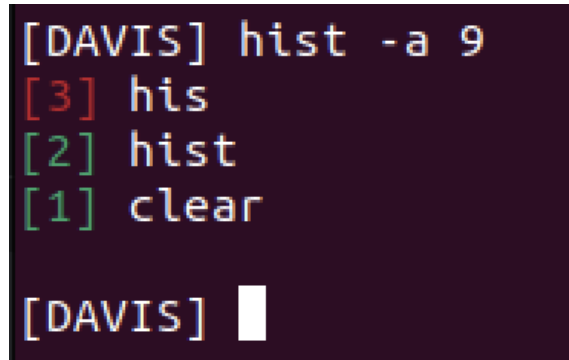
Now we are going to talk about the usage of the command `hist` displaying the recently executed commands. Yet again, we implemented several flags making it possible to further enhance precision. In the output, as displayed in 7, the general structure can be seen. The digit inside the brackets is the index of the executed command. In our case `clear` was the first executed command followed by `hst`. But since `hst` is not a valid command we color the brackets red. Equivalently `clear`'s index is shown in green.

```
[DAVIS] hist
[2] hst
[1] clear

[DAVIS]
```

Figure 7: Output of `hist`.

The first flag we added was `-a`. As an effect it displays every command, whether it was successfully executed or not. With the previously shown example of `hist`, it is clear to see, that the default usage of `hist` uses the flag `-a`. Additionally, we added the possibility to add a particular amount of past entries we want the command to show. This number can be added to any usage of `hist` no matter the flag. The default configuration of `hist` is equivalent to `hist -a 5`.

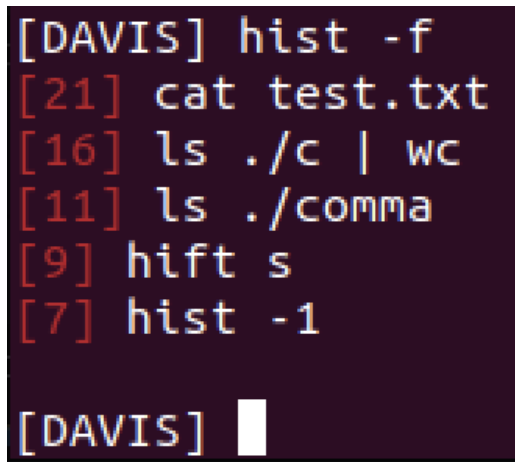


```
[DAVIS] hist -a 9
[3] his
[2] hist
[1] clear

[DAVIS] 
```

Figure 8: Output of `hist -a` with a specified amount of entries.

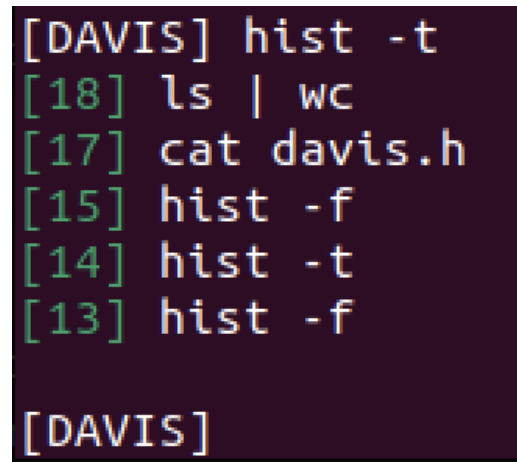
We have now covered a flag which displays every command despite their execution status. In similar fashion we added the flags `-t` and `-f`, which only search for successful commands or failed commands. In parallel to the default usage of `hist`, both flags display maximal five commands. The history command always displays the status of the executed command, whether the command included a pipe, which needed two processes to execute the command, a regular system command with one forked process or merely a self implemented command. This status was previously set by the method executing the command.



```
[DAVIS] hist -f
[21] cat test.txt
[16] ls ./c | wc
[11] ls ./comma
[9] hift s
[7] hist -1

[DAVIS] 
```

Figure 9: Output of `hist -f` ...



```
[DAVIS] hist -t
[18] ls | wc
[17] cat davis.h
[15] hist -f
[14] hist -t
[13] hist -f

[DAVIS] 
```

Figure 10: ... and of `hist -t`.

