Corporate Research / Advance Autonomous Systems (CR/AAS3)

Robotic Systems Engineering

Internship Report

Made by:
Huu Duc Nguyen

Supervised by:
Robert Schirmer
Marco Lampacrescia

February, 2022

Acknowledgments

I would like to thank the Robert Bosch GmbH for this opportunity, Robert Schirmer M.Sc. and Marco Lampacrescia M.Sc. for supervising and guiding me during this internship.

Contents

General abbreviations

BASH	Bourne Again SHell
CI	Continuous Integration
CLI	Conmand Line Interface
CPP	Coverage Path Planning
CSP	Covering Salemans Problem
EE	End effector
PID	Process identification number
POD	Plain Old Data
PP	Path Planning
ROS	Robot Operating System
TSP	Travelling Salemans Problem

1 Introduction

In the last semester of the robotic master program, there are two paths to follow: either focusing on academic research or gaining more industrial experience. With a desire to gain more hands-on experiences and skills, I decided to pursuit the industrial track, which requires a mandatory internship at an external company. I have already finished the master thesis at the Robert Bosch GmbH before the internship. Around the end of the thesis, my supervisors and I have discussed and agreed to continue on an 12-week internship.

Robert Bosch GmbH [Bos], also known as Bosch, is a multinational engineering and technology company. It was founded by Robert Bosch in 1886 in Stuttgart and currently is headquartered at Gerlingen. The company majors in four business sectors: Mobility Solutions, Industrial Technology, Consumer Goods, and Energy and Building Technology.

During the internship, I implemented additional features, which relate to my master thesis, into the department's robotic pipeline. The goal is to support the development of the next generation cleaning robot. This report summarizes what I have learn during the time spending at the company.

2 Coding Languages And Tools

Across the time working on the thesis and the internship, I had the opportunity to experience the professional workflow with state-of-the-art methods. This section summarizes common languages, tools for developers and the workflow. In addition the presented languages and tools, I also either worked with or got to know more about Docker [Mer14], Python, Vim, git, etc.

2.1 Bash

2.1.1 Introduction

Linux is a family of operating systems based on the Linux kernel. For public users, it is less common and well-known comparing to Windows and Mac. These two receive more attention for application developments, however, give little access to the source code and are more prone to malware. On the other hands, Linux is open-source and enables more accesses to modify the system configurations. For engineers and developers, Linux induces a better working environment.

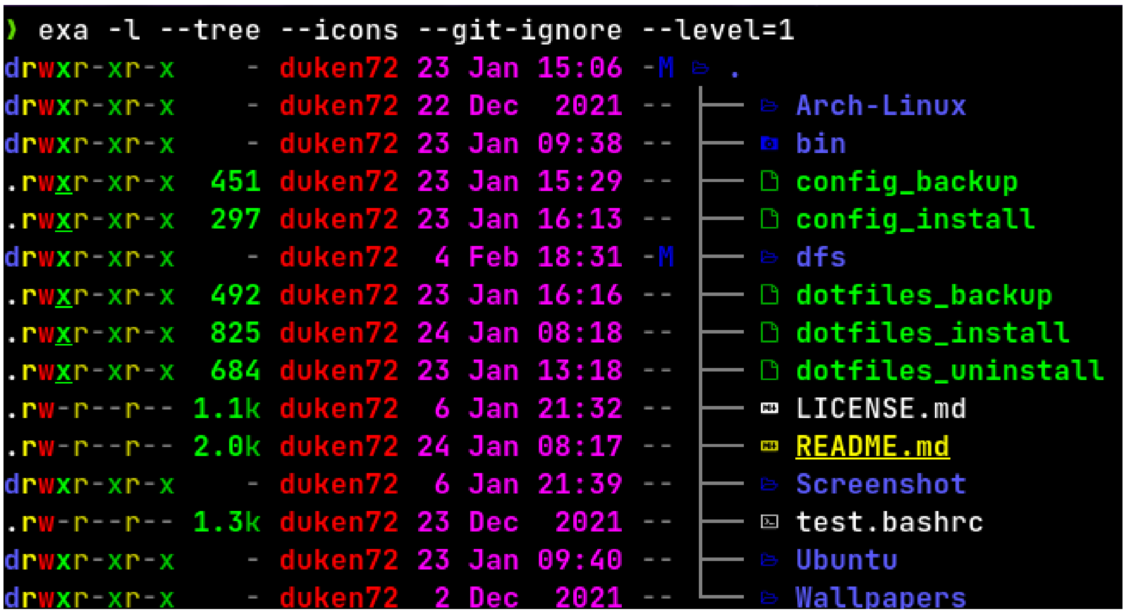
Bourne Again SHell (BASH) is a Unix shell and command language. It is the default login shell for most Linux distributions. Bash allows users to interact with the operating system, accessing, managing file system.

2.1.2 Basic commands

Developers would spend a significant part of his time working in the Conmand Line Interface (CLI). It is important to familiarize themselves with common BASH/Shell commands and programs:

- View the file system: `ls, tree`, `broot` [Sé], `exa` [Sag], `ranger` [Ran].

  Working in a coding project, developers would find themselves going through either existing or green field file system. It is crucial to know what files or directories there are, their positions, whether one could read, write or execute the files, etc. `ls` and `tree` are the basic commands to view the file system and related information. `broot,` `exa` and `ranger` are more advanced commands with more features, e.g., color coding, interactive actions (change directory, move, delete files, etc.). Files and directories have permissions, which can be view from the long descriptions. "d" implies directory, otherwise "." implies a file. The next 9 characters show the permissions of three groups, i.e., owner, owning group and everyone else. Each group could have 3 permissions: "r" read, "w" write and "x" execute. In addition, these commands could display information regarding file size, file owner, modified date, git tracking status (if available).



```
) exa -l --tree --icons --git-ignore --level=1
drwxr-xr-x    - duken72 23 Jan 15:06 -M  .
drwxr-xr-x    - duken72 22 Dec  2021 --  ├── Arch-Linux
drwxr-xr-x    - duken72 23 Jan 09:38 --  ├── bin
.rwxr-xr-x  451 duken72 23 Jan 15:29 --  ├── config_backup
.rwxr-xr-x  297 duken72 23 Jan 16:13 --  ├── config_install
drwxr-xr-x    - duken72  4 Feb 18:31 -M  ├── dfs
.rwxr-xr-x  492 duken72 23 Jan 16:16 --  ├── dotfiles_backup
.rwxr-xr-x  825 duken72 24 Jan 08:18 --  ├── dotfiles_install
.rwxr-xr-x  684 duken72 23 Jan 13:18 --  ├── dotfiles_uninstall
.rw-r--r-- 1.1k duken72  6 Jan 21:32 --  ├── LICENSE.md
.rw-r--r-- 2.0k duken72 24 Jan 08:17 --  ├── README.md
drwxr-xr-x    - duken72  6 Jan 21:39 --  ├── Screenshot
.rw-r--r-- 1.3k duken72 23 Dec  2021 --  ├── test.bashrc
drwxr-xr-x    - duken72 23 Jan 09:40 --  ├── Ubuntu
drwxr-xr-x    - duken72  2 Dec  2021 --  └── Wallpapers
```

Figure 2.1: Viewing file system with `exa`.

- View files or directories descriptions: `file`, `du`, `cloc` [AlD], `tokei` [XAM]

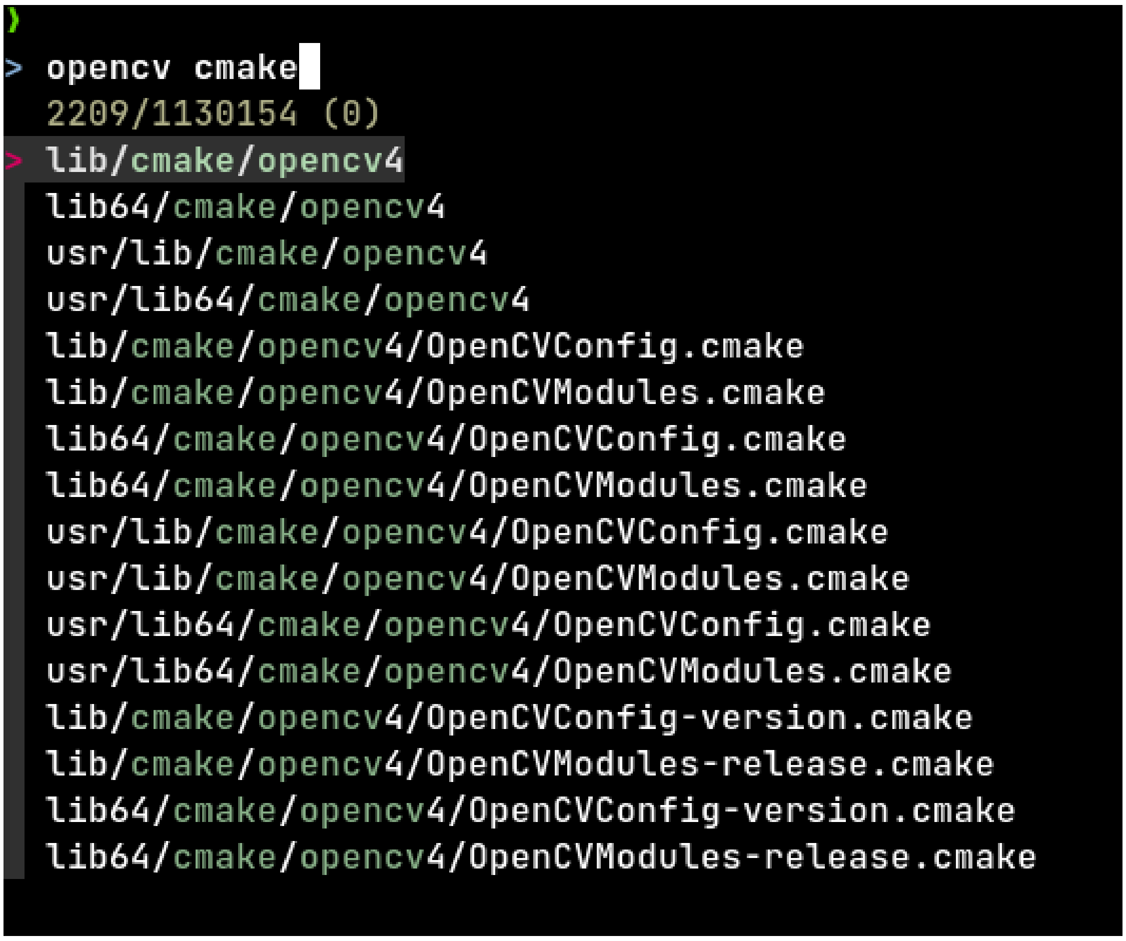
  `file` - determine file type, `du` - estimate file space usage, `cloc` - count lines of code, `tokei` - display statistics about your code. Some additional info regarding number of code lines, files, file and directory memory size, comments, etc., is also needed. These info can be accessed and included with these commands. Example of `tokei`:

```
================================================================
Language          Files     Lines      Code   Comments    Blanks
================================================================
BASH                  8       118        85         14        19
C Header             26      1871       986        528       357
C++                  16      3373      2627        326       420
Makefile              2        60        38          4        18
Python                1       173       110         36        27
Shell                 4      1669      1392        162       115
Plain Text            4        98         0         86        12
Zsh                  14     20719     13838       5146      1735
----------------------------------------------------------------
Markdown              8      2998         0       2225       773
|- BASH               3       119       103          6        10
|- C++                1       137       132          0         5
|- Lua                2         8         8          0         0
|- YAML               2         6         6          0         0
|- Zsh                3       167       136         20        11
(Total)                      3435       385       2251       799
================================================================
Total                83     31079     19076       8527      3476
================================================================
```

- Change ownership, permission: `chown`, `chmod`.

