Politecnico di Torino

Master’s Degree in Mechatronic Engineering

Master’s Thesis

Hardware acceleration for robotic perception

Supervisors

Prof. Marcello Chiaberge

Ing. Andrea Merlo

Ing. Patrick Roncagliolo

Candidate

Federica Parisi

289819

April 2023

Abstract

A significant portion of current and future space missions involves exploring unstructured environments, such as the surfaces of the Moon and Mars. These environments are characterized by complex and often unpredictable terrain, which presents unique challenges for autonomous robotic systems. Perception and mapping strategies play a crucial role in ensuring safe and efficient navigation in these environments. As a result, much research has been devoted to developing advanced techniques for perception and mapping in unstructured environments, and this is an active area of study in the field of robotics and space exploration. In particular, the increasing demand for real-time and reliable robotic perception systems has motivated the development of hardware acceleration algorithms. Hardware acceleration consists of the use of special-purpose hardware, which is specially designed to perform specific functions more efficiently than software running on a general-purpose CPU. Some of the advantages of hardware against software include speedup, lower power consumption, lower latency, and increased parallelism, at the cost of longer development times and reduced ability to update the designs after manufacturing. In the context of robotic perception, these algorithms aim to speed up the processing of visual and sensory data, allowing robots to make quick and accurate decisions in dynamic environments. For this purpose, hardware accelerators such as Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGA) and Application-Specific Integrated Circuits (ASICs) have been adopted. This Master's thesis focuses on the implementation of a hardware acceleration algorithm for the calculation of the surface represented by a point cloud. The surface can be determined through the computation of its normal vectors, which provide valuable information about the surface shape. In particular, the thesis work is centered on the development of a hardware unit that exploits the Principal Component Analysis (PCA). Indeed, the PCA can be used to find the principal directions of a dataset that, in the case of a point cloud, returns information about the vectors that are normal to the surface. The design of this computational unit was carried out to be implemented on a FPGA board. The results of this study demonstrate the feasibility of using hardware acceleration algorithms in robotic perception and provide insights into the trade-offs involved in the design of such systems.

Contents

Chapter 1

Robotic perception techniques for planetary rovers

Over the past five decades, robotic platforms have experienced a significant growth in their usage in planetary exploration missions, spanning a diversity of technologies, such as orbiting spacecrafts, space telescopes, stationary landers [1]. One of the most important sources of exploratory information is represented by planetary rovers, whose emphasis is increasing since this kind of robots is considered as the key to detailed planetary exploration because of the capability to move to different locations for wider area exploration. Indeed, planetary rovers are uniquely useful for almost all types of planetary missions on planets with solid surfaces ranging from small bodies, such as asteroids and comets, to the moons of gas giants or to terrestrial-type planets such as Mars [2]. Moreover, some of the other advantages of planetary rovers are the high degree of mobility, the ability of physical experimentation, the autonomous navigation, and the microscopic level of observations. These skills have been developed in the past couple of decades, during which planetary rovers have become increasingly complex and intelligent, employing a range of onboard sensors that enhance their autonomous capabilities. Concerning what has been stated, it is fundamental the role played by robotic perception algorithms. Robotic perception refers to the ability of robots to sense and interpret the environment around them. It is a critical component of autonomous robots, as it allows the robot to understand and interact with its surroundings. Some of the main techniques used in robotic perception are:

1. Computer Vision: a field of study that focuses on enabling computers to interpret and understand visual information. In robotics, computer vision is used to analyze and understand the images captured by cameras, such as depth cameras, RGB cameras, and stereo cameras. This information is then used to

determine the position, orientation, and movement of objects in the environment.

2. Lidar (Light Detection and Ranging): a technology that uses laser light to measure distances and generate 3D maps of the environment. Lidar is commonly used in robotics for perception tasks such as obstacle avoidance, navigation, and object recognition.
3. Radar (Radio Detection and Ranging): a technology that uses radio waves to determine the position, velocity, and distance of objects in the environment. It is often used in robotics for tasks such as obstacle detection, navigation, and mapping.
4. Sonar (Sound Navigation and Ranging): a technology that uses sound waves to measure distances and determine the shape and location of objects in the environment. It is commonly used in robotics for underwater perception tasks, such as navigation, object detection, and mapping.
5. Inertial Measurement Units (IMUs): sensors that measure the orientation and acceleration of a robot. They are often used in robotics to track the position and orientation of the robot, as well as to provide feedback for control and navigation algorithms.

1.1 Vision perception for planetary rovers

For what regards planetary rovers, sophisticated vision systems, supported by onboard software, have been crucial in expanding their autonomous capabilities. In this context, the NASA's Mars rover Curiosity pioneered the autonomous selection of rock targets for scientific analysis by its laser and telescopic camera suite, Chemistry and Camera (ChemCam), using the Autonomous Exploration for Gathering Increased Science (AEGIS) software:

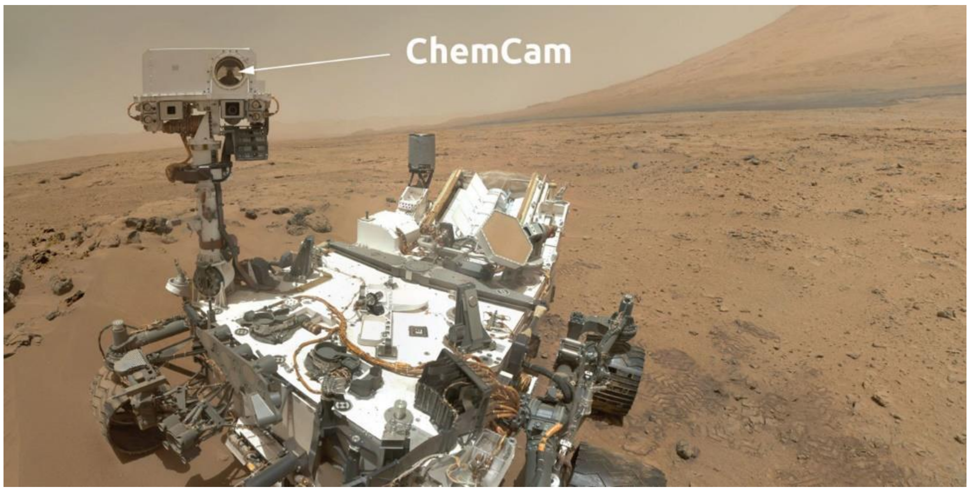


Figure 1 ChemCam mounted on Curiosity rover

This is just one of the many capabilities that complex vision systems provide on board planetary rovers. In fact, onboard vision systems have become critical components for rover autonomy in performing complex tasks, such as high-level surface mapping and relative localization using topological vision data, low-level visual feature detection, recognition and landmark tracking, and complex scientific procedures such as identification of the chemical compositions of Martian soil. It is possible to make a distinction in the vision techniques, based on the type of sensor used to capture scene data: cameras or LiDARs.

1.1.1 Cameras

In past and current missions, most of rovers employs cameras for terrain perception. In general, it is possible to distinguish between depth cameras and RGB cameras, which are all image acquisition devices used in computer vision, but they differ in the information they capture:

1. Depth cameras, also known as range cameras or 3D cameras, directly capture depth information of the object, providing a depth map of the surrounding environment. Depth cameras capture images in which each pixel contains information about the distance to the object in the scene. They use a variety of techniques to measure depth, including structured light, time-of-flight, and stereo vision. Structured light depth cameras project a

pattern of light onto the scene and use the distortion of the pattern to compute the depth of each pixel. Time-of-flight depth cameras measure the time it takes for a pulse of light to travel from the camera to the object and back and use this time to calculate the distance. Stereo vision depth cameras use two cameras to capture the scene from different viewpoints and use the differences in the images to triangulate the depth of each pixel. Depth cameras are useful in a variety of applications, such as robotics, gaming, and augmented reality.

2. RGB cameras, also known as color cameras, capture images in the visible spectrum of light. RGB cameras capture three color channels (red, green, and blue) and a brightness channel, also known as luminance. The camera lens focuses the incoming light onto a sensor, which is typically a charge-coupled device (CCD) or a complementary metal-oxide-semiconductor (CMOS) sensor. The sensor is made up of millions of tiny light-sensitive elements called pixels, which convert the incoming light into electrical signals. The camera then processes these signals to create a digital image. Each pixel in an RGB camera is sensitive to a specific range of wavelengths of light, corresponding to the red, green, and blue color channels. By combining these three - color channels, the camera can capture a wide range of colors and shades of brightness. The brightness channel, or luminance, is often calculated as a weighted average of the three - color channels. RGB cameras are widely used in mobile devices, digital cameras, and surveillance systems due to their low cost and wide availability.

Often, depth cameras can be used in conjunction with RGB cameras to provide additional information about the environment. For example, depth information can be used to separate objects in the scene from the background or to apply depth-based effects to images or videos.

In past and current planetary missions, stereo vision is considered as the baseline method for scene reconstruction and perception for planetary rovers. In this scenario, the pioneer is the rover Soujourner, which was landed by NASA on Mars in the 1997 Mars Pathfinder mission. The rover could navigate through a simple light-stripe sensor that measured twenty – five elevation point in front of the rover.

The lander had a multispectral stereo camera pair on a pan/tilt mast about 1.5 m high. The processing of the stereo imagery was performed on Earth by JPL's real-time stereo algorithm and it were produced excellent maps of terrain around the lander for rover operators to use in planning the mission. In this way, the stereo algorithm performance was validated with real Mars imagery.

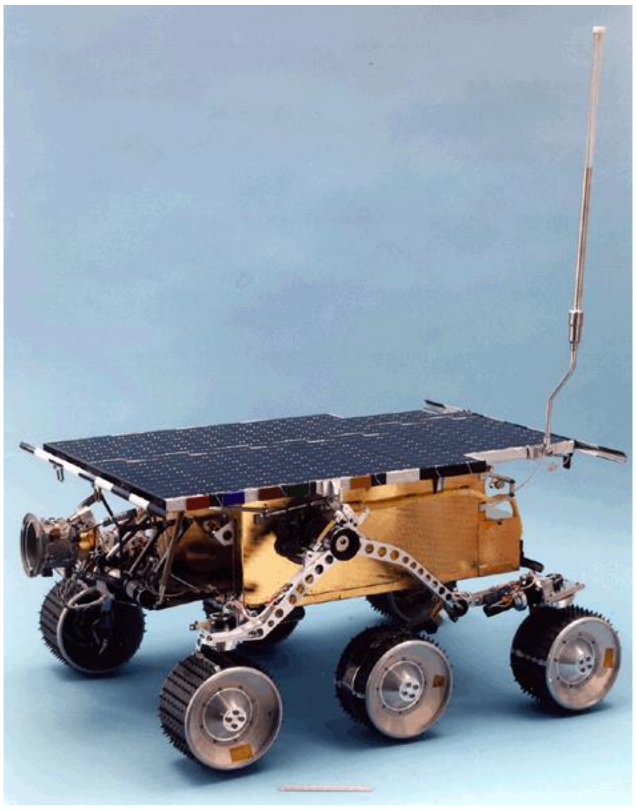


Figure 2 Soujourner rover of the 1997 NASA's Mars Pathfinder mission

Other examples of the usage of stereo vision algorithm are given by the NASA's twin Mars Exploration Rovers (MER), that are Spirit and Opportunity. With respect to Soujourner, they were designed to accomplish more robust navigation tasks, such as obstacle detection and avoidance. For this reason, MER rovers have been equipped with three sets of stereo camera pairs: one pair of "hazcams" (hazard cameras) looking forward under the solar panel in front, another pair of hazcams looking backward under the solar panel in the back and a pair of "newcams" (navigation cameras) on the mast.

Figure 3 The twin Mars Exploration Rovers (MER), Spirit and Opportunity

For what regards future missions, the ESA's ExoMars rover, whose launch is scheduled for 2028, makes use of a perception system that uses a pair of stereo images to generate a disparity map. In detail, the rover is equipped with the so-called Panoramic Camera System (PanCam), which consists of two wide-angle stereo cameras and a third high-resolution camera, used for capturing the surrounding terrain and for navigation. Over the past few decades, planetary rover missions have demonstrated and validated the viability of using stereo cameras and stereopsis as the primary technology for onboard 3D perception. This approach has several advantages, including its solid-state design, which makes it more mechanically robust and durable. In addition, research has shown that the Martian terrain offers enough textural information to support stereo vision almost anywhere on the planet. Several algorithms have been developed that can perform stereopsis and produce accurate and dense range imagery at a sufficient speed using the available computing resources. As a result, this approach has been widely regarded as the best trade-off between cost, risk, and performance for 3D terrain perception on planetary rovers.

1.1.2 LiDARs

Light Detection and Ranging (LIDAR) technology is commonly used by terrestrial rovers covering very long distances. The functioning of LiDAR is based on measuring the time – of – flight of a laser pulse between emission and return of the reflected signal. The laser pulse is emitted by the LiDAR device and, once it reaches the surface of the object, it is reflected and returned to the LiDAR sensor. The LiDAR sensor detects the time taken by the laser pulse to return and, using the speed of light as a constant, it calculates the distance from the object. In space applications, LiDAR is used for spacecraft assistance with rendezvous and docking, depth estimation and mapping, scientific analysis and geological surveying [1]. An example of employment of the LiDAR sensor is the mission OSIRIS-Rex, aimed to map the carbonaceous asteroid Bennu for the purpose of studying its physical and chemical properties. The OSIRIS-REx Laser Altimeter (OLA) is the LiDAR sensor involved in the 3D mapping of asteroid Bennu's shape and it has already completed all its requirements for the OSIRIS-REx mission. OLA's scans of Bennu's surface were used to create the high-resolution 3D global maps of Bennu's topography that were crucial for selecting the primary and backup sample collection sites.

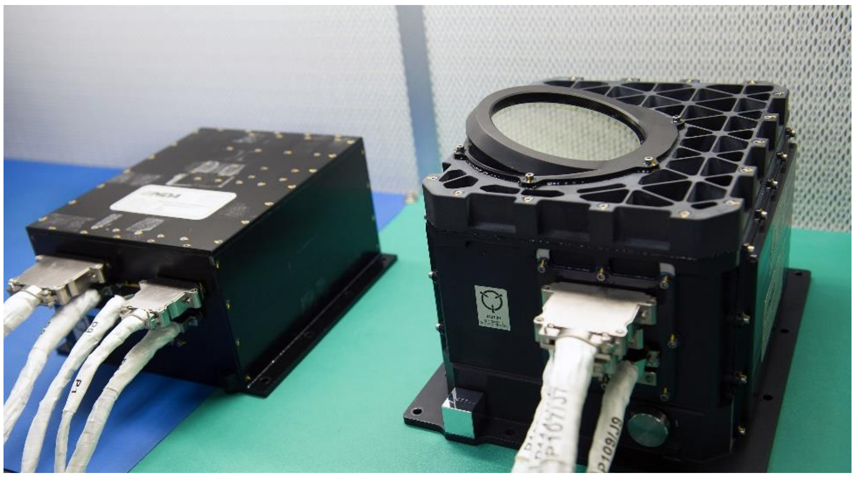


Figure 4 OSIRIS-Rex Laser Altimeter (OLA)

Different projects using LiDAR as a potential technology for planetary rovers have been developed, highlighting both the pros and cons. Some of the limitations with these sensors could be: weight and size, since LiDAR can be quite heavy and bulky, which could make difficult to fit such a device onto a lander or rover, where space is limited; cost, since LiDARs can be expensive and represent a significant expense for a planetary exploration mission; the need for significant computational resources necessary to process 360° complete resolution scans. These cons are some of the reasons why planetary exploration missions often use other types of sensors. However, LiDAR could be used in the future in planetary exploration missions to enhance the ability to detect obstacles and map the surrounding environment.