Another feature of a typical shell, we both adore, is the possibility to run the n-th previously executed command. To achieve that, we added the flag "-e" enabling us to re-run a specific command. After running f.e. "hist -e 15" we once again run the command "hist -f" (see 10). But instead of adding "hist -e" as an entry to our history, we add the executed command "hist -f" instead. Moreover you can iterate over the previous inputs using the arrow keys. The resulting behaviour is identical to most shells.

```
[DAVIS] hist -e 15
[21] cat test.txt
[16] ls ./c | wc
[11] ls ./comma
[9] hift s
[7] hist -1

[DAVIS]
```

Figure 11: Output of hist -e 15.

```
[DAVIS] hist
[23] hist -f
[22] hist -f
[21] cat test.txt
[20] hist -f
[19] hist -t

[DAVIS]
```

Figure 12: Output of hist after hist -e 15.

3.4 wordle

```
[ m ][ o ][ i ][ s ][ t ]
[ d ][ o ][ u ][ g ][ y ]
[ d ][ o ][ r ][ l ][ y ]
[ - ][ - ][ - ][ - ][ - ]
[ - ][ - ][ - ][ - ][ - ]
[ - ][ - ][ - ][ - ][ - ]
[WORDLE] dolly
[WORDLE] You won!
```

Figure 13: Successful game of wordle.

We wanted to create a user-friendly and fun shell to work with. Therefore we came up with the idea to implement a game. Our choice was the classical game wordle. It is a table with six rows and five colons. The goal is to guess a five-letter word within six tries. You may only enter words with exactly five characters. After entering a word, the given word gets added into the table. A cell gets colored green if a character is in the correct position, yellow if a character is in the word but at the wrong position and left white if a character is not part of the word whatsoever. However, if you would like to have a hint about the word you are guessing you can do so by either using "-true" to reveal a letter of the solution word or "-semi" to uncover a correct letter

in a most likely incorrect position. In 13 you can see this in action (Rows 2 and 3). In order to be able to afford a hint, you need to have a sufficient amount of points. Those points can either be obtained by entering commands on the command line or by playing wordle. If you want to quit the game, you can do so with "quit". After the game finished it too gets an entry inside the history. The status of the entry gets changed to successful if you manage to solve the puzzle. Otherwise the command "wordle" will appear red in the history listings.

3.5 echo

This command is most likely the most known system command. It prints out the parameters given in the terminal. In order to make it more appealing, we decided to add flags. Those flags just make the output look more appealing by adding color or style to it. Those flags can be combined, but only one flag per color category will be selected. This means, if you use `echo -bd Hello` your output will be displayed blue and the black color will be ignored. The colors are being checked in the order they are shown in 1. As soon as a flag has been given to a color, it gets set and the rest of the colors will be omitted. On the contrary, the style flags can be combined arbitrarily (see 14).

Flag	Color
b / B	blue
d / D	black
r / R	red
g / G	green
y / Y	yellow
m / M	magenta
w / W	white
c / C	cyan

Table 1: Lookup-table for `echo`'s color-flags.

Flag	Style
F	bold
I	cursive
U	underlined

Table 2: Lookup-table for `echo`'s style-flags.

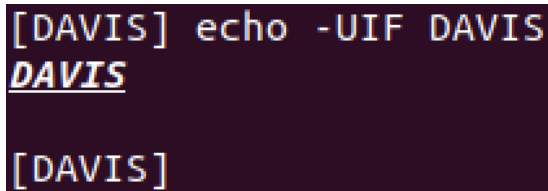


Figure 14: Usage of combined style flags ...

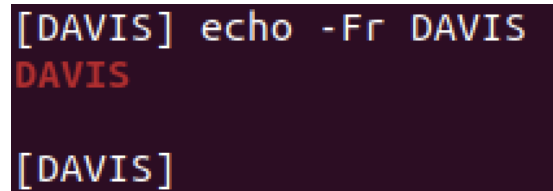


Figure 15: ... and of color and style.

In addition to that, we implemented the possibility to write text into one or several files. Although this usage of `echo` ignores given flags, it is nevertheless very useful. To write a text into specified files you need to enter the command displayed in 16. This command even allows you to create multiple empty files in one command, by simply omitting the text (see 17).

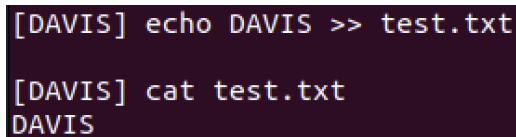


Figure 16: Redirecting text into a file.

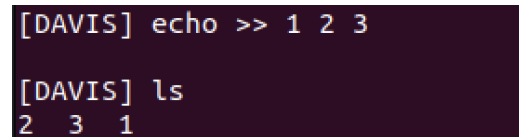


Figure 17: Creating files using `echo`.