  `chown` - change file or directory ownership, `chmod` - change file or directory permissions.
- Search for file: `locate, find`, `fdfind` [Petb], `fzf` [Cho].

  `locate` - list files in databases that match a pattern, `find` - search for files in a directory hierarchy, `fdfind` - user-friendly alternative to `find`, `fzf` - a general-purpose command-line fuzzy finder.



```
❯
> opencv cmake
  2209/1130154 (0)
> lib/cmake/opencv4
  lib64/cmake/opencv4
  usr/lib/cmake/opencv4
  usr/lib64/cmake/opencv4
  lib/cmake/opencv4/OpenCVConfig.cmake
  lib/cmake/opencv4/OpenCVModules.cmake
  lib64/cmake/opencv4/OpenCVConfig.cmake
  lib64/cmake/opencv4/OpenCVModules.cmake
  usr/lib/cmake/opencv4/OpenCVConfig.cmake
  usr/lib/cmake/opencv4/OpenCVModules.cmake
  usr/lib64/cmake/opencv4/OpenCVConfig.cmake
  usr/lib64/cmake/opencv4/OpenCVModules.cmake
  lib/cmake/opencv4/OpenCVConfig-version.cmake
  lib/cmake/opencv4/OpenCVModules-release.cmake
  lib64/cmake/opencv4/OpenCVConfig-version.cmake
  lib64/cmake/opencv4/OpenCVModules-release.cmake
```

Figure 2.2: Searching for file with fuzzy finder `fzf`.

- Read guidance: `man`, `which`, `apropos`, `type`.

  `man` - an interface to the system reference manuals, `which` - shows the full path of (shell) commands, `apropos` - search the manual page names and descriptions, `type` - write a description of command type.
- Manage files and directories: `mkdir`, `rmdir`, `touch`, `cp`, `mv`, `rm`, `ln`.

  `mkdir` - make directories, `rmdir` - remove empty directories, `touch` - change file timestamps, `cp` - copy files and directories, `mv` - move (rename) files, `rm` - remove files or directories, `ln` - make links between files.
- View file content: `cat, more, less`, `bat` [Peta].

  `cat` - concatenate files and print on the standard output, `more` - file filter for paging through text one screenful at a time, `less` - provide `more` emulation plus extensive enhancements, `bat` - a `cat` clone with syntax highlighting and Git integration.

- Edit file content: `nano`, `gedit`, `vi`, `vim`, `sed`, `sd` [Gre], `awk`, or file-specific application.

  Example commands for `sd` and `sed`:

  ```
  sd [prev_expression] [new_expression] [file]
  sd [prev_expression] [new_expression] [file] -p #p-preview
  # just print out the results
  sed 's/[prev_expression]/[new_expression]/g' [file.txt]
  # replace a string with another
  sed -i 's/[prev_expression]/[new_expression]/g' [file.txt]
  sed '/searchStr/c\newLine' [file.txt] # replace a line contain a string

  awk '/pattern/ {print $2}' file.txt
  awk -f awkFD.awk file.txt
  ```

- Archive: `zip`, `unzip`, `zipcloak`, `zipslit`, `tar`, `gzip`, `gunzip`, `unrar`.

  `zip` - package and compress (archive) files, `unzip` - list, test and extract compressed file in a ZIP archive, `zipcloak` - encrypt entries in a ZIP file, `zipslit`, `tar` - an archiving utility, `gzip`, `gunzip` - compress and expand files, `unrar` - uncompress RAR archive .

  Some of the tags for `zip` are: `-r` for recursive search, `-e` for encryption, `-v` for more verbose output, `-9` for better compression. Example of recursive compression with encryption:

  ```
  zip -er9 output.zip file1 file2
  ```

  Some tags for `unzip` are: `-x` for files exclusion, `-o` for overwrite, `-n` for not-overwrite, `-d` for output directory, `-l` for listing content. Examples of uncompress a `.zip` file:

  ```
  unzip -o input.zip -x *.h -d target_dir
  upzip -l input.zip
  ```

  Additional commands for `.zip` files:

  ```
  zipcloack file.zip # add password
  zipsplit -n [size_in_bytes] file.zip # split to size restriction
  ```

  Some tags for `tar`: `v` for more verbose output, `f` for files, `c` for create, `z` for `.gunzip` file, `x` for extract, `t` for listing. Examples of `tar` commands:

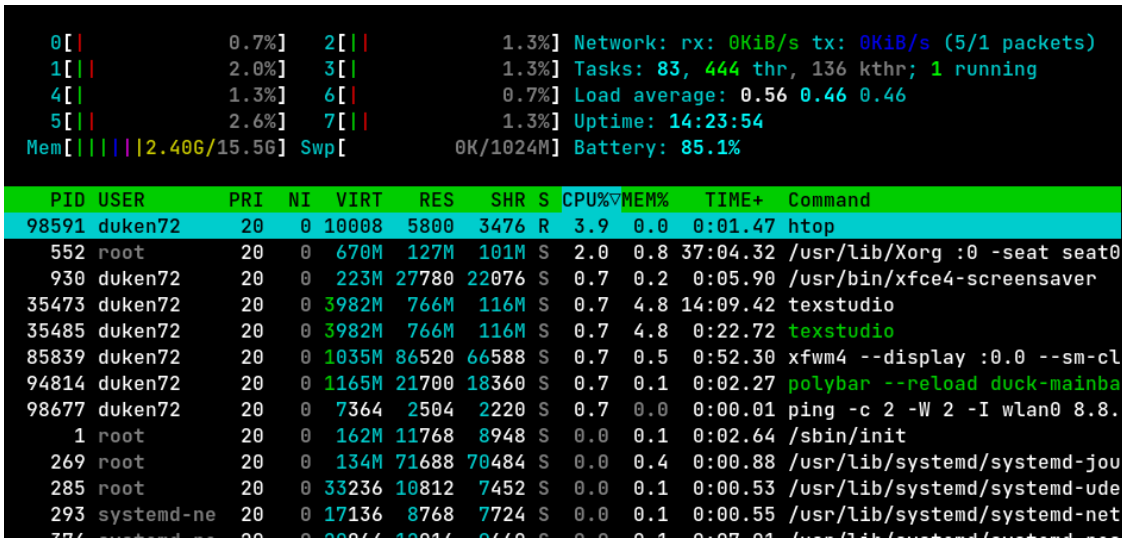
  ```
  tar cvf [target_file.tar] [files/dirs] # create .tar
  tar zcvf [target_file.tar.gz] [files/dirs] # create .tar.gz
  tar tvf [file.tar] # list out files in file.tar
  tar xvf [file.tar] -C [dirs] # extract tar file
  tar xvfz [file.tar.gz] [dirs] # extract tar.gz file
  ```

  Other commands with `gzip, gunzip, unrar` for archiving and compression:

  ```
  gzip [file.tar] # compress the tar file to .tar.gz file
  gunzip [file.tar.gz] # uncompress the .tar.gz file to .tar file
  unrar x [file.rar]
  ```

- Monitor your system: `htop`, `neofetch`

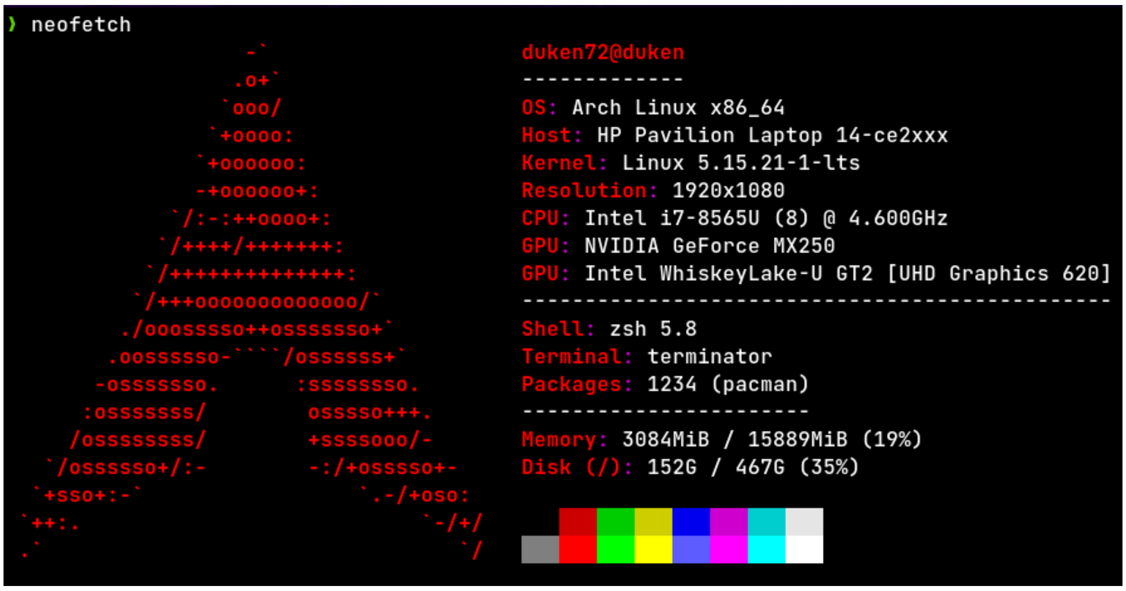
  `htop` is the equivalent Linux-version of Task Manager in Windows, from which user can inspect current running process, application, in terms of memory, CPU percentages, end a process if desired.



```
    0[|             0.7%]    2[||             1.3%] Network: rx: 0KiB/s tx: 0KiB/s (5/1 packets)
    1[||            2.0%]    3[|              1.3%] Tasks: 83, 444 thr, 136 kthr; 1 running
    4[|             1.3%]    6[|              0.7%] Load average: 0.56 0.46 0.46
    5[||            2.6%]    7[||             1.3%] Uptime: 14:23:54
  Mem[||||||2.40G/15.5G] Swp[            0K/1024M] Battery: 85.1%

    PID USER       PRI  NI  VIRT   RES   SHR S CPU%▽MEM%   TIME+  Command
  98591 duken72     20   0 10008  5800  3476 R  3.9  0.0  0:01.47 htop
    552 root        20   0  670M  127M  101M S  2.0  0.8 37:04.32 /usr/lib/Xorg :0 -seat seat0
    930 duken72     20   0  223M 27780 22076 S  0.7  0.2  0:05.90 /usr/bin/xfce4-screensaver
  35473 duken72     20   0 3982M  766M  116M S  0.7  4.8 14:09.42 texstudio
  35485 duken72     20   0 3982M  766M  116M S  0.7  4.8  0:22.72 texstudio
  85839 duken72     20   0 1035M 86520 66588 S  0.7  0.5  0:52.30 xfwm4 --display :0.0 --sm-cl
  94814 duken72     20   0 1165M 21700 18360 S  0.7  0.1  0:02.27 polybar --reload duck-mainba
  98677 duken72     20   0  7364  2504  2220 S  0.7  0.0  0:00.01 ping -c 2 -W 2 -I wlan0 8.8.
      1 root        20   0  162M 11768  8948 S  0.0  0.1  0:02.64 /sbin/init
    269 root        20   0  134M 71688 70484 S  0.0  0.4  0:00.88 /usr/lib/systemd/systemd-jou
    285 root        20   0 33236 10812  7452 S  0.0  0.1  0:00.53 /usr/lib/systemd/systemd-ude
    293 systemd-ne  20   0 17136  8768  7724 S  0.0  0.1  0:00.55 /usr/lib/systemd/systemd-net
```