1.2 Surface reconstruction

Some of the previously mentioned sensors are used to acquire point clouds. A point cloud is a 3D representation of an object or a surface, composed of a set of three-dimensional points where each point represents a measurement taken by a sensor. A common step in the information extraction procedures is the surface reconstruction. Surface reconstruction is the process by which a 3D object is inferred, or 'reconstructed', from a collection of discrete points that sample the shape [4]. The surfaces considered in surface reconstruction are 2-manifolds that might have boundaries and are embedded in the Euclidean space $\mathbb{R}^3$ [5]. In the surface reconstruction problem, the initial assumptions are a finite sample $P \subset \mathbb{R}^3$ of an unknown surface and the task is to compute a model of S from P. This problem is referred as the reconstruction of S from P and the obtained result should match the original one both geometrically and topologically. It is important to notice that the process of reconstruction is usually made up by two stages: first, a piece-wise linear surface is reconstructed, and second, a piecewise-smooth surface is built upon the mesh [5]. However, finding out the appropriate surface, that correctly matches the geometric and topological properties, is not an easy task and it depends on the characteristic of both the surface and the point cloud. Sometime additional information of the surface can be available, such as oriented/unoriented normals or presence of breaklines (i.e. feature line or polyline representing a ridge or some other feature, that the user wishes to preserve in a mesh made up of polygonal elements [6]). In general, an increasing sampling density may ensure a better recovering of the surface, especially if the sample is dense in detailed area and sparse in featureless parts. The correct reconstruction algorithm depends on the final application and, for each application, reconstruction methods vary based on several factors. For example, it is possible to distinguish between techniques that interpolate a point cloud without any additional information and methods that assume some priors to fix the imperfections in point cloud. Most reconstruction procedures are specially designed for static objects and scenes but advance in scanning techniques has enabled the acquisition of point clouds that vary dynamically and consequently the development of algorithms for dynamic reconstruction. Or alternatively, there exist specific algorithms aimed at surface reconstruction for urban environments while other methods are designed for specific-application recognition: the first class of reconstruction techniques does not require the reconstruction of fine details such as individual bricks on a building while dense coverage scans could be required, for example, in the field of archeology where it is needed for high-detail reconstruction.

1.2.1 General overview

In general, surface reconstruction algorithms may be divided into two classes: systems based on object measurements and systems that do not use measurements [6]. Systems based on objects measurements require the acquisition of measurements of the objects in the environment, which are used to reconstruct the surface. This class comprises both methods based on triangulation and approaches that estimate surface normals instead of 3D data. These kinds of method rely on data coming from either passive sensor or active sensors. On the other hand, systems that do not use measurements rely on other types of data, such as images or videos, to reconstruct the surface. The generation of 3D models start from simple elements like polygonal boxes [6]. Overall, the main difference between these two approaches is the type of input data used, with systems based on object measurements requiring more specialized sensors and equipment, but potentially providing more accurate results, while systems that do not use measurements can be more versatile and use more widely available sensors but may be less accurate in certain situations. However, even if there exist thousands of surface reconstruction techniques, most of them are generally based on four steps:

1. Pre-processing, which consists of the editing operations on measured points. The pre-processing operations usually are:
   - data sampling, which could be uniform or based on the curvature of the points, but the uniform sampling is preferred since it allows to reduce certain types of errors.
   - noise reduction and removal of outliers. Points that are randomly distributed near the surface are traditionally considered to be noise while points far from the true surface are classified as outliers. While several algorithms infer the surface by passing near the point but not overfitting the noise, outliers should never be used for reconstruction purpose.
   - holes filling since physical constraints of the scanning devices cause missing data. These gaps are filled adding new points and using the density of surrounding points.
2. Study of the global topology, that is the determination of the neighborhood relations between adjacent parts of the surface.
3. Generation of the polygonal surface, which consists of the conversion from the given point cloud to a polygonal mesh, which is a collection of triangular or quadrilateral contiguous, non-overlapping faces, joined together along their edges [6]. The dataset is divided into small elements, typically triangles in 2D or tetrahedra in 3D, typically by means of finite element methods. The result of this step is the generation of vertices, edges and faces.
4. Post-processing, that include a whole set of operations to refine and smooth the polygonal surface. Some of the most common post-processing activities are edges correction or triangles insertion, aimed at filling holes.

1.2.2 Traversability concept

The processing of data coming from several sensors, especially visual ones, is a fundamental step to extract information about the surrounding environment, which can be further used to navigate the mobile robot toward the safest and most traversable area. The perceived data can be processed to create a map of the environment, which could be employed in a variety of applications including navigation, localization, and exploration. Maps, which can be either 2D or 3D, can include information about the localization of objects, the geometry of the environment, and other relevant features, such as obstacles and paths. In general, in mobile robot navigation, occupancy-based approaches are some of the most used methods. The occupancy-based algorithms use two or three-dimensional maps of the surrounding environment to determine whether a particular area of space is occupied by obstacles or other entities. This allows mobile robots to avoid obstacles and navigate safely and efficiently in complex environments. One of the most common methods used in occupancy-based approaches are occupancy-grid, which employs a 2D or 3D division of space into cells, where each cell stores a probabilistic estimate of its state (occupied or not).

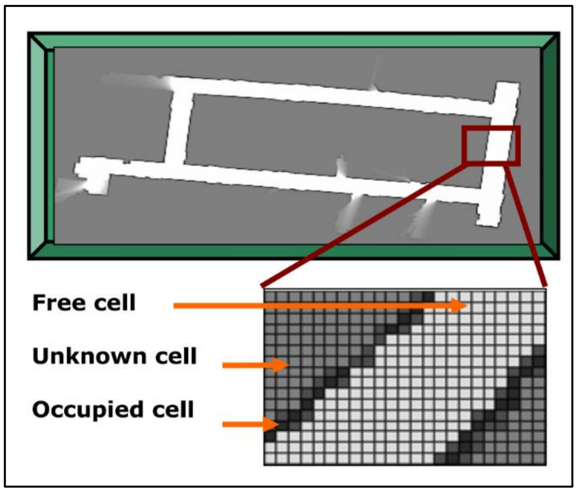


Figure 5 Example of occupancy grid

Other approaches refer to the level of environment representation, that can be sorted in geometrical, topological, or topo-geometrical levels [7]. Geometrical maps are representations of the environment that are based on geometric properties, such as distances, angles, and shapes. These maps are typically represented in a coordinate system, such as Cartesian or polar coordinates and the result is usually very accurate. It is necessary a big amount of data to model the environment through this approach, therefore this method is of limited use, especially in poorly structured outdoor areas. An example of geometrical map is shown in the figure below.

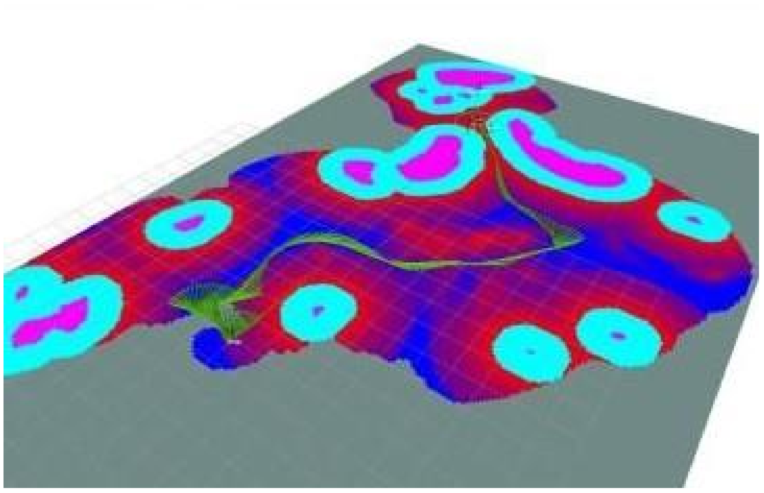


Figure 6 Example of geometrical maps

In a topological representation, the environment is represented as a graph, through a set of distinctive places, and the robot associates a particular sensory information to each of them, that makes them recognizable. Topological maps need the presence of repetitive elements, so they are mainly used in modelling indoor environments, but they are not suitable for a-priori unknown environments, especially outdoor ones. Moreover, hybrid representations have been developed, like a topological graph based on occupancy grid.

One of the main tasks of visual data processing is to determine which area are traversable, so which localizations are suitable to be navigated. Depending on the characteristics of the crossed environment, the concept of traversability has a proper meaning. Concerning outdoor navigation, it is necessary to estimate some parameters that describe the propension of the terrain to be crossed but several works mainly make use of two parameters for the definition of traversable area, which are terrain slope and roughness degree, that represent the amount of deviation from the smoothed surface on a smaller scale. Across the literature, several computation methods for these parameters may be found. For example, in [7], terrain slope is defined as the existing angle between the surface normal vector ($\vec{N}$) and the vector which is perpendicular to the horizontal surface ($\overrightarrow{W_\pi}$), as shown in the picture below:

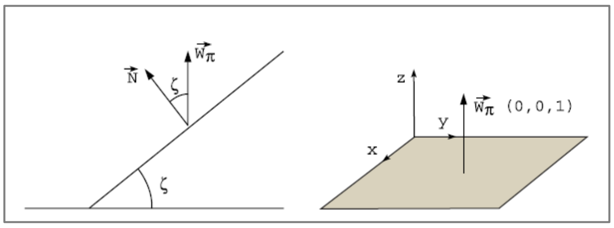
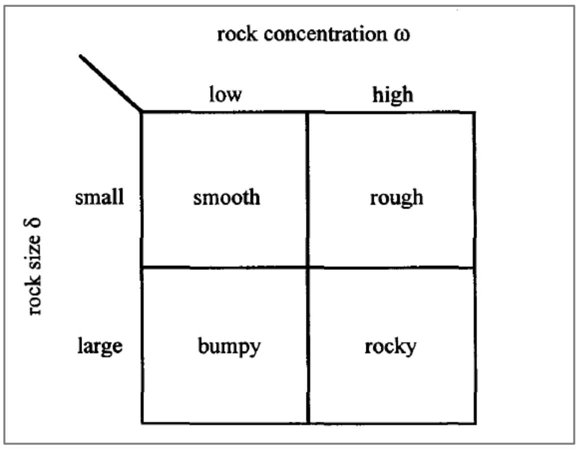


Figure 7 Slope definition and visualization

On the other side, the roughness degree is defined as the measurement of the surface deviation and the computation of the roughness is based on the normal vector deviation in each point, with the calculus of a statistic quantity named spherical variance, which expresses the variation of the normal vector in a local region. At this point, traversable areas are defined considering both these parameters. It is important to underline that several parameters concur to the definition of traversability for a terrain, like the characteristics of the mobile robot itself or the task the robot is going to perform.

1.2.3 Examples of traversability in planetary exploration

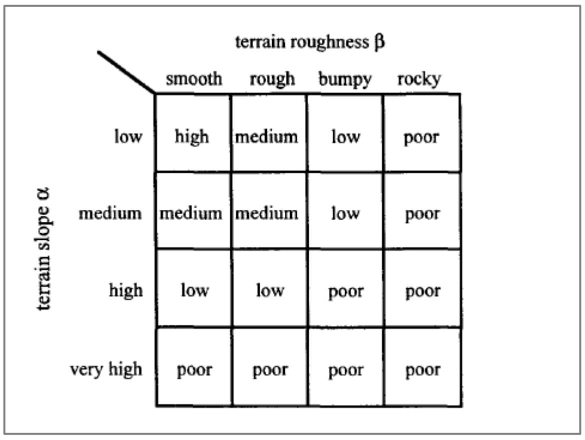
For what regards planetary environments, the first concept of traversability for mobile robots operating in planetary environments is introduced in [8]. This index is expressed by linguistic fuzzy sets, quantifying how traversable a particular terrain is for a given rover [8]. In particular, the index is computed based on the two physical variables mentioned before, the terrain slope and the terrain roughness. Both these quantities can be computed starting from data provided by the on-board stereo vision, as indicated in [9]. For the definition of the traversability index, the slope is represented through four linguistic fuzzy sets {LOW, MEDIUM, HIGH, VERY HIGH} and, using this approach, a precise measurement is not needed but it is necessary only to define to which set the slope belongs. The terrain roughness can be computed by fuzzy inference from the measurements of rock size and rock concentration on the terrain and it could be represented by four linguistic fuzzy sets {SMOOTH, ROUGH, BUMPY, ROCKY}, as shown in the table below:



rock size δ \ rock concentration ω	low	high
small	smooth	rough
large	bumpy	rocky

Figure 8 Fuzzy rules for terrain roughness

At this point, the traversability index is defined as a set of fuzzy relations, depending on the slope and the roughness of the terrain. The traversability index is represented through the four linguistic fuzzy set T = {POOR, LOW, MEDIUM, HIGH} and it is determined according to the rule shown in the following table:



terrain slope α \ terrain roughness β	smooth	rough	bumpy	rocky
low	high	medium	low	poor
medium	medium	medium	low	poor
high	low	low	poor	poor
very high	poor	poor	poor	poor

Figure 9 Fuzzy rules for traversability index

It is fundamental to outline that the traversability index does not depend only on the characteristics of the terrain but also on the properties of the rover, like size or climbing capability. Finally, just to point out the importance of parameters like the traversability index, it is crucial to say that the mentioned criteria is used to develop an autonomous navigation strategy that allows the robot to autonomously move in an a priori unknown environment.

Chapter 2

Overview about FPGAs

In chapter 1, it has been highlighted how much image processing is becoming an important part for both terrestrial and planetary rovers. Real time vision systems require the availability of processors which can work at speeds in the gigahertz range, while consuming a certain amount of power. The problems arise when space system require to be low-power and such processors are unavailable. Given those restrictions, the use of FPGAs with embedded processors has become an increasingly attractive technique for embedded processing [10]. Integrating a sequential processor to do sequential tasks, and FPGA fabric to do vector and/or parallel processing enables the low power and high computation ability required for robotic applications.

FPGAs, which stands for Field Programmable Gate Arrays, are integrated circuits (ICs), consisting of an array of programmable logic blocks which can be configured to implement simple logic functions (e.g., AND, OR) or to perform complex combinational functions. In both cases, hardware description languages (HDLs), such as VHDL or Verilog, can be exploited to design the FPGA configuration in the field. Modern FPGAs have become fast and powerful enough to enable the implementation of various algorithms in hardware, resulting in faster performance compared to software-only implementations on general-purpose microprocessors, as demonstrated in the articles [11-15]. Many researchers have focused on using FPGA accelerators to speed up the computationally intensive parts of programs, using different approaches to achieve acceleration, but all relying on some form of parallelism. Several applications are demanding increasing amounts of processing capability to achieve higher computational speeds but also lower power consumptions, acceptable manufacturing and packaging costs, rigorous time-to-market requirements. The main fields in which FPGAs are widely applied are Digital Signal Processing (DSP) and Digital Image Processing (DIP) [17]. This technology brings several advantages. First, FPGAs are faster than software algorithms on microprocessors because the hardware is tailored to a specific algorithm, so a speed increase of 10-100 times that of the equivalent software algorithm can be achieved, as demonstrated by authors in [18]. Additionally, FPGA implementation of software algorithms results in reduced power consumption, as FPGA clock frequencies are substantially lower (almost one tenth the speed) than those of microprocessors. Moreover, the usage of FPGAs implies a reduction of the payload per computation since most control is configured into the logic itself so overhead instructions (such as array indexing and loop computations) need not be emulated. However there exist other classes of ICs, like Application-Specific Integrated Circuits (ASICs), whose logic is fixed at fabrication time. Compared to ASICs, FPGAs are less dense and fast but they also bring several benefits to users, including faster time to market, re – programmability for users, no non-recurring engineering costs for fabrication or pre-tested silicon for use by the designer.