3.6 plot

The plotting feature is one of the core additional features of DAVIS. With the `plot` command you can represent a dataset visually in a graph. The `plot` command works over the plotting program gnuplot. We added our own flags and arguments and parsed the whole command to gnuplot. As a result of the command, a new window 18 will pop up with the created graph in it. Our plotting command `plot` takes a path to a file with some data as an input. The file has to be a .csv or a text file. The file has to have a column with the x-values and a column with the y-values of a graph. If there is only one column the row number will be used as a x-value. If you want to plot a mathematical function like for example `sin(x)` you can do that too. The `plot` command can take both either a file path or a mathematical function as an input.

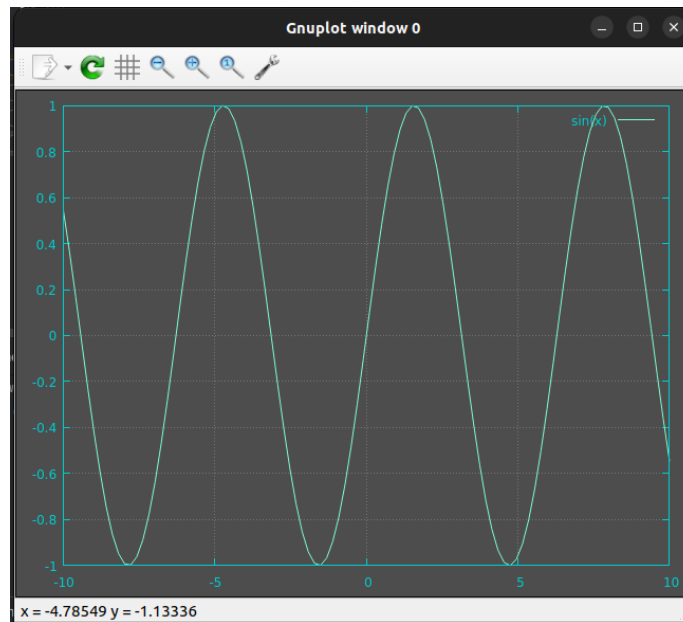


Figure 18: The gnuplot window that gets opened with the command `plot sin(x)`

3.6.1 Flags

Additionally there are arguments and flags you can input optionally. In 18 you can see the default settings for the `plot` command in DAVIS. The colors are changed and the background is dark. With the flag `-d` you can disable the darkmode and the background will be white again. To see the borders better, the color of the borders are switched back to black. The flag `-g` disables the grid you can see in the background. With the flag `-b` the top and the right border are not shown, so you only have the x- and y-axis.

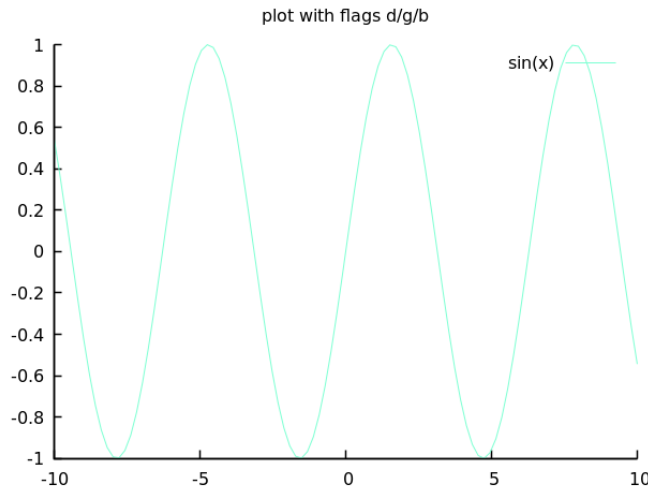


Figure 19: Plot to show the flags `-d`, `-g` and `-b` with the command `plot sin(x) -dgb`

In gnuplot a plot is per default represented only with points. We decided to use lines in the graph per default. For changing from lines to points you can use the flag `-p`. If you want to represent your plot with lines but also show each data-point explicitly, you can use the flag `-p` plus the flag `-1`. The flag `-1` alone will just leave the default setting with the lines.

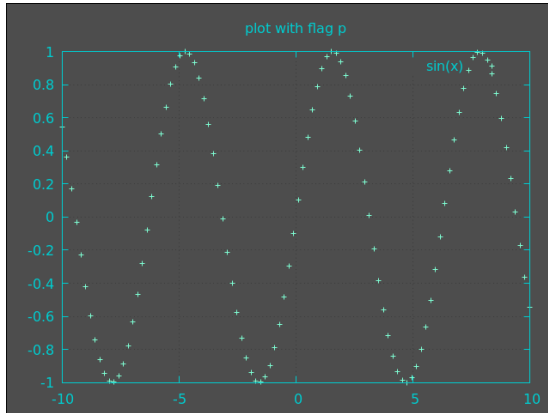


Figure 20: Plot with flag `-p` ...