Figure 2.3: System inspection with `htop`.



```
❯ neofetch
                    -`                     duken72@duken
                   .o+`                    -------------
                  `ooo/                    OS: Arch Linux x86_64
                 `+oooo:                   Host: HP Pavilion Laptop 14-ce2xxx
                `+oooooo:                  Kernel: Linux 5.15.21-1-lts
                -+oooooo+:                 Resolution: 1920x1080
              `/:-:++oooo+:                CPU: Intel i7-8565U (8) @ 4.600GHz
             `/++++/+++++++:               GPU: NVIDIA GeForce MX250
            `/++++++++++++++:              GPU: Intel WhiskeyLake-U GT2 [UHD Graphics 620]
           `/+++ooooooooooooo/`            -----------------------------------------------
          ./ooosssso++osssssso+`           Shell: zsh 5.8
         .oossssso-````/ossssss+`          Terminal: terminator
        -osssssso.      :ssssssso.         Packages: 1234 (pacman)
       :osssssss/        osssso+++.        ------------------------
      /ossssssss/        +ssssooo/-        Memory: 3084MiB / 15889MiB (19%)
    `/ossssso+/:-        -:/+osssso+-      Disk (/): 152G / 467G (35%)
   `+sso+:-`                 `.-/+oso:
  `++:.                           `-/+/
  .`                                 `/
```

Figure 2.4: System inspection with `neofetch`.

- Cryptography: `openssl`

  Example commands:

```
openssl aes-256-cbc -salt -pbkdf2 -in <file.name> -out <file.enc.name>
openssl aes-256-cbc -d -pbkdf2 -in <file.enc.name> -out <file.dec.name>
cmp <file1> <file2> | echo \$? \#compare 2 file, 0 if same, 1 if different
```

- Managing working processes: `fg`, `bg`, `jobs`, `kill`.

  `fg` - run jobs in the foreground, `bg` - run jobs in the background, `jobs` - display status of jobs in the current session, `kill` - terminate a process.

2.1.3 Shell scripting

As a scripting language, user-defined aliases and programs could incorporate different basic commands. On top of a Shell program, the shebang must be included: `#!/bin/sh` or `#!/bin/bash`. The Shell language allows defining variables, accessing arguments, logical operator, flow control, etc.

Variables to access

The variables are accessed via `$` sign.

- `$0` - Name of the script
- `$1` to `$9` - Arguments to the script. `$1` is the first argument and so on.
- `$@` - All the arguments
- `$#` - Number of arguments
- `$?` - Return code of the previous command
- `$$` - Process identification number (PID) for the current script
- `!!` - Entire last command, including arguments, e.g.: `sudo !!` if last command fail due to permission
- `$_` - Last argument from the last command.

Logical operators

There are three logical operators:

- The `||` operator:

  ```
  false || echo "Oops, fail" # Oops, fail
  true || echo "Will not be printed"
  ```

- The `&&` operator acts like an `AND` operator:

  ```
  true && echo "Things went well" # Things went well
  false && echo "Will not be printed"
  ```

- The `;` operator acts like an `OR` operator:

  ```
  true ; echo "This will always run"
  false ; echo "This will always run"
  ```

Flow controls

With BASH, there are flow controls just like in common languages, e.g., Python or C++.

- If-else statement:

```
if conditions; then
  commands
[elif conditions; then
  commands]
[else conditions; then
  commands]
fi

# Multiple conditions:
if ([condition1] || [condition2]) && [condition3]; then
  commands
fi
```

- For loop:

```
for i in word1 word2 word3; do
  echo "$i"
done

for i in "$@"; do
  echo $i
done
```

- While loop:

```
while [ condition ] do
  command1
  command2
  command3
done
```

Comments

- Single line comments with "#"
  Example: `# Flow Control`
- Block comments

```
: <<'END'
echo "I hope that there is no error"
END
```

Logic conditions

Common logical conditions relate with file system. There are syntax to check whether a file exists and is of the expected type.

```
if [ -d file ]; then echo "file is a directory" fi
if [ -e file ]; then echo "file exists" fi
if [ -f file ]; then echo "file exists and is a regular file" fi
if [ -L file ]; then echo "file is a symbolic link" fi
if [ -r file ]; then echo "file is a file readable by you" fi
if [ -w file ]; then echo "file is a file writable by you" fi
if [ -x file ]; then echo "file is a file executable by you" fi
if type cmd; then echo "cmd exists" fi

if [ file1 -nt file2 ]; then echo "file1 is newer than file2" fi
if [ file1 -ot file2 ]; then echo "file1 is older than file2" fi
```

In addition, there are syntax logic conditions to comparison between `string`:

```
if [ -z string ]; then echo "string is empty" fi
if [ -n string ]; then echo "string is not empty" fi
if [ string1 = string2 ]; then echo "string1 equals string2" fi
if [ string1 != string2 ]; then echo "string1 does not equal string2" fi
```

2.1.4 CLI Hotkeys

It is useful to practice and remember some basic hotkeys while working on the CLI. It should noted that different shells have different features regarding line completion and movements.

Commands	Actions
`CTRL-A / E`	move to home/end of line
`CTRL-F / B`	move for/backward a character
`ALT -F / B`	move for/backward a word
`CTRL-H`	delete char
`CTRL-W`	delete a word back
`CTRL-K`	clear to end of line
`CTRL-U`	clear the line
`CTRL-S / Q`	toggle screen output
`CTRL-L`	clear screen output
`Ctrl-D`	exit shell
`CTRL-C`	stop current process

2.2 CMake

2.2.1 Introduction

CMake is a tool for build system, usually for projects or programs written in C or C++ language. Programmers write and work with source code. However, in order for the computers to understand and execute those programs, the source code needs to be built (compile and link with libraries) into binary file. Previously, `Makefiles` compile and build the source code, thus there is a need to maintain and update these `Makefiles`. As the project scale grows, this maintaining and updating tasks become more tedious. E.g., when working with ROS, current package might use multiple external packages and libraries, which could have more than one executable, different build and test configurations. Build tools like CMake are developed to automate the complex build process. Instead of manually manage `Makefiles`, developers manage `CMakeLists.txt` file, which will create `Makefiles`. This section gives an overview on CMake.

2.2.2 CMake Syntax

CMake has its own scripting language, which includes variables, statements, flow control, commands, etc. Some basic commands, syntax are as follows:

- Print message

  ```
  message("Hello world!")     # Print "Hello world!"
  message("Hello ${NAME}!")   # Substitute a variable into the message
  ```

- Assign variables

  ```
  set(THING duck)
  message("We want the ${THING}!")
  ```

  In CMake, every variables are strings, which in most case relate to paths, names of files, libraries or configurations.

- Math operation

  ```
  math(EXPR MY_SUM "1 + 1")    # Store 2 in MY_SUM
  message("The sum is ${MY_SUM}.")
  ```

- Flow control

  – If-else statement

    ```
    if(WIN32)
      message("You're running CMake on Windows.")
    else()
      message("You're not running CMake on Windows.")
    endif()
    ```

  – While loop

    ```
    set(A "1")
    set(B "1")
    while(A LESS "7")
      message("A = ${A}")         # Print A
      math(EXPR T "${A} + ${B}")  # Store the sum of A and B in T
      set(A "${B}")               # Assign the value of B to A
      set(B "${T}")               # Assign the value of T to B
    endwhile()
    ```

  – For loop

```
foreach(ARG These are separate arguments)
  message("${ARG}")       # Prints each word on a separate line
endforeach()
```

- Combine commands with variable:

```
set(ARGS "EXPR;T;1 + 1")
math(${ARGS})   # Equivalent to math(EXPR T "1 + 1")
message("${ARGS}")
```

- List

```
set(MY_LIST These are separate arguments)
message("${MY_LIST}")
# Prints: These;are;separate;arguments

set(MY_LIST These are separate arguments)
list(REMOVE_ITEM MY_LIST "separate")
# Removes "separate" from the list
message("${MY_LIST}")
```

- Function

```
function(doubleIt VALUE)
  math(EXPR RESULT "${VALUE} * 2")
  message("${RESULT}")
endfunction()

# Prints: 8
doubleIt("4")
```

- Marco

```
macro(doubleIt VARNAME VALUE)
  # Set the named variable in caller's scope
  math(EXPR ${VARNAME} "${VALUE} * 2")
endmacro()

# Tell the macro to set the variable named RESULT
doubleIt(RESULT "4")

# Prints: 8
message("${RESULT}")
```

2.2.3 CMake Target and Properties

Targets are executable files, binary files. As the name itself suggests, they are the targets, the goal of CMake, to compile and build source code (in C++) to these executables. In other cases, instead of executable(s), the targets to build could be libraries, which are groups of functionalities that could be used somewhere else. The commands to define these targets are:

- `add_executable`: define the target
- `add_subdirectory, add_library, add_custom_target`: call `CMakeLists.txt` from lower directory level.