2.1 Main aspects of FPGAs

Technology markets have been driven by developments in silicon technology according to the progress described by Moore's law, which predicted a doubling of the number of transistors every 18 months. Moreover, the reduction in the costs of transistors has been crucial. Early electronic systems were created on printed circuit boards (PCBs), by aggregating standard components such as microprocessors and memory chips with digital logic components [20]. The increasing number of transistors and input/output pins as well as the complexity of the implemented systems caused the integration step on PCBs to become harder and harder. Moreover, the need to develop systems that could easily adapt to evolving design requirements raised, pushed also by the desire to have the same flexibility allowed by microprocessors. In this context, the idea of Field-Programmable Device (FDP) emerged. The term FPD refers to a class of integrated circuits aimed at the design of digital hardware, where the chip is directly configured by the end user.

2.2 Evolution of FDPs

One of the first user-programmable chip was the Programmable Read-Only Memory (PROM). A PROM consists of a fixed AND plane, connected to a programmable OR plane, as shown in Figure 10. The fixed AND plane acts as a decoder so it is responsible for reducing the number of input pins that go into the memory, by decoding the address input pins, while the data is stored in a storage area or memory array. The decoder generates various address lines using AND gates, and the outputs are combined using OR gates. In this way, given n inputs and m output lines, the structure could store n^2 m-bit words. Since the programming step is realized through the burning of the fuses in the OR plane, it is easy to understand that this kind of technology can be programmed only once. This simple structure successfully allowed the implementation of logic functions, which were simply expressed as sum of products since designers usually made use of logic minimization techniques, such as those based on Karnaugh maps or Quine-McCluskey minimization [20]. However, these kind of logic functions rarely need more than a few product terms while a PROM contains a full decoder for its address inputs, so this technology appears to be quite inefficient and nowadays it is not so adopted.

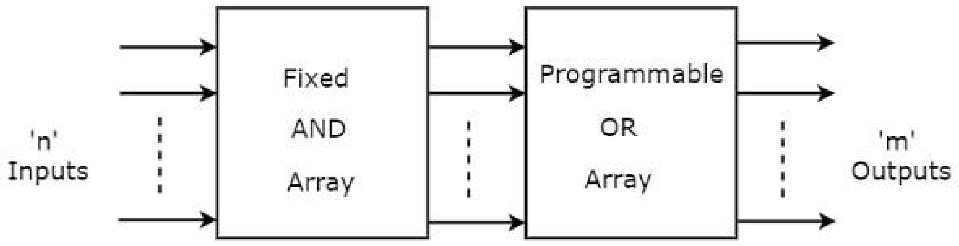


Figure 10 Structure of a PROM

Then Programmable Logic Array (PLA) was introduced, which is made by a programmable wired AND plane and a programmable wired OR plane and whose general structure is depicted in Figure 11:

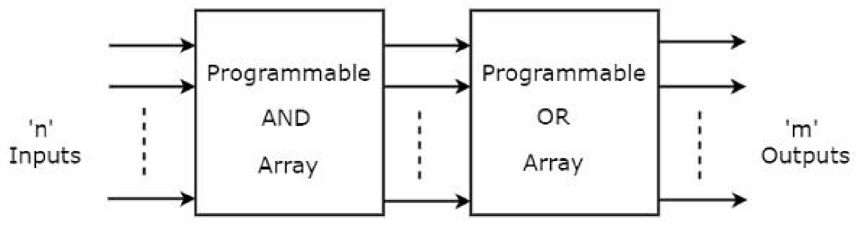


Figure 11 Structure of a PLA

In this kind of structure, it is possible to implement logic functions expressed as sum of products. Indeed, any of the inputs or their complements can be ANDed together in the AND array, so it is possible to generate only the required products by using these AND gates. At the same way, each output in the OR plane can be configured to produce the logical sum of any of the AND-plane outputs. An example of PLA is shown in Figure 12, implementing a sum – of – products function:

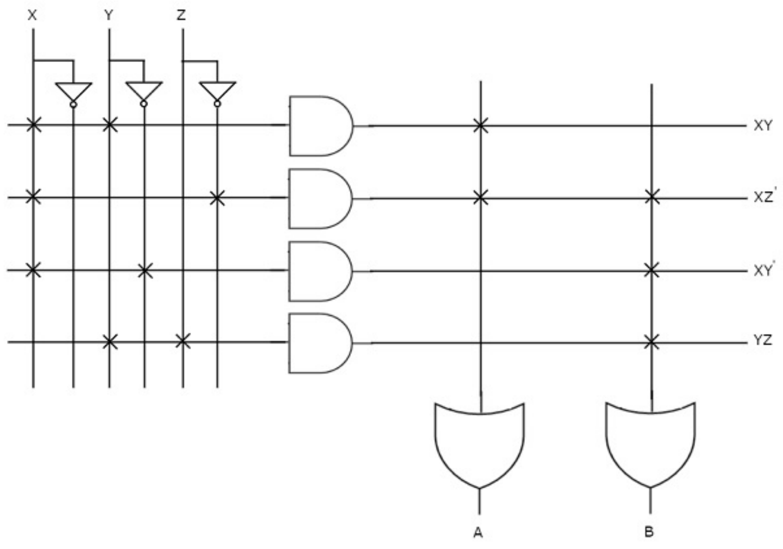


Figure 12 Example of a PLA

However, the presence of two programmable logic arrays caused not only difficulties in the manufacturing but also not so much satisfying performances in terms of speed. This is the reason why PLA were substituted by Programmable Array Logic (PAL), which are simply realized by the connection between a programmable AND plane and a fixed OR matrix. The general structure of a PAL is shown in figure 2.4:

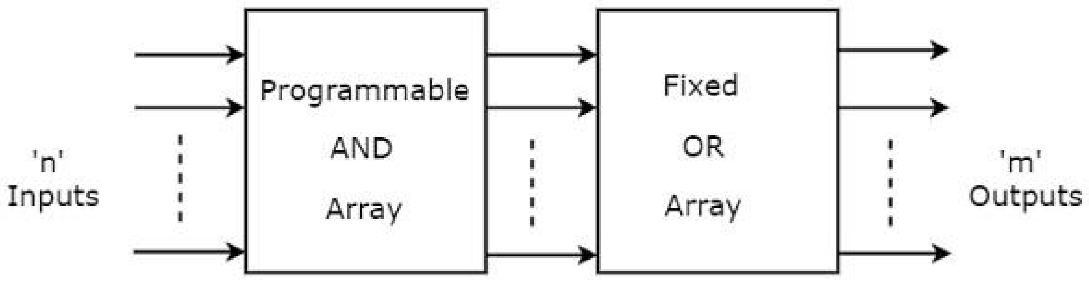


Figure 13 Structure of a PAL

As in PLAs, the inputs of AND gates are programmable so, based on the requirements, it is possible to program any of those inputs. On the other side, the inputs of OR gates are not programmable, therefore the number of inputs to each OR gate will be fixed. As the previous one, this structure allows to implement functions in the form sum – of – products. Figure 14 shows an example of sum – of – product function, implemented on a PAL device:

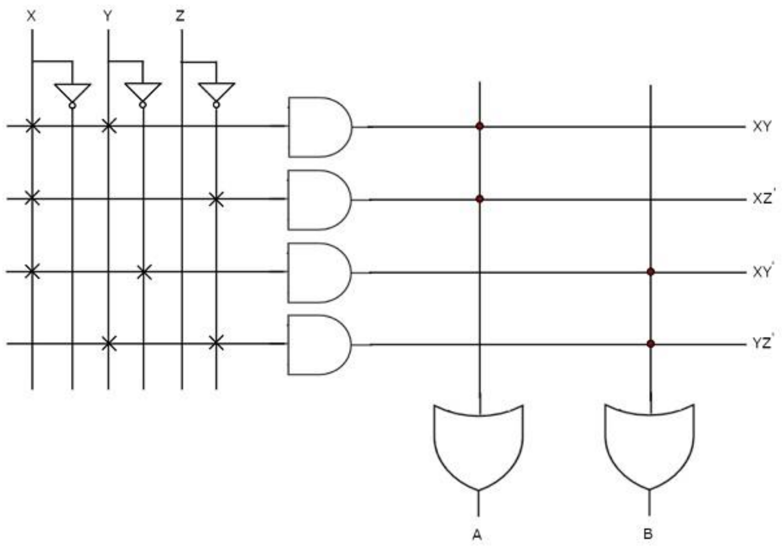


Figure 14 Example of a PAL

Since the PAL is characterized by less flexibility with respect to the PLA, it is implemented with several variants, like the number of inputs and outputs. The PAL16L8 is a demonstration of what has already been stated: indeed, it is a particular PAL implementation, provided with 16 inputs and 8 outputs:

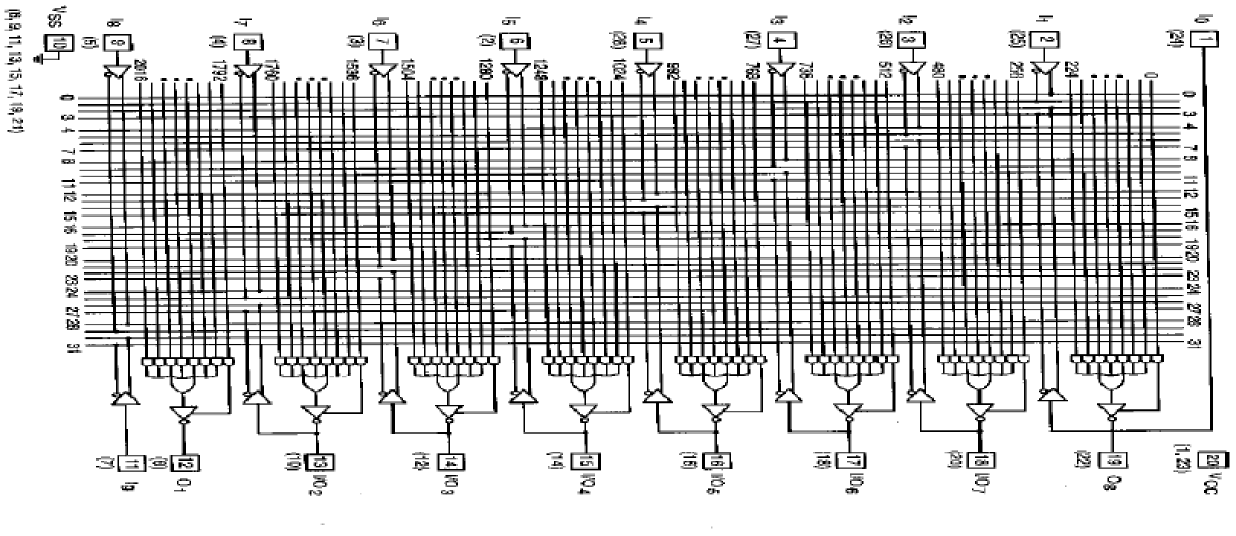


Figure 15 PAL16L8

As it is possible to see from the previous image, PAL are often supplied with flip-flops to implement not only combinational but also sequential logic. In general, the last two types of devices, which have been presented till now, fall under the categorization of Simple Programmable Logic Devices (PLDs). Although their pros, such as low cost and versatility, the cons are represented by the fact that the structure of the programmable logic planes grow too quickly in size as the number of inputs is increased [21]. These problems have been solved by the introduction of Complex Programmable Logic Devices (CPLs), which consists of the arrangement of multiple SPLD-like blocks on a single chip. The main idea behind CPLD is to implement an architecture containing several logic blocks, each like a small PLD, instead of building larger PLDs with more inputs or product terms. Then the logic blocks can communicate with each other using signals routed through a programmable network of interconnects. Figure 16 shows an example of the generic architecture of a CPLD, consisting of four PLD sub-blocks:

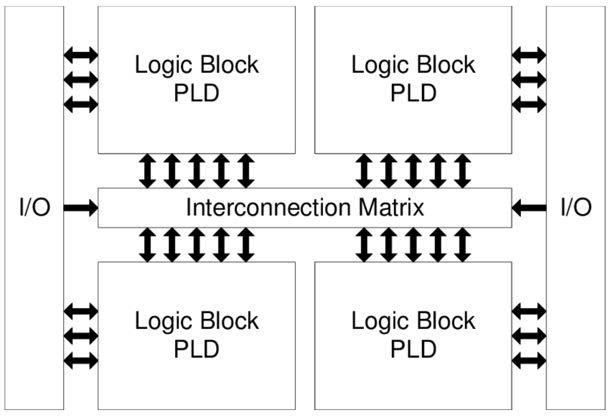


Figure 16 Example of architecture for a CPLD

However, it results quite complex to extend these architectures to high density designs. At this point, FPGAs were introduced as a trade-off between the high – density capabilities of ASICs and the flexibility provided by PLDs. Early architectures for FPGA were composed by:

- programmable logic units, that could be programmed to implement logic functions;
- programmable interconnections;
- programmable I/O pins.