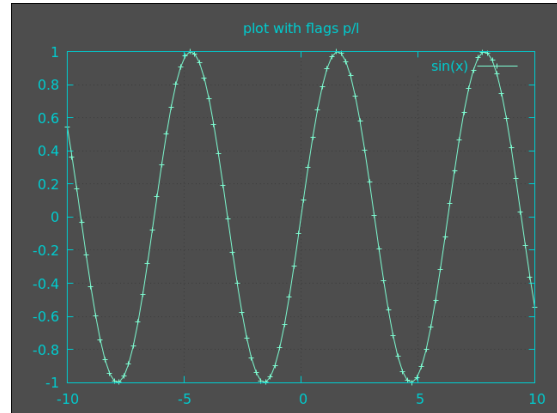


Figure 21: ... and flags `-p`, `-l`.

Another flag is the `-e` flag. With this flag the error bars respectively the error lines are shown. If the flag `-p` is set, the error bars are shown with no lines between the datapoints. If the flag `-l` or both `-l` and `-p` are set, the error bars are shown with the lines between. For this flag `-e` it is important that the input file (here: `etest.txt`) has a third column with the error data in it.

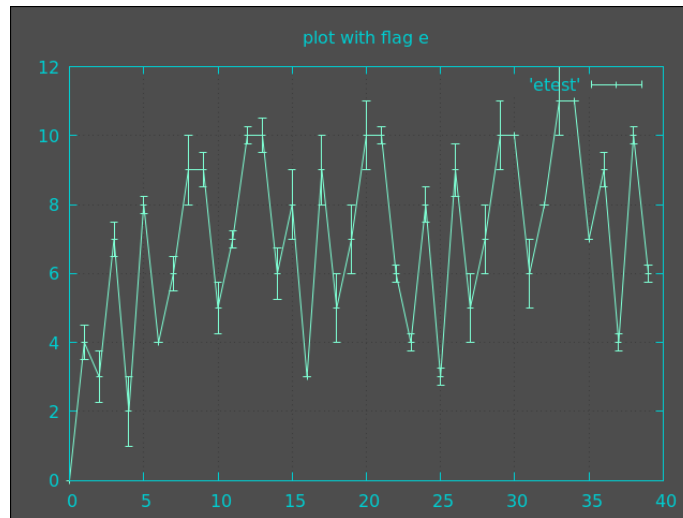


Figure 22: Plot to show the flag `-e` with the command `plot etest.txt -e`

The last flag we implemented is the `-s` flag. With the flag `-s` you are saving a .png of the produced graph directly to the directory you are using DAVIS from.

3.6.2 Arguments

There are more settings you can define in our `plot` command. There are arguments you can input like this: `plot filename argument:input`. You have to write the argument type followed by an ":" and then the value you want to input. In the following part you will see what argument types are possible.

The first argument is the `title`. With `title:<title>` you can determine what should be written above the plot. The argument `xlabel:<label>` resp. `ylabel:<label>` will put a label to the x-axis resp. the y-axis. We implemented these three arguments such that you can write an underline "_" instead of a space if you want to have more than one word.

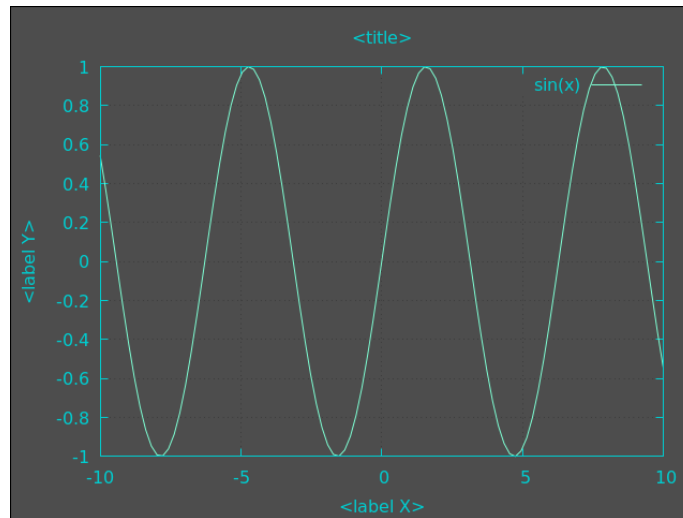


Figure 23: Plot to show the arguments `title`, `xlabel` and `ylabel` with the command `plot sin(x) title:<title> xlabel:<label_X> ylabel:<label_Y>`

You can change the color of the plot line resp. the points with the argument `color:<colorname>`. For the `color` argument you can either write the name of the color (e.g. red, blue, green, etc.) or give the hexa value of the color (e.g. #FF0000, #0000FF, #00FF00, etc.)