These executables have properties, which are to be managed in order for the source code to be compiled and linked properly. Some common properties are:

- `INCLUDE_DIRECTORIES`

  Include directories are where to find the C++ header files. With correct paths, header files can be included on top of the `.cpp` files: `#include <something.h>`. This is where the compiler will find function declarations, type definitions, classes, structures, etc.
- `LINK_LIBRARIES`

  While compiler concerns with the declarations, linker concerns with functions, methods definitions. These are handled by link libraries. In the build process, a target has to be compiled and linked in order to become a executable.
- `COMPILE_DEFINITIONS`
- `COMPILE_OPTIONS`

The commands to change or update these properties:

- `(target_)include_directories`
- `(target_)link_libraries`
- `(target_)compile_definitions`
- `(target_)compile_options`
- `set_property`
- `get_property`

For example, the following line gets the the `SOURCES` property of the target `MyApp` and assigns it to `MYAPP_SOURCES`:

```
get_property(MYAPP_SOURCES TARGET MyApp PROPERTY SOURCES)
```

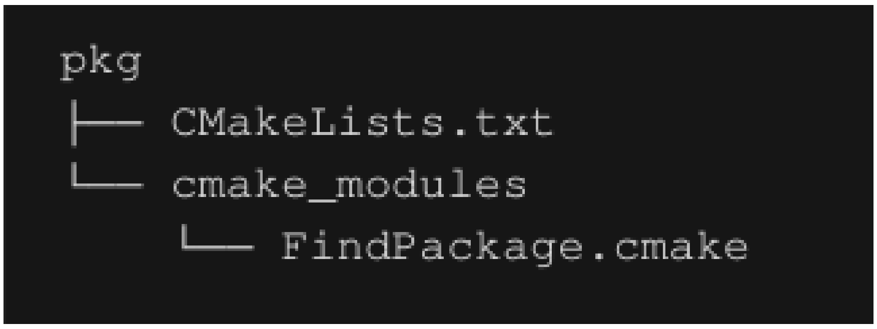
2.2.4 Build Types

There are different build types:

- `Release`: high optimization level, no debug info, code or asserts.
- `Debug`: No optimization, asserts enabled, custom debug (output) code enabled, debug info included in executable (so you can step through the code with a debugger and have address to source-file:line-number translation).
- `RelWithDebInfo`: optimized, with debug info, but no debug (output) code or asserts.
- `MinSizeRel`: same as Release but optimizing for size rather than speed.

In case of missing `FindPackage.cmake` error, one could search for customized script online and put it in the package directory as follow:
In the top-level `CMakeLists.txt`, add this line before `find_package`:

```
list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_LIST_DIR}/cmake_modules)
```


```
pkg
├── CMakeLists.txt
└── cmake_modules
    └── FindPackage.cmake
```

2.3 C++

C++ is a statically typed programming language. On the other hand, Python is a dynamically typed programming language, which implies that developers don't have to think and define the type of all variables with care. C++ allows low-level memory and hardware control like the C language, but with high-level abstraction with classes and structures. With its computational and memory efficiency, the language has usage in game engines, databases, embedded systems, desktop software, etc. When working with Robot Operating System (ROS), the coding languages are C++ and Python. However, for most real-time operations and features, C++ is more preferable.

2.3.1 Memory Management and Pointer

Variable types and sizes

The primitive data types vary in the memory size that they occupy. Be aware that sometimes, the actual memory size also depends on the compiler or the data models. In addition, there are modifiers to the type:

- Signedness
  - `signed` (default) - have signed representation, loosing 1 bit
  - `unsigned` - have unsigned representation
- Size
  - `short` - be optimized for space and will have width of at least 16 bits.
  - `long` - have width of at least 32 bits.

A `boolean` just needs 1 bit, but you can only access byte, thus it still occupies 1 byte, which is 8 bits. We could however store 8 `bool`(s) in 1 byte. Suffixes are used to differentiate types when assigning variables. These suffixes are case-insensitive: i.e., ‘u‘ is equivalent to ‘U‘, ‘UL‘ is equivalent to ‘uL, Ul, ul‘.

Stack and Heap Memory

Allocating on the stack is fast, simple as only one CPU commands. There is a lot happen behind the scene when assigning a value on heap memory. To find the free space, assign, book keeping, .. In case you have something very memory heavy, then you have to use the heap.

Type	Size	Typical Range
boolean	8 bits = 1 byte	$0 - 1$
(signed) char	8 bits = 1 byte	-2^7 to $2^7 - 1$
unsigned char	8 bits = 1 byte	0 to $2^8 - 1$
(signed) int	32 bits = 4 bytes	-2^{31} to $2^{31} - 1$
unsigned int	32 bits = 4 bytes	0 to $2^{32} - 1$
(signed) short int	16 bits = 2 bytes	-2^{15} to $2^{15} - 1$
unsigned short int	16 bits = 2 bytes	0 to $2^{16} - 1$
(signed) long int	32 bits = 4 bytes	-2^{31} to $2^{31} - 1$
unsigned long int	32 bits = 4 bytes	0 to $2^{32} - 1$
(signed) long long int	64 bits = 8 bytes	-2^{63} to $2^{63} - 1$
unsigned long long int	64 bits = 8 bytes	0 to $2^{64} - 1$
float	32 bits = 4 bytes	$3.4E \pm 38$ (7 digits)
double	64 bits = 8 bytes	$1.7E \pm 308$ (15 digits)

Table 2.1: Data types with their size and range.

Data Type	Suffix	Meaning
int	U	unsigned int
int	L	long
int	UL / LU	unsigned long
int	LL	long long
int	ULL / LLU	unsigned long long
double	F	float
double	L	long double

Table 2.2: Suffixes for data types.

- Allocate with the stack:
  `int value = 5;`
- Allocate with the heap:
  ```
  int* hvalue = new int;
  *hvalue = 5;
  ```

new and delete operator

The `new` operator does two things: allocating memory and calling the constructor. The main purpose of `new`, is to allocate memory, on the HEAP specifically. The `delete` operator calls the destructor and then frees the memory.

Additional notes:

- Arrays created with `new []` must be destroyed with `delete[]`.
- Using `new`, the object created remains in existence until you `delete` it. Without using `new`, the object will be destroyed when it goes out of scope.
- Every time you type `new`, type `delete`.

Smart Pointers

Unconscious not deallocating a pointer causes a memory leak that may lead to crash of the program. For languages with Garbage Collection Mechanisms to smartly deallocate unused memory, e.g., Java and C#, programmers don't have to worry about memory leak. C++11 comes up with its own mechanism: Smart Pointer. When the object is destroyed, it frees the memory as well.

With `#include <memory>`:

- `std::unique_ptr` stores one pointer only. We can assign a different object by removing the current object from the pointer.
- `std::shared_ptr` allows more than one pointer pointing to this one object at a time and it'll maintain a Reference Counter using `use_count()` method.
- `std::weak_ptr` also allows more than one pointer pointing at one object at a time, but without a Reference Counter.
- `std::auto_ptr` is replaced by `std::unique_ptr`, with similar functionality, improved security, added features and support for arrays.

Good practice is to use `std::make_shared` as a simple and more efficient way to create an object and a `std::shared_ptr` to manage shared access to the object at the same time.

Let look into an example with these smart pointers. Assuming we have a `class A` with a simple method `show()`:

```
#include <memory>
#include <iostream>

class A
{
public:
  void show()
  {
    std::cout << "Run A::show()" << std::endl;
  }
};
```

Initially, we can define also `std::auto_ptr`. After C++11, this is depreciated.

```
int main()
{
  std::auto_ptr<A> p1(new A);
  p1->show();
  std::cout << "Memory address of p1 " << p1.get() << std::endl;
  std::auto_ptr<A> p2(p1);
  p2->show();
  std::cout << "Memory address of p1 " << p1.get() << std::endl;
  std::cout << p2.get() << std::endl;
}
```

Example define and use `std::unique_ptr`:

```
int main()
{
  std::unique_ptr<A> p1(new A);
  p1->show();
  std::cout << "Memory address of p1 " << p1.get() << std::endl;
}
```

Since std::unique_ptr allows only one pointer, attempt to copy will lead to error. The only way is to transfer the ownership

```
// Error: can't copy unique_ptr
// std::unique_ptr<A> p2 = p1;

// transfers ownership to p2
std::unique_ptr<A> p2 = move(p1);
p2->show();
std::cout << "Memory address of p1 " << p1.get() << std::endl;
std::cout << "Memory address of p2 " << p2.get() << std::endl;
std::cout << std::endl;
```

Example create and use std::shared_ptr

```
int main()
{
  std::cout << "smart_ptr example:" << std::endl;
  std::shared_ptr<A> p3(new A);
  std::cout << "Memory address of p3 " << p3.get() << std::endl;
  p3->show();
  std::shared_ptr<A> p4(p3);
  p4->show();
  std::cout << "Memory address of p3 " << p3.get() << std::endl;
  std::cout << "Memory address of p4 " << p4.get() << std::endl;
  // Returns the number of shared_ptr objects
  // referring to the same managed object.
  std::cout << "Counts: " << p3.use_count() << std::endl;
  std::cout << "Counts: " << p4.use_count() << std::endl;
  // Relinquishes ownership of p1 on the object
  // and pointer becomes NULL
  p3.reset();
  std::cout << "Memory address of p3 " << p3.get() << std::endl;
  std::cout << "Counts: " << p4.use_count() << std::endl;
  std::cout << "Memory address of p4 " << p4.get() << std::endl;
  std::cout << std::endl;
}
```

In practice, it is better to create this pointers with std::make_shared or std::make_unique

```
int main()
{
  // Good practice
  std::shared_ptr<A> p5 = std::make_shared<A>();
  std::unique_ptr<A> p5 = std::make_unique<A>();
}
```

2.3.2 Class Features

Constructor member initializer lists

Normal constructor initialization:

```
Something::Something(int memIn1, double memIn2)
{
  mem1 = memIn1;
  mem2 = memIn2;
  mem3 = 'c';
};
```

There are cases that initialization of data members inside constructor doesn't work and Initializer List must be used: non-static constant data members, reference members, performance reasons, avoid unnecessary call to a default constructor, etc. Initializing member values with initializer lists is better. The initializer list is inserted after the constructor parameters, after the ":" and separated by a ",".

```
Something::Something(int memIn1, double memIn2)
: mem1{memIn1}, mem2{memIn2}, mem3{'c'} {};
```

Structure versus Class

Some features to differentiate between the use of structure and class in C++:

Structure:

- Default access specifier will be `public`.
- The instance of the structure is known as "Structure variable".
- A `struct` is a bundle of several related elements, which needed to be tied up together in a certain context. When a collection of Plain Old Data (POD) is needed.

Class:

- Default access specifier will be `private`.
- The instance of the class is known as Object of the class.
- A `class` can do things with its methods and members.
- Operators to work on new data type can be defined using special methods (over-loading operators).
- One class can be used as the basis for definition of another. If you use inheritance, don't use `struct`, use `class`.
- Declaration of a var of the new class type requires just the name of the class: `classA varA;`, not: `struct structA varA;`

Virtual and Override Specifier

These two concerns with Runtime Polymorphism. Without `virtual`, you get early binding. With `virtual`, you get late binding.

```
class Base
{
public:
  void Method1() { std::cout << "Base::Method1" << std::endl; }
  virtual void Method2() { std::cout << "Base::Method2" << std::endl; }
};
```

```
class Derived : public Base
{
public:
  void Method1() { std::cout << "Derived::Method1" << std::endl; }
  void Method2() { std::cout << "Derived::Method2" << std::endl; }
};

Base* basePtr = new Derived ();
//  Note - constructed as Derived, but pointer stored as Base*
basePtr->Method1();  //  Prints "Base::Method1"
basePtr->Method2();  //  Prints "Derived::Method2"
```

In the above example, `Method1()` is defined without `virtual` specifier, thus, when we call `basePtr->Method1();`, the early binding calls the definition from the `Base` class. On the other hands, `Method2()` is defined in the `Base` class with `virtual` specifier, which implies late binding. Therefore, when `basePtr->Method2();` is called, the method of the `Derived` class is called.