A schematic architecture of early FPGAs is shown in Figure 17:

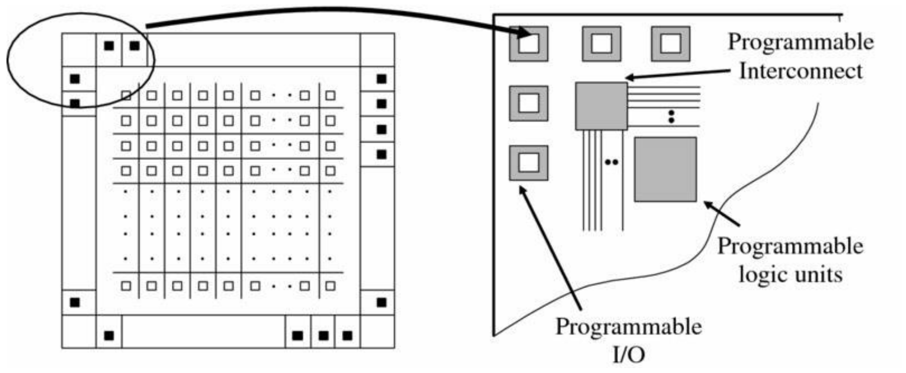


Figure 17 Architecture of early FPGAs

Today, FPGAs are still made by this array of simple circuit elements, called logic element, and interconnectable resources, which can be programmed by the end user. Each logic block consists of digital logic components, such as multiplexers, flip-flops, look-up tables (arrays of data to map input values to output values, approximating mathematical functions) or adders. The strength of FPGA is represented by the fact that the logic blocks within it are built with interconnections that can be reconfigured by the user using a hardware description language (HDL). The actual structure is quite similar to the architecture of the first ones but the number of cells in the same device has strongly grown. Indeed, according to the technology process described by Moore’s law, the increasing in the number of transistors is related to the shrinking in the physical dimensions of transistors, allowing for a higher number of logic blocks integrated on the same unit. Initially, the devices were made just by a single cell. The increasing in the number of cells has led to the need of connecting not only each single cell to the outside border of the physical component but also between themselves, so a programmable structure has been introduced to route the signal inside the component. This evolution is shown in the following picture, provided by Altera:

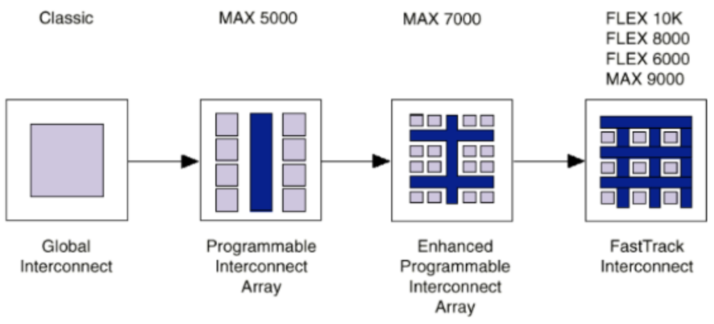


Figure 18 Altera architecture evolution

Moreover, the input/output capability was strongly affected by the increasing in the number of transistors so modern FPGAs can drive numerous signals, up to one thousand. So, nowadays, FPGAs, as illustrated in Figure 19, consist of an array of programmable logic blocks of potentially different types, including general logic, memory and multiplier blocks, surrounded by a programmable routing fabric [22].

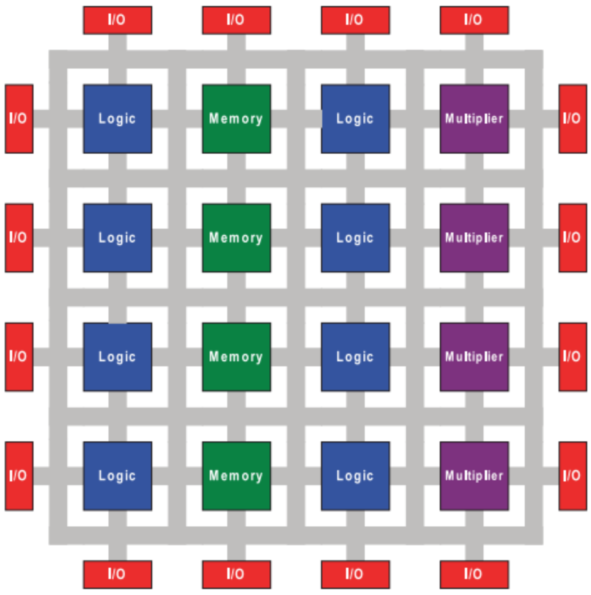


Figure 19 Basic FPGA structure

2.3 Structure of modern FPGAs

The main components of a FPGA device are:

- <u>Configurable Logic Blocks (CLBs)</u>

  A CLB is the fundamental piece of an FPGA. In its most basic form, a FPGA is a chip made by thousands of configurable logic blocks. The function of a single CLB and their interconnection can be programmed by the designer to perform any logic function. An individual CLB consists of a number of discrete logic components itself, such as look-up tables (LUTs) and flip-flops. Figure 20 shows an example of a logic cell, which is the building block of a CLB:

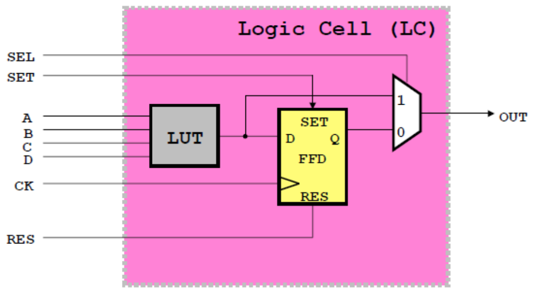


Figure 20 Example of logic cell

A look-up table is a type of configurable logic block that can implement any Boolean function of its input signals. A LUT consists of a small block of memory, typically 4 to 6 bits wide, and a decoder circuit that selects one of the memory cells based on the input signals. The memory cells in the LUT are pre-programmed, with truth table values for the Boolean function that the LUT is intended to implement. For example, a 4-input LUT might have 16 memory cells, with each cell representing one of the 16 possible input combinations and containing the corresponding output value for the Boolean function. When the input signals to the LUT change, the decoder selects the appropriate memory cell based on the input values, and the output of the LUT is set to the corresponding value stored in the memory cell. Moreover, the LUTs can also serve as a distributed RAM or a shift – register. The fact that registers are available in CLB means that it is possible to create sequential logic circuits by connecting the output of LUTs to flip-flops. Some logic cells are provided with carry logic blocks, useful to implement fast arithmetic circuits.

Typically, logic cells are grouped in slices and the proper interconnection between slices leads to the formation of configurable logic blocks. There exist both local and global interconnections, to offer wider opportunities of programmability.

- Digital Signal Processing (DSP) Slice

  A DSP is a component designed to carry out digital signal processing functions, such as filtering or multiplying, much more efficiently than if the same functions were implemented using many CLBs. Each variable-precision DSP block offers a range of multiplicative and additive support functionalities.

- Transceivers

  Transceivers transmit and receive serial data to and from the FPGA at extremely high rates. The task of converting information on the FPGA into serial data, as well as receiving serial data externally and converting it into useful information, while checking for errors in the data becomes more difficult to do with the configurable logic of the FPGA as speeds increase. This dedicated component allows for high-speed data transfer without consuming the logic resources of the FPGA.

- Block Random Access Memory (BRAM)

  FPGAs are usually provided with on – chip Block Random Access Memories (BRAMs) as well as on – board SRAM or DRAM. Memory blocks can be particularly useful, for example if there is the need to store some previously data coefficients or if it is necessary to delay some values. BRAM is directly built into the FPGA fabric in such a way to offer more efficient storage for large amounts of data. BRAM is typically used to implement memory-intensive algorithms, such as image and video processing, where fast access to large amounts of data is critical. On the other side, both DRAM and SRAM are external to the FPGA device, but they offer larger amounts of storage capacity than BRAM. DRAM is commonly used in FPGA designs that require large amounts of data storage, such as high-performance computing and networking applications, while SRAM is often designs that require high-speed access to small amounts of data, such as cache memory or register files.

- Input/Output (IO) Blocks

  Input/output blocks are the components through which data are transferred into the FPGA or out of it. Input/Output pins are grouped in the so-called IO banks, which are configurable depending on the type of data to receive or transmit. They are like transceivers but operate at lower speeds and can maintain more functional flexibility.

2.4 Programming technologies for FPGAs

An important criterion to distinguish the different types of FPGAs is based on the programming method through which connections are made. There are mainly three different types of FPGAs:

- <u>SRAM FPGAs</u>

  SRAM-FPGAs make use of SRAM-based memory to store the logic and wiring information. The basic element of SRAM-FPGAs is a static random – access memory cell, which is based on CMOS technology. Each static memory cell is usually made by six transistors, four of which form two cross – coupled inverters, as shown in Figure 21:

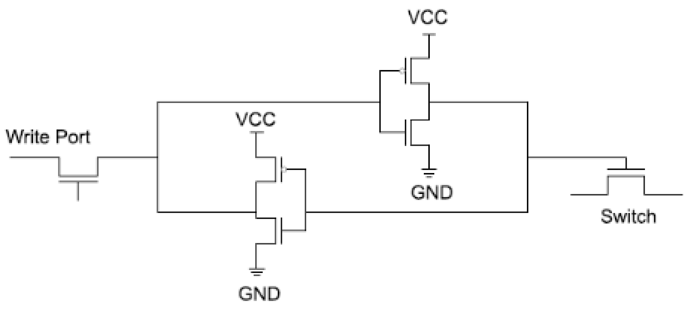


Figure 21 SRAM-based memory element

SRAM can keep the stored information as long as power remains on and it does not need to be periodically refreshed but it is a volatile component, so the information is lost when the power is turned off. SRAM cells can be used both to store data in look-up tables and to select lines of the multiplexers, necessary to route interconnection signals. Because of the volatile nature of SRAMs, the configuration data shall be stored in an external nonvolatile memory, to be uploaded at power up. SRAM-FPGAs are the most common, not only due to the fact that they are reconfigurable but also because the adopted CMOS technology allows for high speeds and low power consumptions.

- Antifuse – based FPGAs

  In antifuse – based FPGAs, desired logic circuit can be realized by burning antifuses off. An antifuse is an electrical device, which behaves oppositely with respect to a fuse: while a fuse breaks a connection in a circuit when it is crossed by a high current, an antifuse has a high resistance at the beginning so it is designed to create an electrically conductive circuit path permanently, typically at high voltage. An antifuse can be implemented by placing a thin barrier of an insulating material between two metal conductors. When a high enough voltage is applied across the insulating material, it breaks down the antifuse and a low resistance path is established for the current to flow.

  With respect to SRAM-FPGAs, the occupied space can be reduced since there is no need of silicon area to realize the connections, even if the high currents that cross the circuit need to be supplied by larger transistors. Moreover, this kind of FPGAs is nonvolatile so there is no need of additional memory where the configuration program has to be stored and the device can instantly work when powered up. However, since antifuse – based FGPAs are based on burning off antifuses, they are one – time programmable (OTP) devices so they cannot be reconfigurable after the first programming. In particular, since anti-fuse-based FPGAs require a nonstandard CMOS process, they are typically well behind in the manufacturing processes that they can adopt compared to SRAM-based FPGAs [22].

- Flash FPGAs

  Flash – FPGAs are based on flash or EEPROM memory cells. In this type of FPGAs, the fundamental block is a floating – gate, an electrode formed within the gate insulator of a field-effect transistor, so it is placed between the normal gate electrode (the control gate) and the channel, as shown in Figure 27:

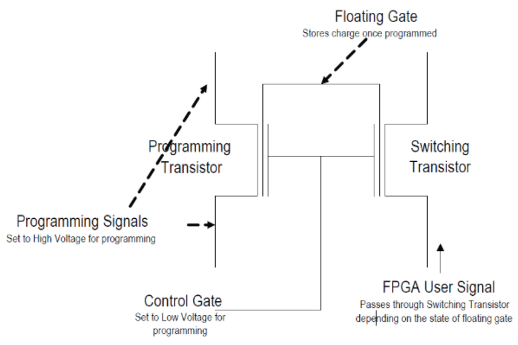

Floating Gate
Stores charge once programmed
Programming Transistor
Switching Transistor
Programming Signals
Set to High Voltage for programming
FPGA User Signal
Passes through Switching Transistor depending on the state of floating gate
Control Gate
Set to Low Voltage for programming

Figure 22 Flash memory cell

The amount of charge stored on the floating gate determines whether the transistor will conduct or not. The switch element consists of two floating gate NMOS-transistors. A switch transistor turns on or off the data path and a programming transistor programs the floating gate voltage. The floating gate is completely surrounded by insulators so it can keep the charge independent of whether the circuit power supply voltage is present. Flash – FPGAs result to be reconfigurable and nonvolatile, so they represent a tradeoff between the two typologies of FPGAs previously discussed. The usage of this kind of FPGAs was not so common in the past because of the need of wide area but now suitable results in the shrinkage of the area have been achieved. One disadvantage of flash-based devices is that they are not infinitely reconfigurable because charge buildup in the oxide could prevent a proper erasing.

2.5 FPGAs in space applications

Recently, FPGAs are becoming widespread in space applications while the usage of microcontrollers is decreasing because of the effects of high ionizing radiations. Indeed, these radiations are absorbed by microcontrollers with a consequent decreasing in performances, while the physical structure of FPGAs allows to reject radiations. For this reason, FPGAs are said to be immune to SEEs (Single Event Effects). The SEEs are caused by a single ionizing particle, which creates an included charge. This latter one can immediately and temporarily affect the correct operation of the device or destroy it [23]. In the following sub-paragraphs, some of the most common examples of usage of FPGAs in space applications are reported.

2.5.1 Sojourner Rover

As anticipated in chapter 1, the Sojourner Rover was the first Mars Rover to land in the Ares Vallis, the outflow channel on Mars, as part of Mars Pathfinder mission.

Figure 23 A representative picture of Sojourner Rover

This rover was equipped by the Athena Software Development Model (SDM), which used a highly distributed approach in adopting a 12 MHz/10 MIPS R3000 CPU (drawing 2–3W of power) supported by FPGAs for low-level motor control [24]. The system comprises six remote engineering units and each unit is provided with an FPGA based motor controller that can control two brushed motors. The motor controller is a PID controller running at frequency of $1\ kHz$, reading quadrature encoders and outputting the direction and pulse width modulated duty cycle to drive the motor.

2.5.2 ESA Lunar Rover Mockup

The Lunar Robotic Mockup was developed to provide a Lunar-like rover platform for mounting robotics payloads [25]. This rover is shown in Figure 24.

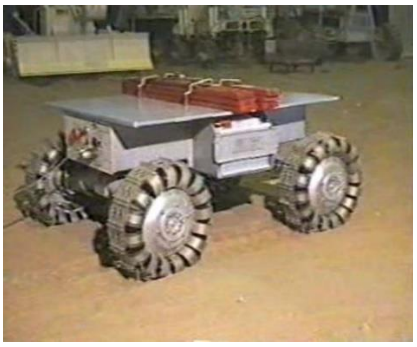


Figure 24 A representative picture of ESA Lunar Mockup

The control software of this rover runs on a microcontroller that receives velocity and motor angle data through a serial line. A PID controller generates commands that are sent to the H-bridges for steering and to an FPGA where the PID controller is implemented [2].

2.5.3 Jet Propulsion Laboratory applications

JPL has developed and applied stereo vision systems to rover in order to perform motion control. Different JPL stereo vision systems were accelerated by using FPGA. Thanks to this electronic device, it is possible to obtain a 16 times speedup in computer vision task with respect to a linear processor.

FPGAs are widely adopted since the space for a processor is more and more constrained in terms of occupied space. It implies that processor speeds in the gigahertz range are unavailable, instead only processors less performant are allowed resulting in a failure in real-time stereo processing. For this reason, FPGAs with embedded processors are widespread in the space field.

JPL is also developing a rover avionics module based on the Xilinx Virtex-II Pro FPGA which includes two embedded PowerPC 405 processor cores with a processing speed of 300MHz with the rest of the FPGA processing at 100 MHz.

The adoption of FPGAs is exemplified in the design of the Kapvik micro-rover by implementing FPGA electronics to process images faster for autonomous navigation [2].

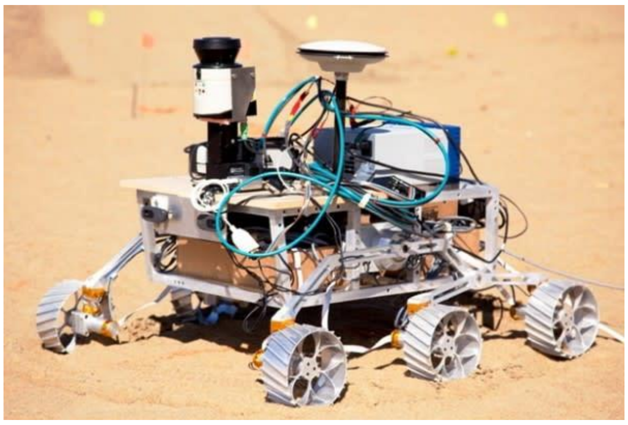


Figure 25 Kapvik micro-rover

This micro-rover is produced by Canadian Space Agency as a demonstration that micro-rovers could have similar functionalities of the larger ones. This last feature implies a cost reduction during the launch phase since the occupied volume and the weight decrease.