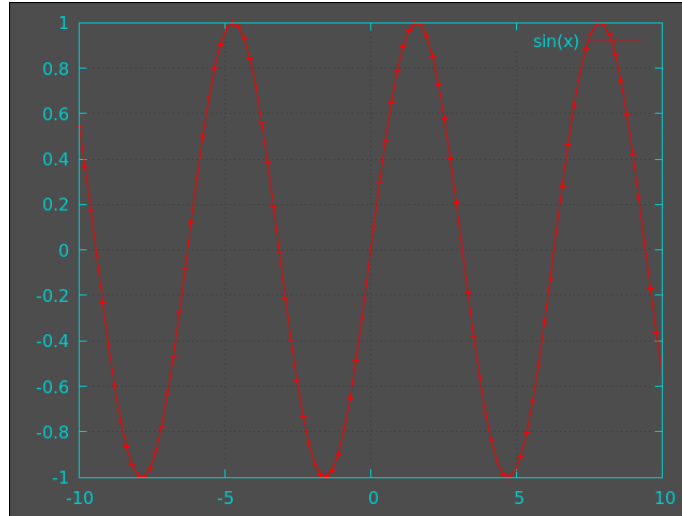


Figure 24: Plot to show the argument `color` with the command `plot sin(x) color:#FF0000 -lp`

To specify the range you want to show in the plot you can use the `xrange` resp. `yrange` arguments. As an input you have to give the range like this: `[<start>:<end>]`.

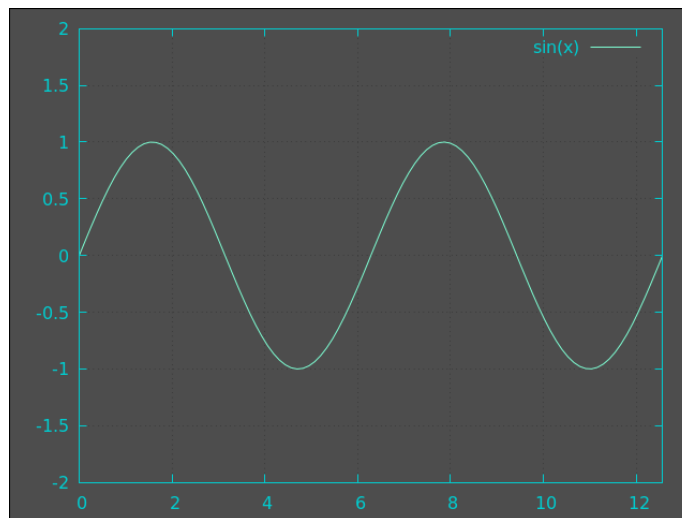


Figure 25: Plot to show the arguments `xrange` and `yrange` with the command `plot sin(x) xrange:[0:12.56] yrange:[-2:2]`

The last argument we implemented to the `plot` command is the `legend` command. This command will change the position of the legend. The legend is per default located in the top right corner of the plot. With `northwest` or short `nw` you can put it to the top left corner. Accordingly `southwest/sw` will change the location to the bottom left corner, `southeast/se` to the bottom right corner. The argument input `northeast/ne` will be just the default setting in the top right corner. Additionally you can put it in the center with `northcenter/nc` resp. `southcenter/sc`. gnuplot calls the legend also `key` or `box`. Because the legend has different names we decided to also accept the commands `key:<location>` and `box:<location>`. Both these commands will have exactly the same effect as the `legend:<location>` command.

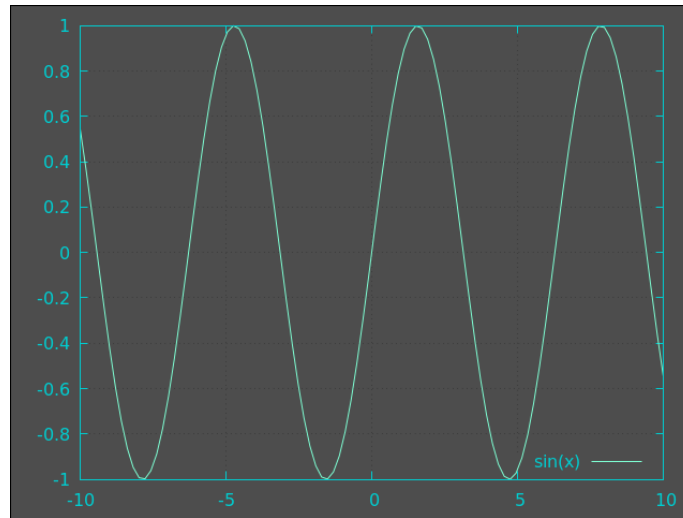


Figure 26: Plot to show the argument `legend`, `key` or `box` with the command `plot sin(x) key:southeast`