When using `virtual` functions, it is possible to make mistakes while declaring the member functions of the derived classes. Using the `override` identifier prompts the compiler to display error messages when these mistakes are made.

```
class Base
{
public:
  virtual void Method2() { std::cout << "Base::Method2" << std::endl; }
};

class Derived : public Base
{
public:
  // override identifier will give Error for miss-typing Metod2
  void Metod2() { std::cout << "Derived::Method2" << std::endl; } override
};
```

2.3.3 Best practices and tips

Inline Specifier

Inlining concerns with the overhead cost for switching time of small functions. When the `inline` function is called, the whole code of the `inline` function gets inserted or substituted. This substitution is performed by the C++ compiler at compile time. Inlining is only a request to the compiler, not a command. There are cases that the compiler can ignore this request. `inline` functions have advantages but also disadvantages.

```
inline int cube(int s)
{
  return s*s*s;
}
```

For Class definition, you only need to add `inline` when defining it, not when declaring it inside the class.

Explicit Specifier

Specifies that a constructor or conversion function (since C++11) or deduction guide (since C++17) is `explicit`, that is, it cannot be used for implicit conversions and copy-initialization. Example of implicit specifier:

```
class A
{
public:
  A();
  A(int);
  A(const char*, int = 0);
};

int main()
{
  A c = 1;
  A d = "Venditti";
}
```

In this example, though variable `c` is of `class A`, which is not an `int`, it is okay to write initialize `A c = 1;` due to implicit conversion. The same happens with `A d = "Venditti";`. The above program exits without error.

However, the above usage might lead to accidental construction that can hide bugs. Thus, we might want to turn off the implicit conversion with `explicit` specifier. Example of explicit specifier:

```
class A
{
public:
  explicit A();
  explicit A(int);
  explicit A(const char*, int = 0);
};

int main()
{
  A a2 = A(1);
  A a3(1);
  A a4 = A("Venditti");
  A a5 = (A)1;
  A a6 = static_cast<A>(1);
  return 0;
}
```

Keyword auto

Use `auto` to increase readability without creating confusion.

```
// good : auto increases readability here
// v could be array as well
for(auto it = std::begin(v); it != std::end(v); ++it) {}

// No type confusion
auto obj1 = new SomeType<OtherType>::SomeOtherType();
auto obj2 = std::make_shared<XyzType>(args...);
```

Using and Typedef Keywords

Purposes of keyword `using` in C++:

- `using` declarations: brings a specific member from the `namespace` into the current scope.

  ```
  int main()
  {
    using std::cout; // declare cout resolve to std::cout
    cout << "Hello world!"; // no std:: prefix is needed here
    return 0;
  } // the using declaration expires here
  ```

- `using` directive: brings all members from a `namespace` into the current scope.

  `using namespace std;`

  In modern C++, `using` directives generally offer little benefit (saving some typing) compared to the risk. Because `using` directives import all of the names from a `namespace` (potentially including lots of names you'll never use), the possibility for naming collisions to occur increases significantly (especially if you import the `std namespace`).

- Bring a base class method or variable into the current class's scope.
- In C++11, the keyword `using` is used for `type alias`, which is identical to `typedef`. In many cases, using has improving readability, compared to the equivalent `typedef`, especially with pointer and template.

2.4 VSCode

VSCode is a code editor made by Microsoft. It is the most popular Development Environment (2019) [Sta] with many features including support for debugging, syntax highlighting, code completion, snippets, refactoring, Git status, etc. Visual Studio Marketplace [Vsc] offers extensions for numerous coding languages, file types, integration with other tools and platforms, etc.

2.4.1 Hotkeys

Some helpful VSCode hotkeys:

Command	Action
Ctrl + P	Find files
Ctrl + Shift + E	File Explorers
Ctrl + Shift + P	Show all commands
Ctrl + Space	Invoke IntelliSense suggestion
Ctrl + C (without text selection)	Copy entire current line
Ctrl + Shift + K	Delete the entire line
Alt + Move Arrow	Move entire selected line(s) up/down
Ctrl + F	Find words in file
Ctrl + H	Find and replace words in file
Ctrl + Shift + F	Find words in workspace
Ctrl + Shift + H	Find and replace words in workspace
F2	Rename Refactoring
F8	Errors and Warning
Ctrl + Shift+I	Formatting
Ctrl + Shift+[or]	Code folding
Ctrl + ‘	Open Terminal
Ctrl + Shift+M	Open Problems
Ctrl + Shift+X	Extensions
Ctrl + Shift+G	Git
Ctrl + ,	Settings
Ctrl + .	Code actions
Ctrl + B	Close side panels
Ctrl + J	Code bottom panels
Ctrl +	Split editor
Ctrl + 2	Extra tab

2.4.2 Snippet

Example of VSCode snippet:

```
"Snippet purpose": {
 "prefix": "prefix_name",
 "description": "Do something",
 "body": [
    "line 1",
    "line 2",
    "",
    "line 3",
    ""
 ]
}
```

2.5 VIM

Vim [Moo] is a free and open-source text editor for Unix. It is the abbreviation of Vi IMproved, which implies it is an improved clone of vi. It’s a tool designed for coder. It has the capabilities to program commands.

2.5.1 Modes

There are three modes, and their corresponding keyboard to enter:

- `ESC` - Command mode
- `i` - Insert mode
- `r` - Replace mode
- `v` - Visual mode

2.5.2 Basic commands

Commands to save and exit:

- `:q` - close a window, check vim window
- `:qa` - quit all window
- `:q!` - quit without writing
- `:w` - write
- `:wq` - save and exit

Commands for navigation:

- `hjkl` - simple movement
- `wbe` - words: next, beginning, end of word
- `WBE` - words, but seperated by space
- `Shift-HML` - screen top, middle, bottom
- `Ctrl-UD` - scroll up/down
- `Ctrl-G` - tell where you are
- `Ctrl-FB` - scroll faster up/down
- `f[char]` - find char in that line
- `t[char]` - to char in that line. right before that char
- `G` - move to bottom of the file
- gg - move to start of the file
- `:number` - go to line number

Commands for editing:

- `i` - insert mode
- `r` - replace mode, then enter insert mode
- `s` - `xi`, delete char, then enter insert mode
- `d{motion}` - delete motion, e.g.: `dw, de, dd, d$, d0`
- `c{motion}` - change motion, e.g.: `cw, ce, cc, c$, c0`
- `y{motion}` - copy motion, e.g.: `yw, yy, y$, y0`
- `D` - delete the remaining of line, equivalent with `d$`

- `C` - change the remaining of line, equivalent with `c$`
- `A` - append text at the end of line
- `x / X` - delete/backspace character
- `o / O` - insert line below/above and enter insert mode
- `p / P` - paste after/before the cursor
- `u` - undo last command
- `U` - undo all commands in the current line
- `Ctrl-r` - redo command
- `.` - repeat the last command, could be somewhere else
- `q<char>` - start record keystrokes into register `<char>`
- `q` - stop record
- `@<char>` - play recorded keystrokes into register `<char>`
- `@@` - repeat last recording
- `ci(` - change inside "(" bracket
- `da{` - delete around "{" bracket, so include "{" and "}"

Commands to replace:

- `:s/str_1/str_2/` - replace first `str_1` with `str_2` in current line
- `:s/str_1/str_2/g` - replace all `str_1` with `str_2` in current line
- `:%s/str_1/str_2/g` - replace all `str_1` with `str_2` in whole file
- `:2,7s/str_1/str_2/g` - replace all `str_1` with `str_2` in lines `2-7`

Commands to search:

- `/[word]+Enter` - search for `word`
- `n / shift-N` - go to next or previous searched `word`

Commands in Visual modes:

- `v` - visual mode
- `Shift-v` - visual line mode
- `Ctrl-v` - visual block mode

Commands with counts:

- `3w` - 3 words forward
- `5j` - 5 lines down
- `7dw` - delete 7 words

Commands with synced Vim window:

- `Ctrl+w, s` - split horizontal
- `:vsp` - split vertical
- `Ctrl+w, v` - split vertical
- `Ctrl+w, q` - close split window
- `Ctrl+w, w` - switch around window
- `Ctrl+w, hjkl` - switch around adjacent window

3 Working experience

3.1 DevOps

Continuous Integration (CI) is a core practice in DevOps, which are the set up practices to build, test and release the code frequently. Whenever developers commit and push their code to the shared repositories, they trigger a workflow on a CI server that automatically builds and tests the changes. In case of integration failure, the server notifies the developers.

The department robotic pipeline has a well-defined structure, using various tools and platforms, e.g., Artifactory, Jenkins, Github, BitBucket, Jira, Confluence.

3.2 Unit Test and Integration Test

Writing tests is as important as writing code itself. In test-driven reverse engineering, the tests are written first, then the code are developed to pass the tests. Reviewers and users should also read the tests to understand the use of functions, classes.

3.3 Licenses

Licenses concern with how others are allowed to use a piece of code. Having no license file would imply others could neither use nor help with the code. From the corporate point of view, there are three classes of licenses:

- **Permissive OSS licenses:** imply others could freely use the source code, while the copyright holders do not hold liability. Some common licenses of this class are: MIT License (Expat), Apache License 2.0, BSD 2-Clause License, BSD 3-Clause License. The code can be used by corporations for commercial purposes with ease.

- **Strong Copyleft licenses:** require others to also open-source their code. These are turn-offs for corporations. Examples: GNU General Public License v3 (GPL-3), GNU General Public License v2.0 (GPL-2.0), Microsoft Reciprocal License (Ms-RL).

- **Weak Copyleft licenses:** are in the middle of the above two. One should be careful when using code under these licenses. To be more certain, consultation with experienced lawyers is advised. Examples of these licenses are: GNU Lesser General Public License v3 (LGPL-3.0), GNU Lesser General Public License v2.1 (LGPL-2.1).

In practice, working with ROS2, the command `ament_copyright` can help checking and add missing licenses.

```
ament_copyright -h
ament_copyright --add-missing "<name copyright holder>" bsd_3clause --verbose .
ament_copyright --list-licenses
```

In practice, it's good to have an updated `3rd-party-licenses.txt` around. Example of `3rd-party-licenses.txt`:

```
Shipped Third Party Components
======================================

This product contains the following open source components



--------------------------------------------------------------------------
Overview
--------------------------------------------------------------------------

package_name/include/package_name/header.h:

Name:       another_package_name
Version:    2.7.5
URL:        https://github.com/another_package_name
License:    MIT
Copyright: Copyright (c) ...
Comment:    God help me

--------------------------------------------------------------------------
Licenses
--------------------------------------------------------------------------

a) another_package_name

MIT License

Copyright (c), etc.
```

3.4 Coding Style

As a programmer, one will have to write code for others to read, review and modify. Thus, it is crucial to write code in a well-presented manner. Coding style concerns with sets of conventions that improve code readability.