2.5.4 Perseverance Rover

Perseverance is a Rover developed by NASA and launched in 2020, aimed at exploring the Jezero crater. It makes use of an FPGA technology (Xilinx Virtex-5) as one of the main processing units. This unit is first responsible for rover entry, descent and landing on Mars and then it is programmed for computer vision tasks by NASA engineers from the Earth. Other units on Perseverance such as UHF transceivers, radar, X-ray (identifying chemicals), and cameras are controlled with XQR4VFX60 and XQR2V3000 FPGAs.

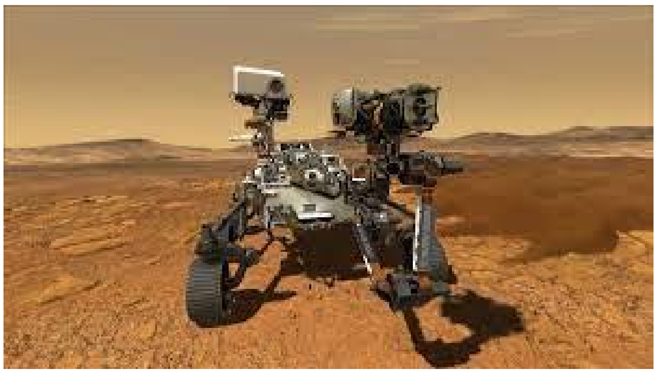


Figure 26 A representative picture of Perseverance

2.5.5 Small Satellite Communication system

FPGAs are also used in software defined radio (SDR) transponder design for the emerging SmallSat and CubeSat industry. Small spacecraft (SmallSats) consists of spacecraft lightweight since the concept is to obtain a system with a mass less than 180 kg and the size is comparable with a large kitchen fridge [26]. Instead, CubeSat belongs to the class of nanosatellites that means a mass spanning from 1 Kg to 10 Kg [26].

Figure 27 An example of CubeSat

The FPGA substitutes, also in this case, the role of the processor since in the SDR the software manages all the function of filtering, carrier recovery, error correction or framing [27]. Article [27] demonstrates how the usage of FPGA in this application highly simplifies the computational payload.

Chapter 3

Normal estimation algorithm

The aim of the thesis work here presented is to develop an algorithm which is able to reconstruct the surface represented by a point cloud. In the future, the algorithm should be part of the vision system mounted on a little rover, available in Thales Alenia Space facilities. As explained in the first chapter, there exist plenty of methods to reconstruct the surface, starting from a dataset representing it. The method followed in this work is the one based on [28].

3.1 Introduction of the problem

A point cloud P is a collection of points in 3D, where the coordinates of each point $p_i = (x_i,\ y_i,\ z_i)$ are expressed with respect to a fixed coordinate system, usually having its origin at the sensing device used to acquire data [29]. Each point of the point cloud returns a measure of the distances between the investigated surface and the three axes, whose intersection point coincides with the assumed viewpoint. In this case, the initial assumption about data is that a time – of – flight camera, mounted on top of a rover, returns a cloud of points equally spaced points. A time – of – flight camera is an imaging system that works by measuring the time it takes for a light signal to travel from a camera to an object and back again. The camera emits a short burst of light, usually in the form of a pulsed laser, which reflects off the object and returns to the camera. By measuring the time it takes for the light to make this round-trip, the camera can calculate the distance to the object. This process is repeated many times per second, allowing the camera to create a 3D map of the scene. In particular, the starting hypothesis is that there is no noise corrupting data.

The aim of the proposed algorithm is to compute surface normals, 3D features which represent a fundamental source of information of the inspected surface. Indeed, surface normals indicate the direction that a surface is facing at each point and can be used to obtain crucial information about the environment, such as the curvature of the surface or the presence of edges, as demonstrated in [30]. For example, article [31] introduces a geometrical index, computed from the three components of the normal vector $N = (N_X,\ N_Y,\ N_Z)$, and the index is employed in distinguish horizontal surfaces from vertical ones.

To compute surface normals, it is important to outline the fact that the acquired point cloud represents a set of point samples on the real surface so there may two possibilities:

- Compute an approximation of the surface from the acquired dataset, using surface meshing techniques and then compute the surface normals from the mesh.
- Use approximations to infer the surface normals from the point cloud directly.

In particular, I decided to develop the second method since it allows for faster computations, which may be crucial in space applications where resources are limited. For example, in [30], the adoption of the statistical approach avoids the clustering phase, since the edges are extracted exploiting only the computation of the eigenvalues of the covariance matrix. The problem of determining the normal to a point on the surface is approximated by the problem of estimating the normal of a plane tangent to the surface, which in turn becomes a least-square plane fitting estimation problem [29]. In particular, the followed approach is the one exploited in [28] and [30], where the Principal Component Analysis (PCA) is exploited in order to compute the normal vector in the considered point. At this point, it is crucial to outline that the computation of the normal involves the definition of a neighborhood of points, so the amount of its surrounding points is an important factor affecting accuracy of the point normal [32]. It is fundamental to define the correct number of neighbors and surrounding points are referred to as k – neighbors, where k is the number of the nearest points around the target one. Article [32] demonstrates how the number k must be carefully chosen since a too small or too large value may lead to inappropriate computation of the normal.

3.2 PCA

Principal Component Analysis (PCA) is an unsupervised learning method, which is widely exploited in the machine learning field to extract important features about the collected data. It raised in the context of psychometrics in the 1930's and nowadays it is a well-established machine learning technique, adopted in a wide range of applications spanning from finance, medicine to engineering or image processing [28]. PCA allows for the computation of the directions of maximal variance for the input data, that are the orthogonal directions along which the variation of data is the highest.

3.2.1 Mathematical explanation

The starting hypothesis is a data matrix $A \in \mathbb{R}^{d,N}$, where d represents the space dimension (i.e., the number of observed features) while N represents the number of performed measurements, so $A = [a^{(1)} \dots a^{(N)}]$, where $a^{(i)} \in \mathbb{R}^d$. At this point, it is possible to define the centered data matrix as:

$$A_c = A - \hat{a}\mathbf{1}_N^T \qquad \hat{a} \doteq \frac{1}{N}\sum_i a^{(i)}$$

At the same way, centered data points are defined as:

$$\tilde{a}^{(i)} = a^{(i)} - \hat{a}, \qquad i = 1, \dots, N$$

Before moving on, it is important to introduce the concept of covariance matrix and the methods that can be used to estimate it.

3.2.1.1 Covariance

Given a collection of numbers $z_1, \dots, z_N$, the variance is a measure of the mean spread distance of the data with respect to the center, which is defined as the average of the z_i:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(z_i - \hat{z})^2$$

Input data may be vectors and not scalar, so let us consider the case in which the input data is a matrix $A \in \mathbb{R}^{d,N}$. In this case, the measure of variation previously provided by the variance is expressed by means of the covariance matrix, which is a measure of how much each of the dimensions varies from the mean with respect to each other:

$$S = \frac{1}{N}\sum_{i=1}^{N}\left(a^{(i)} - \hat{a}\right)(a^{(i)} - \hat{a})^T = \frac{1}{N}A_C A_C^T$$

The covariance matrix is a square matrix of dimension dxd. It is symmetric and positive – semidefinite.

A direction in the space can be identified by a unit vector $v \in \mathbb{R}^d$ so it is possible to define the score of an input datum $x \in \mathbb{R}^d$ along direction v as the projection of x onto v, which is computed by means of the scalar product $v^T x$. At the same way, the scores of the centered data points along direction v are expressed as:

$$s_i = v^T\left(a^{(i)} - \hat{a}\right) \quad i = 1, \dots, N$$

The values s_i provides an insight about the distribution of the data points along the considered direction v. In particular, the variance of the data along direction v is given by:

$$\begin{aligned}\sigma_v^2 &= \frac{1}{N}\sum_{i=1}^{N} s_i^2 = \frac{1}{N}\sum_{i=1}^{N} v^T(a^{(i)} - \hat{a})(a^{(i)} - \hat{a})^T v \\ &= v^T\left(\frac{1}{N}\sum_{i=1}^{N} v^T\left(a^{(i)} - \hat{a}\right)(a^{(i)} - \hat{a})^T\right)v = v^T S v\end{aligned}$$

Where S is the sample covariance matrix. Then, the principal direction is the directional of maximal variance, so it can be obtained by solving the optimization problem, with respect to the direction v, along which the variance assumes its higher value:

$$v_{max} = \max_{||v||_2=1} v^T S v$$

Once the direction of highest variance has been found, it is possible to find the others, by following the deflation method. It consists of projecting the data points on the subspace which is orthogonal to the previously computed direction and then finding the direction of maximal variance for projected data. If the dimension of input space is equal to d, the process can be repeated d times or it can be stopped at a certain $k < d$. Figure 28 is an example of a two – dimensions point cloud and its principal directions:

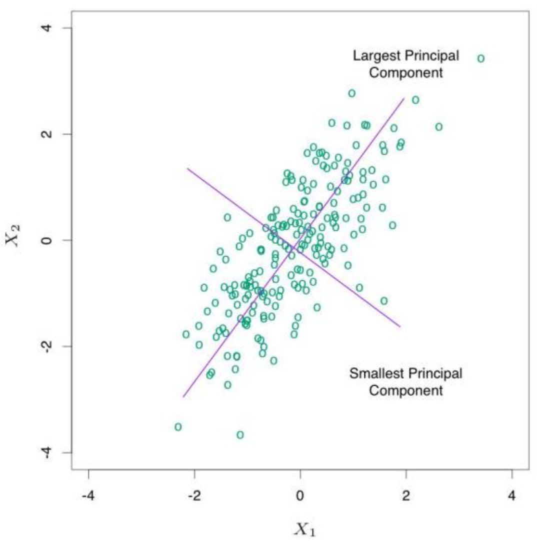


Figure 28 Principal components of a dataset

In this case, the first principal direction is the direction that maximizes the variance of the projected data while the second principal direction coincides with the smallest principal component, along which the variance is the smallest. There exist two main method that allows for the determination of the principal directions:

- Eigenvalue decomposition (EVD) of the covariance matrix;
- Singular value decomposition (SVD) of the (centered) data matrix.

Only the first one is investigated in this thesis work.

3.2.1.2 Eigenvalue decomposition of the covariance matrix

First, it is necessary to introduce the eigenvalue decomposition and the Rayleigh variational representation theorem. The algebra theorem for eigenvalue decomposition for symmetric matrices states that every symmetric matrix $S \in \mathbb{R}^{m,m}$ can be decomposed as:

$$S = U \Lambda U^T$$

In the equation above, $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_m)$ is a diagonal matrix, whose entries are the eigenvalues of S, $\lambda_1 \geq \cdots \geq \lambda_m$ while $U = [u_1, \dots, u_p]$ is a $m \times m$ orthogonal matrix, containing the eigenvectors u_i of S.

The Rayleigh variational theorem states that, given a symmetric matrix $S \in \mathbb{R}^{m,m}$, then its largest and smallest eigenvalue can be computed by solving the two optimization problems:

$$\lambda_{MAX} = \lambda_1(S) = \max_{w \,:\, ||w||_2 = 1} w^T S w$$

$$\lambda_{MIN} = \lambda_m(S) = \min_{w \,:\, ||w||_2 = 1} w^T S w$$

The two optima are attained, respectively, at $w_{MAX} = u_1$ and $w_{MIN} = u_m$.

Before, it has been asserted that the principal direction can be find by solving the following optimization problem involving the sample covariance matrix:

$$v_{max} = \max_{||v||_2 = 1} v^T S v$$

Since the covariance matrix is symmetric, its eigenvalue decomposition can be written as $S = \sum_{i=1}^{d} \lambda_i u_i u_i^T$, with $\lambda_1 \geq \cdots \geq \lambda_d$ and $U = [u_1, \dots, u_d]$ is orthogonal. According to the Rayleigh variational theorem, the above optimization problem is equal to:

$$\lambda_{MAX} = \max_{||v||_2 = 1} v^T S v$$

As a consequence, the solution for the optimization problem is u_1, which is the eigenvector of S corresponding to its largest eigenvalue. At the same way, the eigenvector u_2, which is the eigenvector of S associated to its second largest eigenvalue, returns the second direction of maximal variance and so on. Finally, the k directions of largest variance can be obtained by performing the eigenvalue decomposition of the covariance matrix.

3.2.2 PCA for the computation of normal vectors

To reconstruct the surface represented by a point cloud, the main idea is to associate a tangent plane with each of the data points $p_i = (x_i, y_i, z_i)$, which locally approximates the surface. This plane represents the best least squares fitting to the point p_i and its neighbors. The general equation for the tangent plane, lying in three dimensions, is:

$$P_T \; : \; ax + by + cz + d = 0$$

The tangent plane can be fully represented by two quantities, a center $p_C = (x_C, y_C, z_C)$ and the unitary normal vector $N = (N_X, N_Y, N_Z)$ which is orthogonal to the surface. The information provided by the normal vector is fundamental since its three components coincide with the coefficients $a,\ b,\ c$ defining the tangent plane. To compute the center and the normal vector, it is necessary to consider the k – neighborhood of p_i, which is the collection of the closest k points with respect to p_i and they are fundamental to capture the local geometry. At this point:

- The center p_C is computed as the centroid of the collection of points made by p_i and its k neighbors, according to the formula:

$$p_C = \frac{1}{k}\sum_{i=1}^{k} p_i$$

Since p_C has three components, the above formula leads to the following three relations:

$$x_C = \frac{1}{k}\sum_{i=1}^{k} x_i \quad y_C = \frac{1}{k}\sum_{i=1}^{k} y_i \quad z_C = \frac{1}{k}\sum_{i=1}^{k} z_i$$

- The normal vector N is computed as one of the eigenvectors of the positive – semidefinite symmetric covariance matrix COV:

$$COV = \begin{pmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(y,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{pmatrix}$$

Each element of the covariance matrix can be computed as:

$$COV(x,x) = \frac{1}{k}\sum_{i=1}^{k}(x_i - x_C)^2$$

$$COV(y,y) = \frac{1}{k}\sum_{i=1}^{k}(y_i - y_C)^2$$

$$COV(z,z) = \frac{1}{k}\sum_{i=1}^{k}(z_i - z_C)^2$$

$$COV(x,y) = COV(y,x) = \frac{1}{k}\sum_{i=1}^{k}(x_i - x_C)(y_i - y_C)$$

$$COV(x,z) = COV(z,x) = \frac{1}{k}\sum_{i=1}^{k}(x_i - x_C)(z_i - z_C)$$

$$COV(y,z) = COV(z,y) = \frac{1}{k}\sum_{i=1}^{k}(y_i - y_C)(z_i - z_C)$$

So, each entry of the covariance matrix is computed as the sum of the squared distances between each point p_j belonging to the neighboorhod of point p_i and the centroid p_C. Then, this sum is divided by the number of points k used for the computation. Since $COV \in \mathbb{R}^{3,3}$, the number of its eigenvalues is equal to three as well as the number of associated eigenvectors. The adopted representation is $\lambda_1 \geq \lambda_2 \geq \lambda_3$ for the eigenvalues and u_1, u_2, u_3 for the respective eigenvectors. Before, it has been explained that PCA allows to find the principal components, which are the directions of maximal variation, as the eigenvectors of the covariance matrix, which is built upon the input data. In particular, the eigenvector associated to the greatest eigenvalue returns the direction of maximal variance while the eigenvector associated to the smallest eigenvalue returns the direction of minimal variance. In this case, the two eigenvectors, u_1 and u_2, associated to the largest and to the second largest eigenvalues, λ_1 and λ_2, define the tangent plane upon which point p_i and its k neighbors lay while the eigenvector u_3, associated to the smallest eigenvalue λ_3, is perpendicular to this plane. It means that the normal vector can be found by computing the eigenvector associated to the smallest eigenvalue of the covariance matrix. It is important to outline that not only the three components of the normal vector have to be defined but also its orientation, in such a way to completely define an oriented tangent plane. Indeed, when computing the eigenvector u_3 associated to the smallest eigenvalue λ_3 of the covariance matrix, it could be chosen either $N = u_3$ or $N = -u_3$. If the point cloud is obtained from range cameras, like the time – of – flight camera, it is easy to infer the orientation from the viewing direction. According to [29], if there is a single viewpoint v_p (i.e., the point from where data are captured) and it is known, the normal vector computed for point p_i, indicated by N_i, is correctly oriented if the following equation is satisfied:

$$\vec{N}_i \cdot (v_p - p_i) > 0$$

If there are multiple acquisition viewpoints, it become harder to correctly orient the normal, but this aspect is not covered in this thesis work since it is out of the scope.