3.7 latex

Another new feature DAVIS has to offer is the `latex` command. With this command you can create templates for L^AT_EX. The newly created template will be saved into a new directory. This directory can be uploaded to any L^AT_EX-program (e.g. Overleaf) and there it can be compiled and used. You can specify the type of template and some additional arguments directly in the `latex` command like this: `latex <path/filename> <template type> {optional arguments}`. The first argument is the path where you want to save the created template including the file's name. The second argument is the type of template you want to have. There are three different templates we created for this `latex` command: `exercise`, `project-report` and `CV`. The optional arguments you can input to the `latex` command can vary from template type to template type. You can see in this table 3 which arguments go with which template type. The input of the arguments are put into the templates. If the argument is not given with the command there will be a place-holder argument you can replace yourself in the L^AT_EX-file itself. Arguments that need a space can be written with an underline instead.

exercise	project-report	CV
author:<Name_Surname>	author:<Name_Surname>	author:<Name_Surname>
course:<courseName>	title:<title_of_project>	color:<template_color>
semester:<semester>	course:<courseName>	picture:<./path/to/picture>
number:<sheet_number>	semester:<semester>	
picture:<./path/to/picture>	university:<name_of_uni>	
	picture:<./path/to/picture>	

Table 3: Arguments for different template types

In each template there is L^AT_EX-code for an example picture included. If you include the `picture` argument in your `latex` command, the picture you gave the path to will be copied to the directory that will be created. If you compile your L^AT_EX-file the picture will be in your template. All other arguments will just replace the place-holder in the templates if you include them in the `latex` command.

3.8 help

This command simply prints out a quick guide about the usage of our own commands. In the guide you can see what a command is doing and which flags and arguments are possible. There is a version in the regular shell as well as a modified instance of it while playing wordle (`-help`). While playing wordle you get a guide on which hints you can use and how you use them.

4 Methodology & Implementation

4.1 Command parser & executor

The key functionality of our shell is to process the user's requests. In order to do that successfully, we need to make sure, that the input given by the user is being parsed correctly into the program and executed accordingly. Our shell registers every single keystroke and adds the character, if it is printable, to the current input. We made this possible by using the library `termios` and set the terminal mode to "canonical". This means, that input is available to the program immediately after each character is typed. This allows for real-time processing of input. On the other hand, the default "raw"-mode (or "Non-canonical") implies, that the input is buffered and only made available to the program after the user presses Enter. We invested some time thinking about how we wanted to parse the input, because we wanted to implement our own commands. We needed to set a special set of rules determining the structure and format of our commands' inputs. Because of this we created a structure "Input" (27), containing two arrays with 30 elements with a size of 128 each. Additionally, the integer `no_commands` is set to 1, unless we have a piped command, then it is set to 2. Merely copying the client's input into a location and executing it seemed to be the incorrect way of doing it. Our final solution was it to parse the input within several steps. The initial input is being parsed into the first array (`cmd_one`). If we encounter the symbol "|", we fill up the second array (`cmd_two`) as well. In the next step, for each array, we assume that the first element is the command itself. Moreover we now operate on the ordering of their contents. We wrote an algorithm looking for elements whose first character is a "-" (flags). This algorithm had to be stable, as it might have influence in the execution of the command. That means, if two elements have the same key in the input, it will ensure that their order in the output remains the same as their order in the input. Afterwards, we concatenate the flags into one single element and move the remaining elements to not have gaps inside the parsed command. Otherwise this could lead to arguments not being processed by the command execution. For example, if you enter "`ls -l ./test -a`" the parsing will turn it into "`ls -la ./test`". This is particularly useful, because the flags given as parameters, would always end up at the same index in the array. With this logic, the system command `echo Hello -l` would return `-l Hello`, which would not match the expected output. Therefore we had to implement `echo` on our own, which allows the usage of flags. Now we have covered the command parser.