3.4.1 Naming conventions

E.g., in C++:

- File name: match case of class name in file
- Parameters, locals: lower case with underscore, eg., `variable_name`
- Constants: uppercase with underscore, eg., `CONSTANT_NAME`

- Function name: camel case, eg., `functionName`
- Member variables: end with underscore, eg., `member_variable_name_`

3.4.2 Header guards

The style for header guard is to be in uppercase and end with underscore. The `#endif` also has the name of the file guard as inline comment. Example of header guard in `foo_bar_baz.h`:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
code...
#endif // FOO_BAR_BAZ_H_
```

3.4.3 The order of includes

1. Related header
2. C system headers
3. C++ standard library headers
4. Other libraries headers
5. Your project headers

Example of include order in `fooserver.cpp`:

```
#include "foo/server/fooserver.h"

#include <sys/types.h>
#include <unistd.h>

#include <string>
#include <vector>

#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/server/bar.h"
```

3.4.4 Indentation

The common indentation in C++ is two spaces, in Python is four spaces. When using linter, the line length is limited, e.g., `cpplint`'s limit is 80 characters, `pylint`'s limit is 100 characters.

3.4.5 Comments

Comments are necessary to explain the usage of variables, functions, classes, etc. The more complex and non-obvious is the function, method, the more comments there should be. In C++, Doxygen style comments are widely used. It has @tags regarding parameters, return value, exception, etc. Python, there is `docstrings` and function annotations with type hints. Doxygen style comment examples:

```
/**
* Sum numbers in a vector.
*
* This sum is the arithmetic sum, not some other kind of sum that only
* mathematicians have heard of.
*
* @param values Container whose values are summed.
* @return sum of ‘values‘, or 0.0 if ‘values‘ is empty.
*/
double sumInt(std::vector<double> values);

/**
* @brief Sum numbers in a vector
* @param values Container whose values are summed.
* @return sum of ‘values‘, or 0.0 if ‘values‘ is empty.
* @throw Exception something
*/
double sumInt(std::vector<double> values);
```

Python docstrings and function annotation examples:

```
def func(abc, fds=4.6):
    # One-line Docstrings, note the "." at the end
    '''Do sth and return sth.'''

    # Multi-line Docstrings, Google Style (check also Sphinx, Numpy style)
    '''Summary line.

    Args:
        a (int): the dog
        b (float): the duck

    Raises:
        RuntimeError: Out of dog.

    Returns:
        dick (float): the dick
    '''

    pass

print(func.__doc__)
help(func)

def func(abc:'int', fds:'float'=4.6) -> 'list':
    pass
print(func.__annotations__)
```

With ROS2, the `ament_package` [Ame] includes programs to check for coding style in CMake, C++, Python, XML. The commands are:

```
# automatically fix them.
ament_uncrustify --reformat
# see the output in isolation
ament_cpplint
ament_cppcheck
```

4 Vacuum robot

The dream of having a robot in our houses doing all tedious work has been around for a long time. Bill Gates [Bil07] predicts that robots will soon be in every home. Despite the huge need and significant engineering efforts, these robots are available to the market slower than expected. This chapter overviews domestic cleaning robots in general, vacuum robots and one of the engineering challenge, i.e., coverage path planning.

4.1 Domestic Cleaning Robots

Cleaning robots have huge business potential. There are countless possible domestic applications for robots: floor vacuuming, mopping, kitchen and bathroom cleaning, dish washing, lawn mowing, etc. Nearly half of those applications involve cleaning. Within all cleaning tasks, around one eighth relates to vacuuming [CT13]. For professional cleaning service, the market share in Europe alone was worth billions of USD [Sch95]. In 2020, 21.6 millions units of domestic robots, worth 5 billions USD in sales value, are sold worldwide [Int20]. Despite this potential and around 20 years of research and development, in terms of efficiency and cost, cleaning robots could not yet fully replace the need of manual labour works [SK16].

Having attracted a lot of academic attention for a long time, cleaning robots are still technically challenging. Prassler et al. [Pra+16] list related technical problems: absolute positioning, dynamic environments, human-robot interaction, power supply, etc. Separately, all of these problems either have proposed approaches or tested solutions. However, they usually work in theoretical rather than real-world conditions, especially from a cost point-of-view. Companies often find themselves either reinvent or adapt to their own industrial needs. [Pra+16]

Kim et al. [Kim+19] give an overview on the directions of recent researches for future cleaning robots. While most of these current models are still constrained to two-dimensional workspace, some researchers focus on multi-purpose cleaning robots for the future (Fig. 4.1). Their goal is to handle furnitures, non-flat surfaces and diverse tasks. These advance robots would involve more learning-based approaches: learning from demonstration, supervised learning, reinforcement learning, etc. [Kim+19]

4.2 Vacuum Robots

Comparing to other domestic chores, floor vacuuming is more challenging. In other tasks, e.g., cleaning of pool, solar panel or window, the robots usually handle static and known space. The environment is mostly two-dimensional, sometimes requires fixed obstacle avoidance. Vacuum robots have to deal with more diverse environment settings, e.g., unknown

Figure 4.1: Multi-purpose cleaning robot [Oka+06].

map, dynamic obstacles, interaction with human. In addition, many customers lack willingness to make physical changes to their rooms for the robots [Vau+14].

4.2.1 Design Characteristics

Early 2000s, some major ground works on coverage problem were laid down [HCH86; Zel+93; Cho00; GR02]. Before the end of 2005, many major companies released their first models of domestic cleaning robots [SK16]. Most of these models and even current ones still have a lot in common:

- Shape: Round-shaped
- Dimension: Varies in height, diameter around 30-40 cm
- Cleaning technology: One suction pump (as the main End effector (EE) tool) and side brush (as supporting tool)
- Common sensors: Sonar, bumper and cliff (stair avoidance) sensors
- Coverage strategy: Bang and bounce (random motion), contour following

Most early models have round shape. The suction pump is limited between the two wheels. Another used design is D-shaped robot (Fig. 4.2), enabling wider brush and more corner coverage.

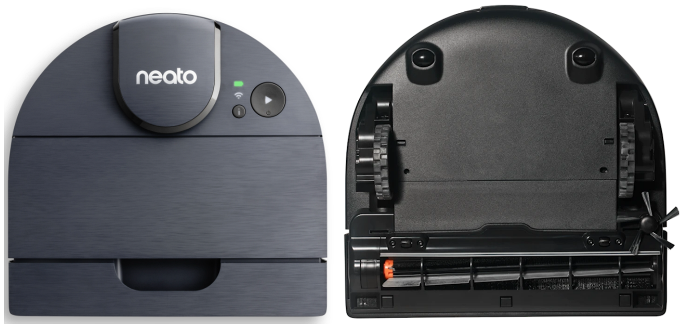


Figure 4.2: Neato Robotics: Neato D8 [Nea21].

Even though it is better to have knowledge of surrounding environment, domestic robots generally have the minimum amount of sensors. A $20 sensor is very expensive for a robot that should cost around $300-400. However, there are still exceptional cases. Costing $3400 , Ottoro has two digital cameras, 12 pairs of ultrasonic sensor and highly sensitive air bumper (Fig. 4.3). With all state-of-the-art technologies, the robot could identify its position with the accuracy of ±3cm and follow better coverage strategies. [SK16]

Figure 4.3: Hanool Robotics: Ottoro [Han21].

Early robot models use simple approaches to coverage problem, i.e., *Bang and Bounce*. With only a bumper sensor at the front, this strategy changes the robot direction whenever it hits an obstacle. Despite not guaranteeing complete coverage, this strategy converges to good space coverage after a large amount of time (Fig. 4.4) [LLZ08]. Some models integrate with contour following, achieved by adding sensor on the robot side for wall detection. As users are expecting more intelligent robots, a random motion is hardly appealing any longer [Goe+13].

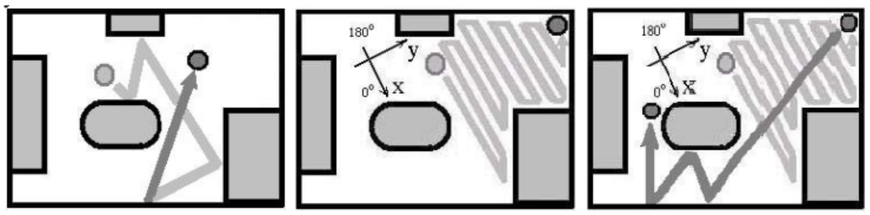


Figure 4.4: Random and fixed motion mode [LLZ08].

4.2.2 Cost

Whether a cleaning robot would be a business success or failure depends heavily on its cost. Early models, with the costs from €800-1750, were 10-20 times more expensive than a high-quality vacuum cleaner. Despite the high cost, many were frequently trapped by electrical cables and need monthly repairs. This led to the lack of trust from customers in the autonomy of these robots [Vau+14]. The first robot successfully entering the domestic market was Roomba from iRobot. Its success was not due to its performance, rather the affordable price, i.e. €350. To take advantage of this market potential, next generation robots should consider not only engineering improvements, but also customer perceptions. [Gut+12; RE15; SK16]

4.3 Coverage Path Planning

4.3.1 Introduction

Coverage Path Planning (CPP) is the problem of finding the most optimal path to fully cover an area. Given a robot description and a map to cover, the output is a robot trajectory. Huang et al. [HCH86] first describe the problem and its usage for robotic lawnmowers. Early seminal works formalize the coverage problem and propose new algorithms that prioritize coverage completeness [Lat91; Zel+93; Cho00]. In recent years, researchers have focused on other aspects of the coverage problem, e.g., extending it to dynamic or 3D environments, accounting for kinematic constraints, planning for several robots.

One application of coverage algorithm is for vacuum robots, which has huge business potential [Sch95; Int20]. The market is currently very competitive with many technological innovations and diverse robot approaches. Traditional round-shaped robot models are easy with to plan with, but have trouble cleaning along the floor edges and corners thoroughly. Other products try to clean better along the edges by resorting to a D-shaped robot, although it is more difficult to navigate with.

In practice, two issues often arise when planning coverage paths that have often been abstracted away in the literature: the need to account for the precise robot shape with EE placement, and a lack of unified metrics for algorithm performance. My thesis examines in details these two issues for robot with asymmetric EE.

4.3.2 Classification

Classification of CPP takes form in many criteria, the most common ones and their corresponding classes are as follows:

- Based on problem characteristics:
  - Offline or Online: Offline algorithm assumes full prior knowledge of a static environment. Online algorithm do not make that assumption. As most online approaches utilize data from real-time sensor, they are also known as sensor-based coverage algorithms.
  - With or without kinematics constraints: whether the robots could easily rotate in place or have a minimum turn radius.
- Based on solution characteristics:
  - Heuristics or Complete: whether the algorithms guarantee complete coverage of space.
  - Type of environment decomposition: exact, approximate or semi-approximate cellular decomposition.

This subsection introduces some well-known coverage approaches.

Trapezoidal Decomposition CPP

Latombe [Lat91] introduces the simplest offline exact cellular decomposition for 2D map. He assumes that map and obstacles are polygonal. When touching a obstacle vertex point, the swiping line divides the map into trapezoidal cells. One downside of this decomposition approach is to generate some redundant cells. E.g., in Fig. 4.5, the 9th and 11th cells can be grouped to one single cell. The less nodes there are, the easier it is to solve the associated Travelling Salemans Problem (TSP).

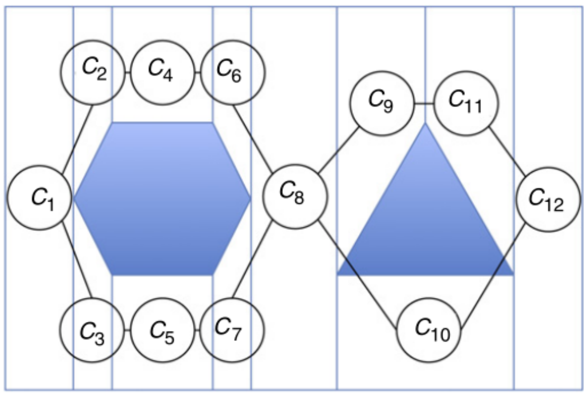


Figure 4.5: Trapezoidal decomposition with polygonal space and obstacles [**galceran2013ras**].

Boustrophedon Decomposition CPP

Choset [Cho00] reduces the number of unnecessary cells. New cells are created only in an *event*, when a cell is split into many or when cells are merged into one (Fig. 4.6). In such *event*, the intersection point between the swiping line and obstacle boundary is a *critical point*. The word *boustrophedon* implies the simple zig-zag coverage pattern. The author also notes that using boustrophedon pattern alone is prone to incomplete coverage. The robot specifically misses areas around the obstacle perimeters, which are not parallel to the moving direction.

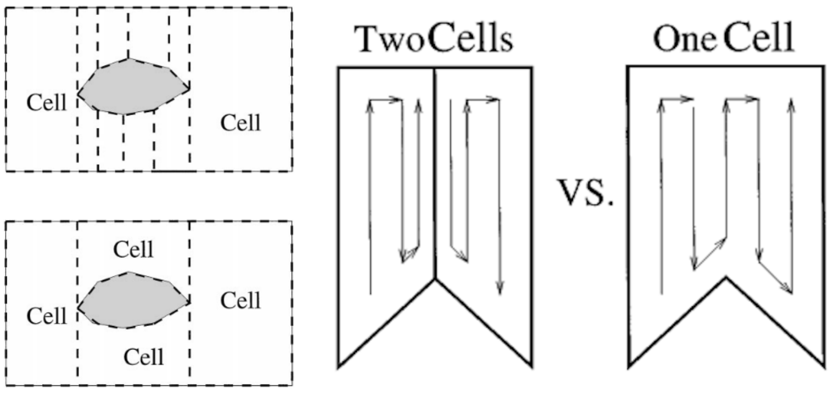


Figure 4.6: Having less cells by creating new ones only in split or merge event.

Decomposition in terms of Critical Points of Morse Functions

Previous presented approaches decompose the map using a line. Choset et al. [Cho+00] generalize the shape of the slice by changing its functional definition, called Morse function. For Boustrophedon Decomposition, the slice function of a line is $h(x) = x$, where x is the first coordinate in 2D space. Different Morse functions induce different cell shapes and their corresponding coverage patterns (Fig. 4.7). Different problem settings may fit better with one coverage pattern than the others. Acar, Choset, and Atkar [ACA01] later develop an online CPP algorithm using Morse function based decomposition.

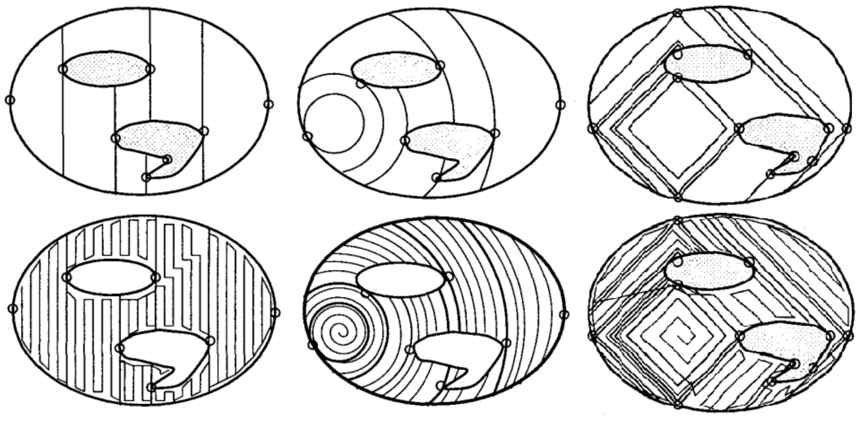


Figure 4.7: Boustrophedon, spiral, diamond cell decompositions and coverage patterns [Cho+00].

Wavefront CPP Algorithm

Zelinsky et al. [Zel+93] use a completely different branch of approaches to CPP. They adopt their approach for Path Planning (PP) problem [Zel91], to create a numeric potential field over the map using the distance transform. The distance wave increases in value as it spreads from goal point to neighbouring grid cells, until every cell is reached. If the robot follows a gradient-descent path, it is the shortest path from start to end point. On the other hands, if the robot follows a gradient-ascent path, it visits all cells and covers the whole map (Fig. 4.8). To reduce high number of turns in *distance transform*, the authors propose to add the distance from walls and obstacles to the potential field, resulting the *path transform* trajectory. The approach is offline and allows start and end points to be specified.