In the context of surface reconstruction techniques, the importance of the definition of the p_i neighborhood has been already pointed out. In [4], some common approaches consist of defining a neighborhood by considering the k nearest points (KNNs) with respect to p_i or by taking into account all the points lying within an ε – ball, that is a spherical neighborhood centered on p_i and having radius ε. In this last case, all the k points that satisfy the property to be distant from p_i less than ε are considered as part of the p_i neighborhood. For example, in [31] the least square best fitting plane of a point p_i is determined by considering the neighborhood points that are contained in a sphere of radius r. In the case where the data contains little or no noise, k is not a crucial parameter since the output has been empirically observed to be stable over a wide range of settings [28]. In most of cases, it is usually better to choose a value of k that automatically changes, based on the considered point p_i. In more general applications, it is important to define an appropriate value for k since, if k is not large enough, the noise prevails over true data and the eigenvalues, computed through the PCA, tend to attain close values. On the other side, if k is too large, the points are much more widespread, and the surface curvature tends to increase the “thickness” of the neighborhood [28]. In the examined case, it is assumed a uniform sampling of the surface, so it is not necessary to adopt a complex technique to establish how many points shall be taken into account or which but simply the $k = 8$ nearest points are included.

3.2.3 Computation of the smallest eigenvector

Previously, it has been asserted that, for each point p_i of the point cloud, the vector which is normal to the surface can be computed by calculating the eigenvector u_3, associated to the smallest eigenvalue λ_3 of the covariance matrix COV. In particular, the covariance matrix is a positive – semidefinite symmetric matrix so it is included in the class of Hermitian matrices (a Hermitian matrix is a square complex matrix, which is equal to its conjugate transpose). For the class of Hermitian matrices, the most powerful and reliable algorithms are available [34]. The computation of the eigenvectors for a square 3×3 matrix is not so difficult to perform by hand but, in this case, it has to be developed to be performed by a FPGA so an iterative method shall be applied.

The simplest eigenvalue problem is to compute the highest eigenvalue along with its eigenvector and, for this task, the power method is the simplest iterative method [34]. Before explaining how the power method works, it is necessary to define the dominant eigenvalue. Given the n eigenvalues $\lambda_1, \dots, \lambda_n$ of a $n \times n$ matrix A, λ_1 is called the dominant eigenvector of A if $|\lambda_1| > |\lambda_i| \; \forall \; i = 2, \dots, n$.

Power method allows for the computation of the dominant eigenvalue and the associated eigenvector. The classical power method is iterative and the first step consists of making an assumption on the initial approximation of the dominant eigenvector, u_0. Then, it is computed the following sequence:

$$u_1 = Au_0$$

$$u_2 = Au_1 = A^2u_0$$

$$\vdots$$

$$u_k = Au_{k-1} = A^k u_0$$

The value k represents the maximum number of iterations. It could be chosen to stop before performing all the k iterations if, for example, the difference between the eigenvector computed at the step j and the eigenvector computed at the step $j - 1$ is smaller than a certain threshold. However, the power method tends to produce approximations with large entries so some methods have been developed, that scale down the approximation before proceeding to the next iteration. One way to accomplish this scaling is to determine the component of Au_i that has the largest absolute value and to divide each entry of Au_i by this value. In this way, the resulting vectors has components whose absolute values are less than or equal to 1. Then, if u_1 is the dominant eigenvector, the corresponding eigenvalue is computed by means of the so-called Rayleigh quotient:

$$\lambda_1 = \frac{Au_1 \cdot u_1}{u_1 \cdot u_1}$$

In [34], the following power method for Hermitian matrices is proposed:

ALGORITHM – Power Method for HEP

```
(1) Choose an initial approximation $u_0 = \bar{u}$ for the dominant eigenvector.
(2) for $k = 1,2,...$
(3)     $u = \frac{u_0}{||u_0||_2}$
(4)     $u_0 = Au$
(5)     $\lambda = u^T u_0$
(6)     if $\|u_0 - \lambda u\|_2 \le \varepsilon_{THR}|\lambda|$ stop
(7) end for
```

```
(8)  Dominant eigenvalue → $\lambda$
     Dominant eigenvector → $u$
```

In this case, the scaling procedure consists of dividing the result of the previous computation (or the initial guess for $k = 1$) by the 2-norm of the vector itself. Then, it is computed an estimate of the dominant eigenvector and the 2-norm of the difference between the newly computed eigenvector and the product between the previously computed eigenvector and the eigenvalue is compared with a threshold, defined as the product between the absolute value of the dominant eigenvalue and a certain coefficient ε_{THR}. Obviously, the smaller is ε_{THR}, the higher is the number of iterations necessary to achieve convergency and the better is the precision.

However, in the considered case of the normal estimation, it is needed to estimate the eigenvector associated to the smallest eigenvalue. For this reason, the inverse iteration method for Hermitian matrices explained in [34] can result really useful. First, let us introduce the algorithm:

ALGORITHM – Inverse power method

```
(1)  Choose an initial approximation $u_0 = \bar{u}$ for the dominant eigenvector.
(2)  **for** $k = 1,2,\ldots$
(3)      $u = \frac{u_0}{||u_0||_2}$
(4)      $u_0 = (A - \sigma I)^{-1} u$
(5)      $\theta = u^T u_0$
(6)      **if** $||u_0 - \theta u||_2 \le \varepsilon_{THR} |\theta|$ **stop**
(7)  **end for**
(8)  Dominant eigenvalue → $\lambda = \sigma + \frac{1}{\theta}$
     Dominant eigenvector → $u = \frac{u_0}{\theta}$
```

The inverse power method is based on the simple consideration that, if an eigenvalue is the smallest one for a matrix, it is also the largest for the inverse of the same matrix. So, it is possible to compute the dominant eigenvalue and the associated eigenvector for the inverse of the initial matrix and, at the end, the smallest eigenvalue of the initial matrix can be obtained as the reciprocal of the largest eigenvalue found for the inverse matrix. Moreover, since inversion is not always possible, it is sufficient to add or subtract to the entries of the matrix a little quantity, σ, called shift. In this case, if the matrix A is not invertible, it is possible to find a close approximation of its smallest eigenvector. Moreover, the inverse power method allows to find the eigenvalue closest to σ and, as we will see later, it results convenient for our algorithm, where surfaces are flat so the eigenvalues tend to be close to zero, as asserted in [30]. In general, if the eigenvalues are known, it is sufficient to choose σ very close to the desired eigenvalue and the inverse iteration can converge very quickly.

3.3 Normal estimation algorithm

Once all the theoretical bases have been introduced, it is possible to explain the algorithm developed in this thesis work. The main idea is to have a time – of – flight camera that returns a matrix of dimensions 25×25, for a total of 625 entries. Each entry of the matrix contains a point p_i of the inspected surface, so it is made up by three coordinates, $p_i = (x_i, y_i, z_i)$. The computation of the normal can be performed only for those points that are surrounded by 8 points. The single cell involved in the computations has the following structure:

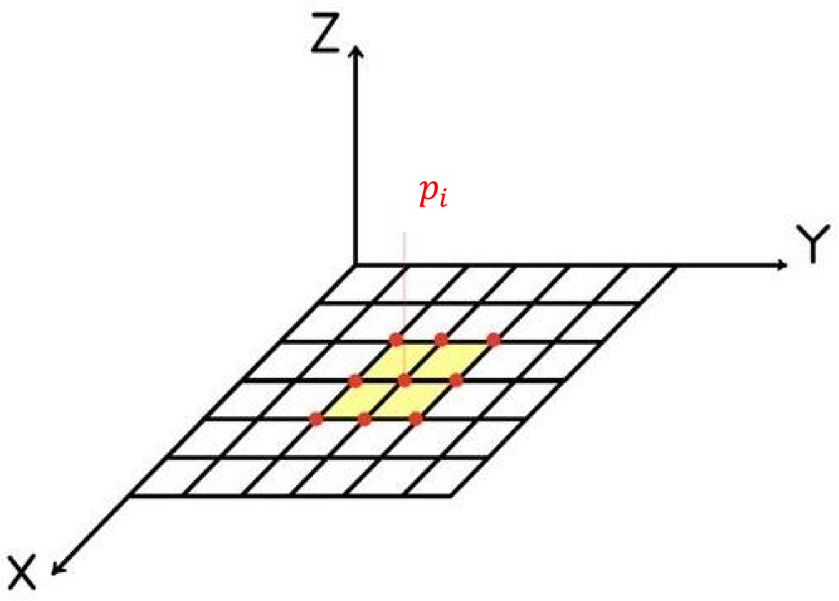


Figure 29 Single cell of the input matrix

It means that, for each point p_i with at least eight neighbors, the following steps are performed:

1. Computation of the centroid $p_C = (x_C, y_C, z_C)$ of the neighborhood, according to the formula:

$$x_C = \frac{1}{9}\sum_{i=1}^{9} x_i \qquad y_C = \frac{1}{9}\sum_{i=1}^{9} y_i \qquad z_C = \frac{1}{9}\sum_{i=1}^{9} z_i$$

2. For each point p_j belonging to the neighborhood of p_i, it is computed the Euclidean distance between the j^{th} point and the centroid p_C, according to the formula:

$$d_j = \sqrt{(x_j - x_C)^2 + (y_j - y_C)^2 + (z_j - z_C)^2} \qquad j = 1, \dots, 9$$

3. At this point, if the distance d_j between the j^{th} point and the centroid p_C is greater than a chosen threshold d_{THR}, it would be discarded and the algorithm proceeds with the computation of the Euclidean distance between

the successive point p_{j+1} and the centroid p_C. Otherwise, if the distance d_j is less than the threshold, the point p_j can be used for the computation of the covariance matrix Σ_j and a counter k is increased. For point p_j, the covariance matrix is given by:

$$\hat{\Sigma}_j = \begin{pmatrix} x_j - x_C \\ y_j - y_C \\ z_j - z_C \end{pmatrix} \begin{pmatrix} x_j - x_C & y_j - y_C & z_j - z_C \end{pmatrix}$$

$$\hat{\Sigma}_j = \begin{pmatrix} (x_j - x_C)^2 & (x_j - x_C)(y_j - y_C) & (x_j - x_C)(z_j - z_C) \\ (x_j - x_C)(y_j - y_C) & (y_j - y_C)^2 & (y_j - y_C)(z_j - z_C) \\ (x_j - x_C)(z_j - z_C) & (y_j - y_C)(z_j - z_C) & (z_j - z_C)^2 \end{pmatrix}$$

Each matrix $\hat{\Sigma}_j$ is added to the covariance matrix computed for the previous points, since the total covariance matrix is given by the sum of the matrices computed for each point p_j.

4. When the procedure to check how many points are at a distance from the centroid less than a certain threshold is completed, it is performed a control on k:

   - if the points within the threshold are less or equal to six, the algorithm cannot proceed with the computation of the normal and it is established that the three components of the normal vector are equal to zero. In this case, $N = (0,0,0)$.
   - If the points within the threshold are greater than six, the algorithm can proceed with the computation of the normal vector. Before going on with this step, it is necessary to divide all the entries of the covariance matrix $\hat{\Sigma}_i$ (where i represents the fact that the covariance matrix is computed for the i^{th} point of the input data) by the number of points k used for the computation of the matrix. The final formula for the covariance matrix is:

$$\hat{\Sigma}_i = \frac{1}{k}\sum_{j=1}^{k} \hat{\Sigma}_j = \frac{1}{k}\sum_{j=1}^{k} (p_j - p_C)(p_j - p_C)^T$$

5. Now, it is exploited the inverse power method for the computation of the eigenvector associated to the smallest eigenvalue of the covariance matrix $\hat{\Sigma}_i$, which corresponds to the normal vector centered on point p_i. The steps to be performed are shown here:

   5.1 Compute the corrected covariance matrix, $\hat{\Sigma}_{i,INV} = (\hat{\Sigma}_i - \sigma I)^{-1}$.

   5.2 Initialize the estimate for the eigenvector, $EV_0 = (1 \quad 1 \quad 1)$.

   5.3 Compute the normalized eigenvector, $EV = \frac{EV_0}{\|EV_0\|_2}$.

   5.4 Compute the new estimate for the eigenvector, $EV_0 = \hat{\Sigma}_i \cdot EV$.

   5.5 Compute the corresponding eigenvalue, $\theta = EV^T \cdot EV_0$.

   5.6 Compare the 2-norm of the vector that is computed as the difference between the estimate EV_0 and θEV with a threshold, defined as the product between the absolute value of the eigenvalue θ and a certain threshold δ. If $\|EV_0 - \theta EV\|_2 \leq |\theta|\delta$, then the algorithm can stop and the normal components are partially given by EV_0. Otherwise, it is necessary to restart from point 5.2. The algorithm can proceed until a maximum number of iterations, N_{MAX} is achieved. However, when computations are completed, the final normal components are set equal to the last computed value of EV_0, divided by the last computed value of θ.