With this information we can now talk about the execution of the command. We distinguish between three different scenarios: piped, built-in command and system command. If you are trying to execute a command, it will always check the scenarios in the order they are in. We are using `forks` if the executed command is either piped or a system command. A `fork` is basically a copy of the current process running the program. We create a single new process, if we run a system command using the `execvp` command. The created process, also known as the "child-process", executes the command, while the "parent-process" waits for the child to terminate. Following that, the parent adds an history entry with the received exit status of the child. And finally, we release the memory used up for the user's input. This memory will be allocated again, as soon as the user enters another command.

```
struct Input {  
    char *cmd_one[MAX_INPUT_COUNT];  
    char *cmd_two[MAX_INPUT_COUNT];  
    int no_commands;  
};
```

Figure 27: `struct Input`, which holds the user's input.

4.2 History

We implemented a doubly linked list to keep track of the user's input. A doubly linked list is a data structure depicted in (29). It consists of nodes, where each node points to their predecessor and successor. The nodes are displayed as circles showing the executed command inside of them. In (28) we used `*prev` and `*next` as variables. The `*` indicates, that it points to a location in the memory, where an object, in this case a struct `Node`, is located. This allows us to iterate over the list by simply following the nodes along their successors or predecessors. Additionally, our history instance constantly keeps track of the first (`head`) and last (`tail`) node, as well as the amount of nodes stored in the structure. We chose to implement it doubly linked, because we wanted to be able to traverse the list from the back or merely take a step back during iteration. This is essential for navigating through the history by pressing the arrow-keys on the keyboard. You might recognise some variables from the previous struct `Input`. This is correct, because our goal is to save the user's input history, therefore it makes sense to use the same notation. Apart from the node pointers and the commands, we are missing the variables `number` and `executed`. Latter was already covered in the previous section 4.1. The integer `number` is the unique identifier of each history entry. This is the number, which you need in order to execute a particular command using the command `hist -e [number]`. The number is been given by the history and can not be changed whatsoever. Moreover, the uniqueness of this number across all entries inside the history is enforced by the history. If you manage to create an amount of entries exceeding the integer value (2'147'483'647) it will reset the history and start from the beginning. Feel free to test that. If the user decides to print out the history, all what happens in the background is, that we iterate over the history and print the selected information. The trickier part is the `hist -e` command. We replace the current input with the input saved inside the history entry of the requested command. After executing the newly modified input, we add the modified input to the history. This means, that the command added to the history is not `hist -e` but the command which was executed. If we were to add the actual input of the client (`hist -e`), we would run into weird behaviour when trying to re-run a particular command.

```
struct Node {  
    struct Node *prev;  
    struct Node *next;  
    char *cmd_one[MAX_INPUT_COUNT];  
    char *cmd_two[MAX_INPUT_COUNT];  
    int no_commands;  
    int number;  
    int executed;  
};
```

Figure 28: Structure of the elements of our history.

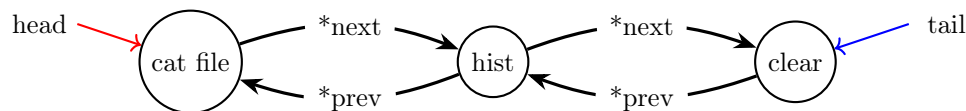


Figure 29: Structure of our history as a doubly linked list.

4.3 Plotting and Latex

The `plot` and the `latex` command where both similar in some ways. We decided to implement a variety of arguments by introducing the syntax `argument:input`. For both commands most arguments consist of the type of the argument and the value the user wants to give to that argument. The type and the value get separated by a ":". The order of the optional arguments is therefore not relevant.

4.3.1 plot

For the `plot` command we used gnuplot. Gnuplot is a program that offers many many functionalities. We had to decide which ones were relevant and which ones were unnecessary. Our goal was to make the plotting simple and to be able to plot a nice looking graph with only one command. For that we had to parse our command syntax to the gnuplot syntax. We build a string out of our commands that we send to gnuplot with the help of a pipe. The arguments in our `plot` command tell us which commands we have to append to the string so gnuplot can plot the right graph. When the string is finished we print it to the pipe and flush it, so gnuplot can execute the commands we sent and show the right graph. Gnuplot has "set" commands and "plot" commands. The "set" commands have to be executed separately before the "plot" command is executed. We had to bypass this by flushing all "set" commands in a separate string and then flush the "plot" command after. All our arguments and flags can be solved with a "set" or a "plot" command from gnuplot. If you want to save a graph as a picture in gnuplot, you have to use some "set" commands and if you already have a picture with the same name in the directory you save it to it will just overwrite it. For the `-s` flag we implemented it that it doesn't overwrite a picture when it has the same name, but rather saves it with an number at the end of the name (e.g. `pic.png` already exists -> `pic(1).png`).