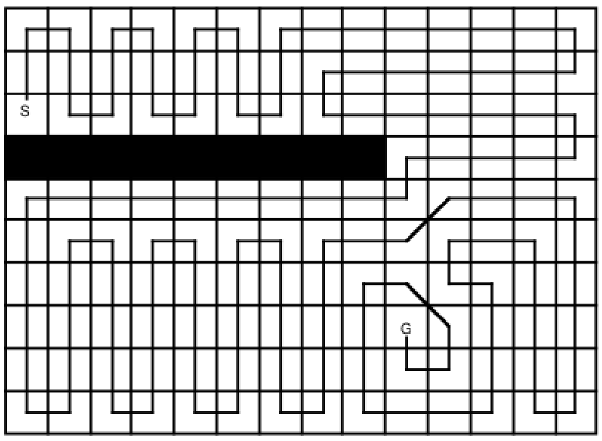


Figure 4.8: Complete coverage path.

Spiral-STC Grid-based CPP Algorithm

Gabriely and Rimon [GR02] propose an online algorithm guaranteeing complete coverage of all free accessible subcells. They first divide the map into large cells, and each one of them contains four small subcells. At each position, the robot explores neighbouring large cell in a certain direction (counter clockwise). As large cells are visited, the spanning tree connecting them extends a new branch. This process iterates until the robot visits all possible large cells. In this exploration stage, the robot has travelled only on one side of the spanning tree.

Then, the robot turns back and traverses on the other side of the tree to cover the rest of the map (Fig. 4.9).

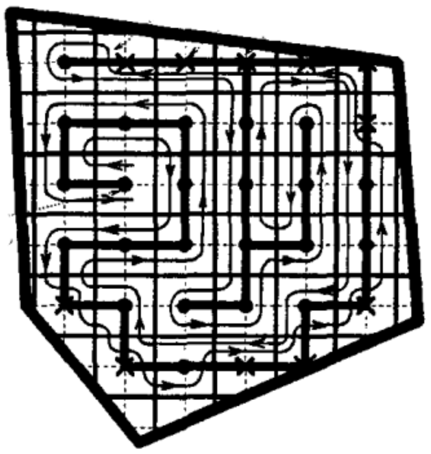


Figure 4.9: An execution of full Spiral-STC.

Neural Network Approach to CPP

The online CPP approach by Yang and Luo [YL04] is inspired by neural networks. Each cell in the grid map also represents a neuron in the network. The receptive field of each cell is only its 8 neighbours. Each neuron has a shunting dynamic equation. Certain thresholds over the shunting inputs allow the positive neural activities (pull activities from free spaces) to spread across the map, while limit the negative neural activities (push activities from obstacles) to stay in each local area. To limit the number of turns, the robot path takes both neural activity and previous poses into account. In the end, it also results in a potential field, similar to the one in Zelinsky et al.'s work (Fig. 4.10). This approach can deal with dynamic and unstructured obstacles. However, unlike other online algorithms, e.g., [But00; ACA01; GR02], for the network setup, the map size must be known a priori.

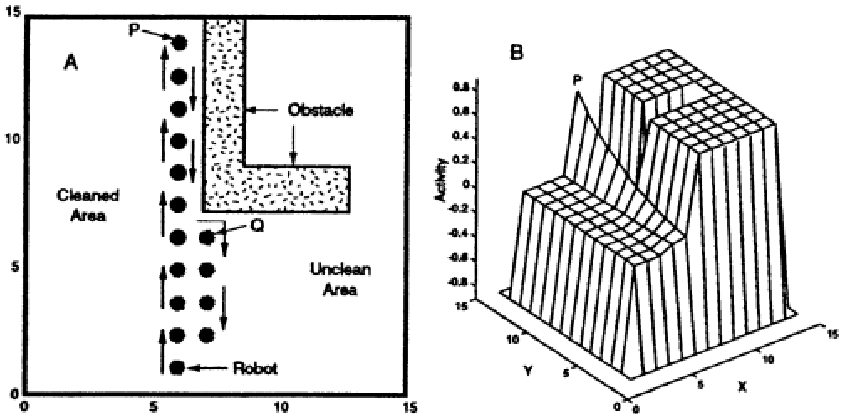


Figure 4.10: Robot path and the activity landscape [YL04].

Convex Sensor Placement Problem to CPP

Arain et al. [Ara+15] solve the convex sensor placement problem by formulating a novel directed graph on top of the grid map. Each cell has four nodes, corresponding to four poses. Each node has connections with other nodes in the same grid cell and one node in the neighbouring cell of the same direction. E.g., if the current node represents the pose going up, it connects with the node going up in the above grid cell, if that exists (Fig. **??**). As an area inspection problem, the range of the end-effector (a gas detection sensor) is much greater than the size of each grid cell. The coverage problem is similar to the Covering Salemans Problem (CSP), which doesn't require to visit all nodes in the graph. The authors find a path traversing the graph using iterative convex relaxation method (Fig. 4.11).

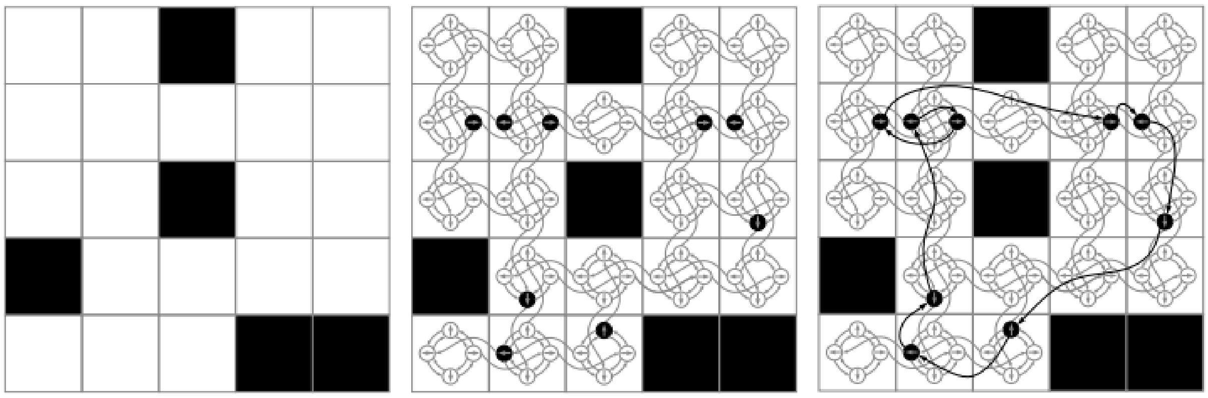


Figure 4.11: Trajectory from traversing the graph.

Turn-minimizing Coverage

The offline algorithm of Vandermeulen, Groß, and Kolling [VGK19] resembles the line-sweep-based decomposition [Hua01], but for grid-based approach. The authors consider the problem as a TSP instance. A *rank* is similar to a sweep line and is a node in the TSP. The robot covers the map by visiting all interior and perimeter ranks (Fig. 4.12). Each rank has three vertices: two end points and a midpoint. As the cost from the midpoint vertex to vertices in other ranks is infinity, it enforces the robot to go from one end to the others within a rank.

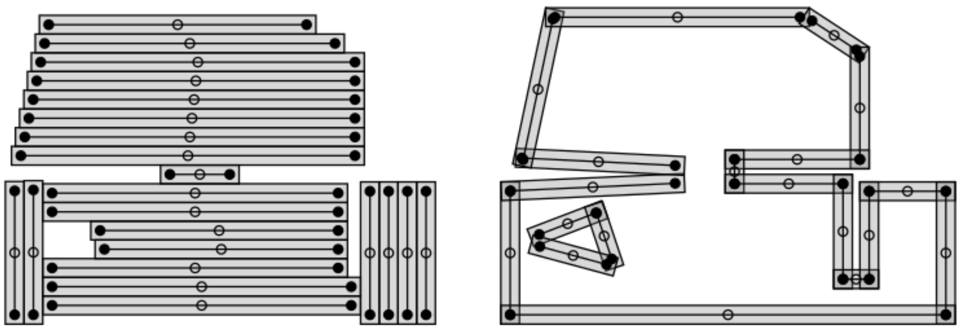


Figure 4.12: Interior and perimeter ranks. [VGK19].

5 Conclusion

The internship at the Bosch CR/AAS3 department was my first experience in a large and professional company. It opens up my point of view about the complete process of software development. I still have many gaps of knowledge to fill, but unlike before, I have more awareness of the direction to follow. During the internship, I mainly code and work with software. The Corona situation made it harder to work and cooperate with others. Regardless, it was a great opportunity, allowing me to learn, experience and prepare for the future.

Bibliography

[ACA01] Ercan U. Acar, Howie Choset, and Prasad N. Atkar. “Complete sensor-based coverage with extended-range detectors: a hierarchical decomposition in terms of critical points and Voronoi diagrams”. In: *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)* 3 (2001), pp. 1305–1311. DOI: `10.1109/IROS.2001.977163`.

[AlD] AlDanial. *cloc*. `https://github.com/AlDanial/cloc`. Accessed: 2022-03-02.

[Ame] *ament*. `https://github.com/ament`. Accessed: 2022-03-02.

[Ara+15] Muhammad Asif Arain et al. “Efficient measurement planning for remote gas sensing with mobile robots”. In: *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)* (2015), pp. 3428–3434. DOI: `10.1109/ICRA.2015.7139673`.

[Bil07] Bill Gates. “A robot in every home”. In: *Scientific American* 296.1 (2007), pp. 58–65.

[Bos] *Robert Bosch GmbH*. `https://www.bosch.com/`. Accessed: 2022-03-02.

[But00] Zack J. Butler. *Distributed Coverage Of Rectilinear Environments*. Carnegie Mellon University, Oct. 2000.

[Cho] Junegunn Choi. *command-line fuzzy finder*. `https://github.com/junegunn/fzf`. Accessed: 2022-03-02.

[Cho00] Howie Choset. “Coverage of Known Spaces: The Boustrophedon Cellular Decomposition”. In: *Autonomous Robots* 9.3 (Sept. 2000), pp. 247 –253.

[Cho+00] Howie Choset et al. “Exact cellular decompositions in terms of critical points of Morse functions”. In: *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)* 3 (2000), pp. 2270–2277. DOI: `10.1109/ROBOT.2000.846365`.

[CT13] Maya Cakmak and Leila Takayama. “Towards a comprehensive chore list for domestic robots”. In: *Proc. of the Int. Conf. on Human-Robot Interaction (HRI)* (2013), pp. 93–94.

[Goe+13] Dhiraj Goel et al. “Systematic floor coverage of unknown environments using rectangular regions and localization certainty”. In: *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)* (2013), pp. 1–8. DOI: `10.1109/IROS.2013.6696324`.

[GR02] Yoav Gabriely and Elon Rimon. “Spiral-STC: an on-line coverage algorithm of grid environments by a mobile robot”. In: *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)* 1 (2002), pp. 954–960. DOI: `10.1109/ROBOT.2002.1013479`.

[Gre] Gregory. *sd*. `https://github.com/chmln/sd`. Accessed: 2022-03-02.

[Gro88] Stephen Grossberg. “Nonlinear neural networks: Principles, mechanisms, and architectures”. In: *Neural networks* 1.1 (1988), pp. 17–61.

[Gut+12] Jens-Steffen Gutmann et al. “The social impact of a systematic floor cleaner”. In: *Proc. of the IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO)* (2012), pp. 50–53.

[Han21] Hanool Robotics Corp. *Ottoro.* 2021. URL: `https://hanool.en.ec21.com/Cleaning_Robot_Vacuum_Cleaner_Floor--2582659_2582660.html` (visited on 09/30/2021).

[HCH86] Yuyu Huang, Z. Cao, and E. Hall. “Region filling operations for mobile robot using computer graphics”. In: *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)* 3 (1986), pp. 1607–1614.

[HH52] Alan L. Hodgkin and Andrew F. Huxley. “A quantitative description of membrane current and its application to conduction and excitation in nerve”. In: *The Journal of physiology* 117.4 (1952), pp. 500–544.

[Hua01] Wesley H. Huang. “Optimal line-sweep-based decompositions for coverage algorithms”. In: *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)* 1 (2001), pp. 27–32.

[Int20] International Federation of Robotics. *World Robotics 2020 Report.* 2020. URL: `https://ifr.org/ifr-press-releases/news/record-2.7-million-robots-work-in-factories-around-the-globe` (visited on 10/08/2021).

[Kim+19] Jaeseok Kim et al. “Control strategies for cleaning robots in domestic applications: A comprehensive review”. In: *Intl. Journal of Advanced Robotic Systems* 16.4 (2019), p. 1729881419857432.

[Lat91] Jean-Claude Latombe. *Robot Motion Planning.* USA: Kluwer Academic Publishers, 1991. ISBN: 0792391292.

[LLZ08] Yu Liu, Xiaoyong Lin, and Shiqiang Zhu. “Combined coverage path planning for autonomous cleaning robots in unstructured environments”. In: *Proc. of the World Congress on Intelligent Control and Automation* (2008), pp. 8271–8276. DOI: `10.1109/WCICA.2008.4594223`.

[ME85] Hans Moravec and Alberto Elfes. “High resolution maps from wide angle sonar”. In: *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)* 2 (1985), pp. 116–121.

[Mer14] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[Moo] Bram Moolenaar. *Vim the Editor.* `https://www.vim.org/`. Accessed: 2022-03-02.

[Nea21] Neato Robotics. *Neato D8 Intelligent Robot Vacuum.* 2021. URL: `https://shop.neatorobotics.com/products/neato-d8` (visited on 10/01/2021).

[Oka+06] Kei Okada et al. “Vision based behavior verification system of humanoid robot for daily environment tasks”. In: *2006 6th IEEE-RAS International Conference on Humanoid Robots* (2006), pp. 7–12.

[Peta] David Peter. *bat.* `https://github.com/sharkdp/bat`. Accessed: 2022-03-02.

[Petb] David Peter. *fdfind.* `https://github.com/sharkdp/fd`. Accessed: 2022-03-02.

[Pra+16] Erwin Prassler et al. “Domestic robotics”. In: *Springer handbook of robotics*. Springer, 2016, pp. 1729–1758.

[Ran] *ranger*. `https://ranger.github.io/`. Accessed: 2022-03-02.

[RE15] Lambèr Royakkers and Rinie van Est. “A literature review on new robotics: automation from love to war”. In: *International Journal of Social Robotics* 7.5 (2015), pp. 549–570.

[Sag] Benjamin Sago. *Exa*. `https://the.exa.website/`. Accessed: 2022-03-02.

[Sch95] M Schofield. “Cleaning robots, Service Robots Int”. In: *J* 1.3 (1995), pp. 11–16.

[SK16] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. Springer, 2016.

[Sta] StackOverflow. *Developer Survey Results 2019*. `https://insights.stackoverflow.com/survey/2019`. Accessed: 2022-03-02.

[Sé] Denys Séguret. *broot*. `https://github.com/Canop/broot`. Accessed: 2022-03-02.

[Vau+14] Florian Vaussard et al. “Lessons learned from robotic vacuum cleaners entering the home ecosystem”. In: *Journal on Robotics and Autonomous Systems (RAS)* 62.3 (2014), pp. 376–391.

[VGK19] Isaac Vandermeulen, Roderich Groß, and Andreas Kolling. “Turn-minimizing multirobot coverage”. In: *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)* (2019), pp. 1014–1020. DOI: `10.1109/ICRA.2019.8794002`.

[Vsc] *VSCode Marketplace*. `https://marketplace.visualstudio.com/`. Accessed: 2022-03-02.

[XAM] XAMPPRocky. *tokei*. `https://github.com/XAMPPRocky/tokei`. Accessed: 2022-03-02.

[YL04] Simon X. Yang and Chaomin Luo. “A neural network approach to complete coverage path planning”. In: *IEEE Trans. on Systems, Man, and Cybernetics* 34.1 (2004), pp. 718–724. DOI: `10.1109/TSMCB.2003.811769`.

[Zel91] Alexander Zelinsky. “Environment exploration and path planning algorithms for a mobile robot using sonar”. In: *PhD Thesis, Wollongong University* (1991).

[Zel+93] Alexander Zelinsky et al. “Planning Paths Of Complete Coverage Of An Unstructured Environment By A Mobile Robot”. In: *Proc. of the Int. Conf. on Advanced Robotics (ICAR)* 13 (1993), pp. 533–538.