3.4 Generation of the data and thresholds

To prove the algorithm, the main idea is to create a matrix data that represents a surface like the one shown in figure:

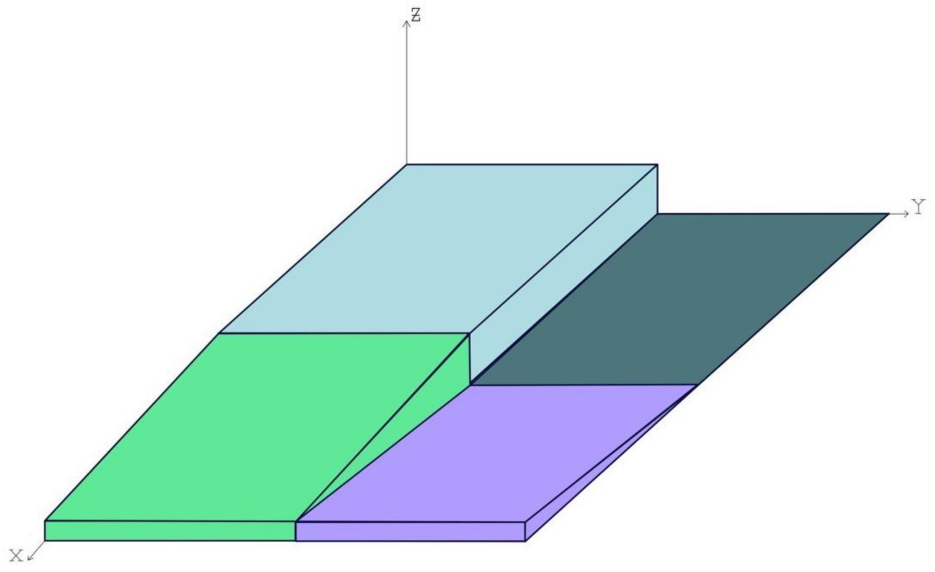


Figure 30 Surface used for the computations

The surface shown in figure should represent a portion of area of dimensions $5 \times 5\ mt$. Each axis is equally divided in 25 parts and it is considered a point for each of these parts, for a total of 625 points. So, the main idea is to uniformly sample the x and y axes representing this surface. For what regards the value of coordinate z, it can assume different values based on the considered region:

- Clear blue region – The coordinate z is equal to the highest value it could attain, that is $z_{HI} = 0.5\ mt$.
- Dark green region - The coordinate z is equal to the lowest value it could attain, that is $z_{LO} = 0\ mt$.
- Clear green region – The coordinate z takes on the values between z_{HI} and a middle value, $z_{MID} = 0.25\ mt$. The set of values between z_{HI} and z_{MID} are obtained by uniformly sampling the interval between the two extremes and the coordinate z tends to decrease in this region, as the x coordinate increases.

- Purple region – The coordinate z takes on the values between z_{MID} and z_{LO}. The set of values between z_{LO} and z_{MID} are obtained by uniformly sampling the interval between the two extremes and the coordinate z tends to increase in this region, as the coordinate increases.

The uniformly distributed samples, representing the surface, are placed as shown in the following picture:

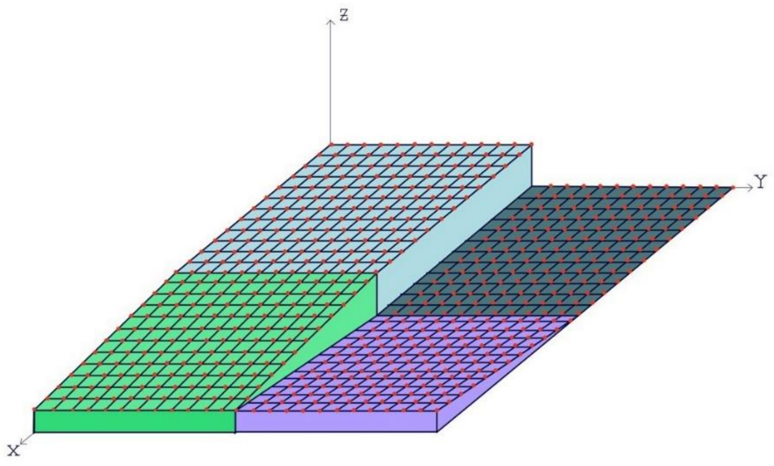


Figure 31 Sampling of the example surface

The points that are placed on the border between the blue light region and the dark green region, as well as the points on the border between the clear – green region and the purple region should not be used for the normal computation since these points represent the limit case for the implemented algorithm. The reason why the discussed data distribution has been chosen is to demonstrate that the algorithm is able to identify in which neighborhood is not possible to compute the normal vector and those in which the computation is feasible. According to the input data, the thresholds and fixed values that are adopted in the algorithm have been set through a trial – and – error procedure, by using some data whose results were already known and by exploiting this data in a C code. The chosen values are the following ones:

- The threshold for the Euclidean distance has been set equal to $d_{THR} = 0.3\ mt$. Then, considered a point p_i and its neighborhood, if a point p_j is distant from p_i more than $0.3\ mt$, it is discarded.

- The shift value used in the inverse power method algorithm is equal to $\sigma = 0.001$. In this way, the entries of the matrix are not so far from the original values, so that the final results are a good approximation of the original matrix eigenvectors.

- The maximum number of iterations has been set equal to $N_{MAX} = 256$.

- The threshold to assert if the new estimate of the eigenvector can be accepted or not is set equal to $\delta = 0.005$.

Chapter 4

Development of hardware unit

The aim of the thesis is not only to develop an algorithm for the computation of the normal vectors but also to implement it on a FPGA. Indeed, as previously explained in chapter 2, a FPGA allows for a high level of parallelism which could result really useful if considered the high amount of mathematical operations performed on the data.

4.1 Overview of embedded system

In general, embedded systems represent the computing elements embedded within an electronic device. The main characteristics that distinguish embedded systems are:

- the property to be single – functioned, so they repeatedly execute a single program;
- the fact that they are relatively low – cost and low – power, while ensuring small dimensions and good performances in terms of speed [1];
- the reactiveness, so they continuously react to changes in the system's environment and they are designed to compute certain results in real – time with almost zero delay.

There exist three main types of architecture, by means of which an embedded system can be realized:

- Single – purpose processors, that are digital circuit designed to execute one program. They are adopted as coprocessor or accelerators, so they are usually adopted to speed up computations. Then, single – purpose processors only contain the element necessary to execute a single program. The structure of a single – purpose processor is shown in Figure 32:

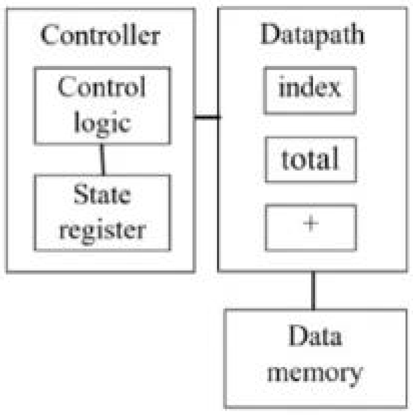


Figure 32 Single – purpose processor architecture

The data path is the set of components used to perform functional operations on data and to store them on registers. It is the entity in charge to perform the computation between primary inputs, which provide the data to be elaborated, and primary outputs, which return the results of computation. The data path includes a set of hardware resources, that could be storage, functional or interconnection units, and it defines how those modules are connected each other. All the RTL components can be allocated in different quantities and types and can be connected at design time through different interconnection schemes, like a mux or a bus. On the other side, the flow of the data through the different blocks is monitored and determined by the controller. The management of the computations as well as the handling of the data flow in the data path is performed by setting control signals values. Controller inputs may come from primary inputs, the so-called control inputs, or from data path components, such as status signals that come as results of comparisons. Finally the results of the performed operations, as well as the input values, can be stored in a data memory.

- Application – specific processors, that are programmable processors optimized for a particular class of applications having common characteristics. As it is possible to see in the figure below, in the architecture, one of the main differences with respect to the single – purpose processor is the presence of a program memory, which is used to store executable code, such as the instructions that the software needs to execute in order to perform the particular task for which the system has been designed. The other major difference is the presence of a more structured control unit, which reads instructions from memory, decodes them, and executes them. Another difference is the presence of a custom Arithmetic Logic Unit (ALU), which performs a wide range of arithmetic and logical operations.

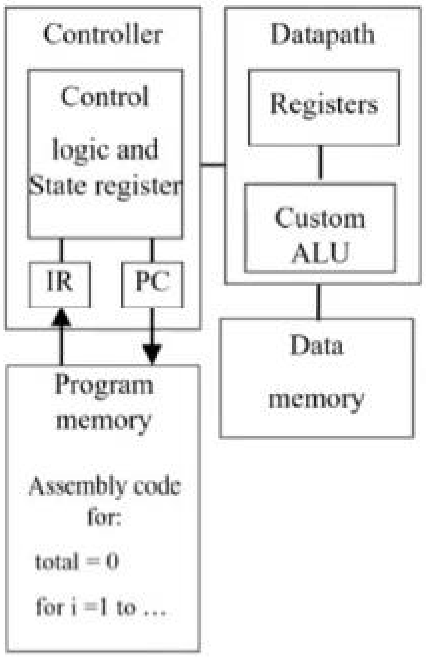


Figure 33 Application – specific processor architecture

- General-purpose processor, which is a type of central processing unit (CPU) that is designed to handle a wide variety of computational tasks, ranging from basic arithmetic operations to complex software programs. The architecture is very similar to the one of the application – specific processor but there is a general and not – custom ALU, which can perform a wider range of operations and the number of instructions that the processor can understand and execute is higher since general – purpose processors are designed to handle a wider range of tasks. The general architecture is shown in the picture below:

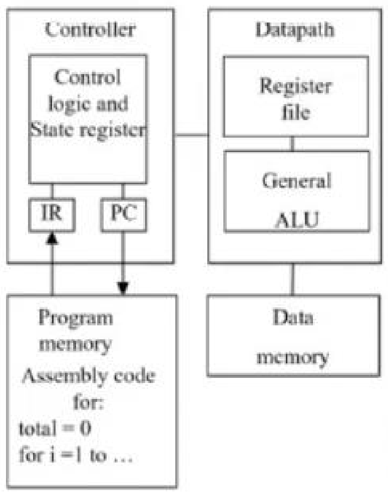


Figure 34 General – purpose processor architecture

Moreover, the design of embedded systems differs in the way in which a digital circuit implementation is mapped onto an integrated circuit. The design could be realized exploiting:

- Full – custom/VLSI, in which the digital implementation of the embedded system is fully customized so all the steps of placing transistors, sizing them and routing wires have to be performed.
- Semi – custom, in which lower layers are fully or partially built so designers have to handle with routing of wires and placing blocks.

Programmable Logic Devices (PLD), in which all layers already exist and connections between integrated circuits can be created or destroyed to implement desired functionality, belong to the second group. Here, it is possible to find FPGAs.

4.2 Register Transfer Level (RTL)

One of the main techniques used for the implementation of embedded system is the Register Transfer Level technique. It is a design technique in which a system is described in terms of data transfer between registers. The data are stored in registers after they have been subjected to a certain number of operations and the flow of data through registers is controlled by means a controller, which is implemented as a finite – state machine. The principal steps to implement an embedded system through the RTL design technique are:

1. Capture a high – level state machine, which described at high level the system's desired behavior. High level state machines are an extension of finite state machines, in which states and transitions are not only simple Boolean operations on single – bit inputs and outputs.
2. Create the data path which is in charge of carrying out the data operations of the high – level state machine.
3. Derive the controller, which is usually represented as a finite state machine. Then, it is necessary to convert the high – level state machine to a finite – state machine for the controller, by replacing data operation with setting and reading of control signal to and from the data path.
4. Connect the data path to the controller.

The most common architecture for the design of digital circuits by means of the register transfer level technique is the following one:

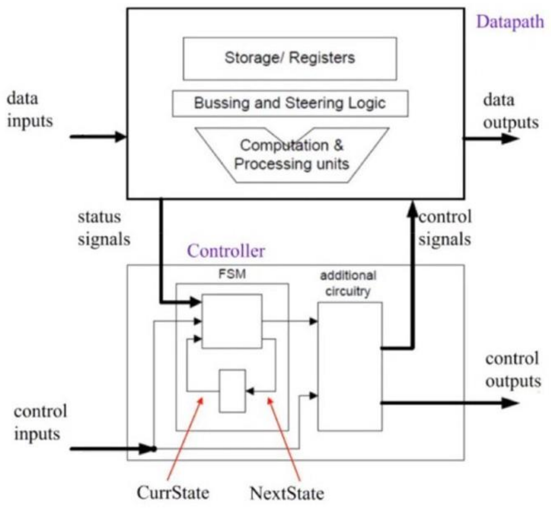


Figure 35 RTL Design Process architecture

4.3 Development of the algorithm

The main task of this chapter is to describe the adopted method to develop a hardware unit which implements the previously – discussed algorithm. In particular, the hardware – unit here proposed is a single – purpose processor, since it is realized to implement the only operations discussed in the algorithm. The design is carried on by exploiting the register transfer level (RTL) technique and the VHDL (VHSIC Hardware Description Language) is the language used for the description of the circuit, which implements the mathematical operations needed to perform the steps of the normal estimation algorithm.

The algorithm is characterized by several sequential operations that can be parallelized by means of the hardware realization of the algorithm. A reduction in the number of sequential operations allows for a reduction in the computation time so this common part of surface reconstruction algorithms, which is quite expensive from a time – consuming point of view, is well-suited to be implemented as a hardware accelerator. Moreover, the normal estimation algorithm' steps are applied for each point that belongs to the point cloud, so it is repeated a huge number of times. This means that, the simple hardware unit here – implemented, could be adopted in future works to optimize the speed of the global surface reconstruction algorithm. Before going on with the explanation, it is important to point out that the design of the digital circuit has been carried on by means of the Intel® software Quartus®.

At this point, the first step to be performed is describe the functions that the algorithm has to implement designing the high – level state machine, which described at high level the system's desired behavior. Since the steps needed to compute the three components of the normal vector are a significant number, the explanation is performed by dividing the subchapters based on the step of the algorithm that has to be performed. In general, it is important to highlight which type of description is chosen for each part of the digital circuit. Indeed, a digital circuit could be represented behaviorally, if only the behavior at a high level of abstraction is specified, or structurally, if the components used and the structure of the interconnection between the components are clearly specified. In this case, the data path is described in structural terms, the controller through a behavioral approach while the common interface between these two components is described structurally. As it has been highlighted previously, the circuit implementing the desired functions is realized through a register transfer level approach, so the result of computations are stored in registers. In particular, the design of the hardware unit is realized through a synchronous state machine so the flow of data is handled by the clock signal and the local storage are updated on clock edges only. Before going on with the explanation of data path and controller for each sub – unit implementing a step of the algorithm, it should be shown the elementary components exploited in the realization of the two subparts. For what regards simple components such as registers, multiplexers, or simple comparators, they have been designed by me while, for the arithmetical operations, the parametric IP cores provided by Intel® have been adopted. This latter choice is due to the fact that the required math operations involve floating point numbers so fast computations can be achieved only if well – optimized blocks, like the ones provided by Intel, are chosen.

4.3.1 Basic blocks

Between the basic blocks which have been designed to implement the algorithm, there are:

- REGISTERS - RegN;
- COUNTERS – Counter3, Counter4, Counter5, Counter8;
- COMPARATORS (EQUALS TO) – CMP_0, CMP_1, CMP_2, CMP_7, CMP_8, CMP_9, CMP_10, CMP_16, CMP_255, CMP_1_D8BIT;
- COMPARATORS (GREATER OR LESS THAN) – CMP_GT_6;
- MULTIPLEXERS - MUX_2TO1, MUX_2TO1_1BIT, MUX_4TO1;
- ROM MEMORY – X_ROM, Y_ROM, Z_ROM;
- OTHERS – CHANGE_K.