4.3.2 latex

With the `latex` command we first had to decide what types of templates we wanted to integrate. We decided to only integrate three types of templates because the goal of this project was not to play around with L^AT_EX but to create a shell. We could always easily add more templates if we wanted because the code is modular. A new template would not need many changes to add. We have the different templates in the `templates` folder in the `resources` folder of DAVIS. If you use the `latex` command we will check if the first argument that is given is a existing path and create a directory there. If there already exists a directory with the given name, we just add a number behind it (e.g. `exampleDir` already exists -> `exampleDir(1)`). The second argument then decides which template of the folder in the resources will be copied into the directory. If there are more arguments we put them at the beginning of the template as a L^AT_EX-variable. L^AT_EX will then put them in all places where the template has a place-holder for that variable. If a variable is not specified by the user, we just put in a default value that can later be changed manually.

5 Problems & Solutions

5.1 Keystrokes

We regularly use shells, which lead to high expectations for our shell on our side. Initially our input would be taken as a whole. This did work well for the general usage of our shell, but was not ideal to handle arrow key inputs, as the keys would only be registered after you pressed enter and not instantly. Registering keystrokes was necessary to navigate through previous inputs. Because it was not possible to do so, we added the command `hist -e` to at least make it possible using commands. In order to react to pressed keys instantly we tried three different approaches. We started off by creating a separate thread listening for special keys. This caused unexpected behaviour, as the standard input (`stdin`) is not thread-safe. The next approach included the external library `ncurses`. With it we were able to capture the keystrokes, but executing commands was not possible anymore. For our last, and successful, approach, we rewrote the method responsible for taking the input from the user with the help of the `termios` library. Now the input gets constructed character by character. Because we are able to react to each character individually, we used a similar logic, as described in 4.2.

5.2 Robustness and Security

In our final phase of development we looked into security related topics. We had the issue, that the program would crash, if the user entered a too large amount of characters causing a buffer overflow resulting in a crash. The bug was, that we read the same amount of bytes from the inputstream as there was memory in the array we would copy the input to. This could lead to the string not being zero-terminated and thus more errors. We went over the code and added checks, whether the created string is actually zeroterminated and no other memory-related problems could occur. Moreover we tried to crash the program, or cause it to do something wrong, with various dummy inputs. This helped us identify and address some loopholes we had either overlooked or not covered during implementation. We did this to enhance the robustness of the program, as these issues could have posed serious problems if left unresolved.

We did some stress-testing for the `plot` command. We tested the plotting with a file of size 3GB and 75'000'000 rows of data and it worked well. The limit of our `plot` command is at about 100'000'000 rows. We can not change that because this is just the limitation of gnuplot we have to work with.

5.3 Gnuplot

A problem with gnuplot was that gnuplot treated the contents of `.txt` files differently than `.csv` files. Additionally gnuplot has problems with bigger `.csv` files as we noticed. Our solution for both these problems was to copy every `.csv` file into a `.txt` file and let gnuplot work with the `.txt` file instead. This worked really good for us and solved many problems.

Another problem we have with gnuplot is that we can't really process any error messaging gnuplot gives back. This problem still exists but we tried to catch every wrong input as good as possible, so gnuplot doesn't fail and almost always will show the plot.

It was generally a problem to use a third-party program like gnuplot, because we had to live with the way gnuplot works and we had to adjust our program correspondingly.

6 Conclusion

6.1 Lessons learned

Since we both know each other for a good amount of time now, we never had problems communicating with each other. Additionally, this is not the first project we have done together. Nevertheless we learned that good planning is important. We have set our goal to finish the project well before its due and this turned out to be a great idea. Both of us have had few experiences with the programming language C. But throughout the development phase of **DAVIS** we got familiar with it. Since we were merely used to high level languages, such as Java, we had to make sure to create more robust code. Moreover the exercises we did in the course helped us on various occasions. Working with a new program like gnuplot was challenging at times but we were learning much about gnuplot and plotting in general. We got to know the internal structure of an unix based system and how a shell works under the hood. On top of that, we had to dive into the topic of memory utilization and general resource management. Now, having done the project, we think we would be able to improve our final product, if we did it again.

7 References

Following we are going to list all external sources we have used for the project:

- Github for colored output (18.05.2024)
- Stackoverflow for general questions
- Our project's repository

8 Appendix

8.1 Declaration of Independent Authorship

We attest with our individual signatures that we have written this report independently and without outside help. We also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, we affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used.

This report may be checked for plagiarism and use of AI-supported technology using the appropriate software. We understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Two handwritten signatures are written on a horizontal line. The signature on the left is a cursive 'L' followed by 'S'. The signature on the right is a cursive 'M' followed by 'F'.