Moreover, it has been designed a block, called NORM_COMPUTATION, which is exploited all the times that the norm of a vector $v \in \mathbb{R}^{3,1}$ must be computed.

4.3.1.1 RegN

The component called RegN is a simple register, which behaves as a memory element when the input load signal Ld is OFF while it stores the input word D when the load signal is ON and there is a rising edge of the clock signal, Clk. Moreover, the register can be reset if the input signal reset Rst is ON. In this case, the component has a synchronous reset Rst since the reset operation is performed only if there is a rising edge of the clock signal Clk. The component is described as a parametric one, so the length of the register is defined by a parameter N that can be set in the data path where the component is instantiated.

4.3.1.2 Counter3, Counter4, Counter5, Counter8

These three components behave all at the same way, but they differ in the maximum value they can reach and, consequently, in the number of bits for each. Indeed, Counter3, Counter4, Counter5 and Counter8 contain respectively registers on 3 bits, 4 bits, 5 bits and 8 bits so they can respectively count up to 7, 15, 31 and 255. The four components are designed according to the same logic: when there is a rising edge of the input clock Clk, if the reset input signal Rst is equal to ‘1’, the output signal I is set equal to zero while, if the load input signal Ld is equal to ‘1’, the current value of the output is increased of a unit. In VHDL, it is not possible to read and write an output variable at the same time so, if the load input signal Ld is active, a signal called CURRENT_I is increased and the value of this signal is continuously assigned to the output variable I.

4.3.1.3 Comparators

These components are simple comparators that receive an input signal of variable length, depending on the variable they have to compare, and compare it with a given threshold, indicated in the name of the component itself. For example, CMP_7 receives as input a signal D whose length is 8 bits and compares it with the number 7 so that, if the two values are the same, its output D_EQS_7 is equal to ‘1’ while, if the two values are not equal, the output is equal to ‘0’. The only different component is CMP_GT_6. It is a comparator that receives as input a signal D whose length is 4 bits and the input value is compared with number six. If the input signal D is greater than six, the output signal D_GT_6 is set equal to ‘1’, otherwise it is set equal to ‘0’.

4.3.1.4 MUX_2TO1, MUX_2TO1_1BIT, MUX_4TO1

The components MUX_2TO1, MUX_2TO1_1BIT, MUX_4TO1 are three multiplexers, controlled by a selection signal. For what regards component MUX_2TO1 and MUX_2TO1_1BIT, the logic is the same: when the input selection signal SEL is equal to ‘0’, the output signal Q is equal to the first input signal D1 while, when the selection signal SEL is equal to ‘1’, the output signal Q is equal to the second input signal D2. The difference between the two components regards the length of the input signals D1 and D2 since they are declared as having length 1 bit in the component MUX_2TO1_1BIT while they are declared of generic length N in the component MUX_2TO1. Component MUX4TO1 is driven by the input selection signal SEL, whose length is 2 bits. Then, depending on the value taken by the selection signal, the output is equal to one of the four input signals, D1, D2, D3 and D4. As in MUX_2TO1, the input signals have generic length, expressed by a parameter N.

4.3.1.5 X_ROM, Y_ROM and Z_ROM

These three components are ROM memories, so each of them can be only read but not written. Inside each component, it is defined an array of 9 words, having length of 32 bits, as a new type. Then, it is defined a constant, called respectively ROMx, ROMy and ROMz, of the new defined type and each word of the array is initialized with some values. The input signal Address, having length of 4 bits, gives information about the array element that has to be read. These components are created since they are used to store the coordinates of the points that form the neighborhood for each point p_i. It is declared and instantiated a ROM memory for each type of coordinate so, due to the fact that a point p_i and its $k = 8$ neighbors have to be considered, each array shall contain nine elements.

4.3.1.6 CHANGE – K

This component is a simple converter that receives as input a signal D_4 having length of 4 bits and returns as output a signal D_16 containing the same value but on 32 bits. Since the component can be used only if the number k is greater than six, the only values it can take are 7, 8 and 9. Then, it is controlled if D_4 is equal to 7, 8 or 9 and then it is put equal to the corresponding number but on 32 bits and in floating point format.

4.3.2 IP Blocks

As it has been stated previously, the arithmetical operations that have to be performed in floating point numbers are quite expensive from a computational point of view so the choice of using predefined arithmetic blocks, which are fully optimized for what regards the clock latency, allows for an optimization of the computational resources and time. In this thesis work, the ALTERA_FP_FUNCTIONS IP core by Intel® has been used. The components of this library are fully configurable for what regards the latency and the frequency so it could be chosen to optimize either the latency, expressed in terms of clock cycles, or the frequency, expressed in MHz. The Quartus Prime® software provides a common interface, from which it is possible to choose the function that the block has to perform, the format of floating point data, the rounding operations, the available ports and the target parameter to be optimized. For each customizable block, the following input and output ports are available:

- Clk, to which all input signals must be synchronous;
- Areset, which is an asynchronous active – high reset;

- a, which is the first data input signal;
- b, which is the second data input signal;
- q, which represents the output data signal.

Moreover, an optional enable input signal can be selected in such a way to allow for a block to be activated only when this signal is high. Moreover, the rounding option can be chosen, in such a way to speed up the computation in exchange of a most approximated final result. The IP blocks provided by Quartus were used to implement the following functions:

- Adder, with a latency of 7 clock cycles;
- Subtractor, with a latency of 10 clock cycles;
- Multiplier, with a latency of 9 clock cycles;
- Divider, with a latency of 16 clock cycles;
- Comparator, with a latency of 2 clock cycles;
- Square root, with a latency of 9 clock cycles;
- Absolute value, with a latency of 1 clock cycle.

In the slowest components, like the divider or the square root, the rounding option was set in such a way to optimize the speed of the implemented circuit.

4.3.3 Explanation of the architecture

To better understand the designed architecture, the explanation will follow the division in paragraphs introduced when the normal estimation algorithm was explained in Chapter 3. The two computational units, the data path and the control unit, are interfaced through the NE1 interface. The signals that command both units are the CLOCK signal and the RESET signal.

4.3.3.1 Computation of the centroid

The state S0 of control unit is entirely dedicated to set the load signals of all registers and counter equal to ‘0’, the clear signals of all registers and counters equal to ‘1’ and the enable signals of all arithmetic blocks are set equal to ‘0’. After this first setup state, it is carried out the computation of the centroid coordinates. As explained before, the computation of the centroid consists of two main operations, which are the sum of the coordinates of the nine points and the division of this sum by the number of points, that is equal to 9. To perform this task, the following components are instantiated in the data path:

TYPE OF COMPONENT	NAME	ALREADY INSTANTIATED
RegN (32-bits)	XC1	No
RegN (32-bits)	YC1	No
RegN (32-bits)	ZC1	No
RegN (32-bits)	XI	No
RegN (32-bits)	YI	No
RegN (32-bits)	ZI	No
RegN (32-bits)	XC	No
RegN (32-bits)	YC	No
RegN (32-bits)	ZC	No
Adder (IP Block)	ADDx	No
Adder (IP Block)	ADDy	No
Adder (IP Block)	ADDz	No
Multiplier (IP Block)	MULTx	No
Multiplier (IP Block)	MULTy	No
Multiplier (IP Block)	MULTz	No
Counter3	OP1_CNT	No
Counter4	OP2_CNT	No
Counter4	I_CNT	No

CMP_7	OP1_CMP_7	No
CMP_9	OP2_CMP_9	No
CMP_9	I_CMP_9	No
X_ROM	ROMx	No
Y_ROM	ROMy	No
Z_ROM	ROMz	No

Table 1 List of components in the data path

The signals which are exchanged between the data path and the control unit are listed below:

TYPE OF SIGNAL	FROM	TO	NAME
Load signal	Control Unit	Data path	LOAD_XC1
Load signal	Control Unit	Data path	LOAD_YC1
Load signal	Control Unit	Data path	LOAD_ZC1
Load signal	Control Unit	Data path	LOAD_XI
Load signal	Control Unit	Data path	LOAD_YI
Load signal	Control Unit	Data path	LOAD_ZI
Load signal	Control Unit	Data path	LOAD_XC
Load signal	Control Unit	Data path	LOAD_YC
Load signal	Control Unit	Data path	LOAD_ZC
Clear signal	Control Unit	Data path	CLEAR_XC1
Clear signal	Control Unit	Data path	CLEAR_YC1
Clear signal	Control Unit	Data path	CLEAR_ZC1
Clear signal	Control Unit	Data path	CLEAR_XI
Clear signal	Control Unit	Data path	CLEAR_YI
Clear signal	Control Unit	Data path	CLEAR_ZI
Clear signal	Control Unit	Data path	CLEAR_XC
Clear signal	Control Unit	Data path	CLEAR_YC
Clear signal	Control Unit	Data path	CLEAR_ZC
Clear signal	Control Unit	Data path	CLEAR_OP1

Clear signal	Control Unit	Data path	CLEAR_OP2
Clear signal	Control Unit	Data path	CLEAR_I
Enable for counters	Control Unit	Data path	INC_OP1
Enable for counters	Control Unit	Data path	INC_OP2
Enable for counters	Control Unit	Data path	INC_I
Enable for math blocks	Control Unit	Data path	EN_ADDER1
Enable for math blocks	Control Unit	Data path	EN_MULT1
Results of comparisons	Data path	Control Unit	OP1_EQS_7
Results of comparisons	Data path	Control Unit	OP2_EQS_9
Results of comparisons	Data path	Control Unit	I_EQS_9

Table 2 List of signals exchanged between control unit and data path

The counter I_CNT allows to cycle on the elements of the memories ROMx, ROMy and ROMz. In parallel, the coordinates x, y, z of the i^{th} point are loaded on registers XI, YI, ZI. It is performed the sum between the value contained on these registers and those contained on registers XC1, YC1, ZC1 by means of the adders ADDx, ADDy, ADDz. Since each adder has a latency of 7 clock cycles, the OP1_CNT is exploited to count till seven. When the OP1_CNT finishes to count 7 clock cycles, the signal OP1_EQS_7 becomes equal to '1', so the control unit understands that the sum operations are completed and the results of the computations are respectively stored on registers XC1, YC1, ZC1. To check if all the points in ROMx, ROMy and ROMz have been taken into account for the computation of the sum of coordinates, the comparator I_CMP_9 provides a signal called I_EQS_9. When this latter signal becomes equal to '1', the controller understands that all the points have been considered. The sums of the nine values for the three types of coordinates are finally stored on XC1, YC1, ZC1 so the division by nine has to performed. Since the division operation is quite expensive from a point of view of latency, it could be performed a multiplication by the inverse of nine, which is approximately equal to 0.11. This value is stored as a constant, CONSTANT_1, and it is fed as input to the MULT1 block. Since the multiplier needs for 9 clock cycles, the OP2_CNT is exploited to count till nine. When the nine clock cycles have been counted, the signal OP2_EQS_9 becomes equal to '1' so that the control unit understands that the multiplier can be disenabled and the final results are stored on registers XC, YC, ZC.

4.3.3.2 Computation of Euclidean distance

After the computation of the centroid, it is required to compute the Euclidean distance between each point p_i from the nine in the considered neighborhood and the centroid. In the data path, the following components are involved in the operations that led to the final result:

TYPE OF COMPONENT	NAME	ALREADY INSTANTIATED
RegN (32-bits)	XI	Yes
RegN (32-bits)	YI	Yes
RegN (32-bits)	ZI	Yes
RegN (32-bits)	XC	Yes
RegN (32-bits)	YC	Yes
RegN (32-bits)	ZC	Yes
RegN (32-bits)	DIFFX1	No
RegN (32-bits)	DIFFY1	No
RegN (32-bits)	DIFFZ1	No
RegN (32-bits)	DIFFX2	No
RegN (32-bits)	DIFFY2	No
RegN (32-bits)	DIFFZ2	No
RegN (32-bits)	SQR_XY	No
RegN (32-bits)	SQR_XYZ	No
RegN (1-bit)	RESULT_1	No

Subtractor (IP Block)	SUBx	No
Subtractor (IP Block)	SUBy	No
Subtractor (IP Block)	SUBz	No
Multiplier (IP Block)	MULTx_2	No
Multiplier (IP Block)	MULTy_2	No
Multiplier (IP Block)	MULTz_2	No
Adder (IP Block)	ADDxy	No
Adder (IP Block)	ADDxyz	No
Comparator (IP Block)	CMP_THR	No
Counter4	I_CNT	Yes
Counter4	K_CNT	No
Counter3	OP1_CNT	Yes
Counter4	OP2_CNT	Yes
CMP_1	RESULT1_CMP_1	No
CMP_7	OP1_CMP_7	Yes
CMP_9	OP2_CMP_9	Yes
CMP_10	OP2_CMP_10	No
CMP_2	OP1_CMP_2	No
X_ROM	ROMx	Yes
Y_ROM	ROMy	Yes
Z_ROM	ROMz	Yes

Table 3 List of components in data path

The signals which are exchanged between the data path and the control unit are listed below:

TYPE OF SIGNAL	FROM	TO	NAME
Load signal	Control Unit	Data path	LOAD_XI
Load signal	Control Unit	Data path	LOAD_YI
Load signal	Control Unit	Data path	LOAD_ZI

Load signal	Control Unit	Data path	LOAD_DIFFX1
Load signal	Control Unit	Data path	LOAD_DIFFY1
Load signal	Control Unit	Data path	LOAD_DIFFZ1
Load signal	Control Unit	Data path	LOAD_DIFFX2
Load signal	Control Unit	Data path	LOAD_DIFFY2
Load signal	Control Unit	Data path	LOAD_DIFFZ2
Load signal	Control Unit	Data path	LOAD_SQR_XY
Load signal	Control Unit	Data path	LOAD_SQR_XYZ
Load signal	Control Unit	Data path	LOAD_RESULT1
Clear signal	Control Unit	Data path	CLEAR_XI
Clear signal	Control Unit	Data path	CLEAR_YI
Clear signal	Control Unit	Data path	CLEAR_ZI
Load signal	Control Unit	Data path	CLEAR_DIFFX1
Load signal	Control Unit	Data path	CLEAR_DIFFY1
Load signal	Control Unit	Data path	CLEAR_DIFFZ1
Load signal	Control Unit	Data path	CLEAR_DIFFX2
Load signal	Control Unit	Data path	CLEAR _DIFFY2
Load signal	Control Unit	Data path	CLEAR _DIFFZ2
Load signal	Control Unit	Data path	CLEAR _SQR_XY
Load signal	Control Unit	Data path	CLEAR _SQR_XYZ
Load signal	Control Unit	Data path	CLEAR _RESULT1
Clear signal	Control Unit	Data path	CLEAR_OP1
Clear signal	Control Unit	Data path	CLEAR_OP2
Clear signal	Control Unit	Data path	CLEAR_I
Clear signal	Control Unit	Data path	CLEAR_K
Enable for counters	Control Unit	Data path	INC_OP1
Enable for counters	Control Unit	Data path	INC_OP2
Enable for counters	Control Unit	Data path	INC_I
Enable for counters	Control Unit	Data path	INC_K

Enable for math blocks	Control Unit	Data path	EN_SUB1
Enable for math blocks	Control Unit	Data path	EN_MULT2
Enable for math blocks	Control Unit	Data path	EN_ADD2
Enable for math blocks	Control Unit	Data path	EN_ADD3
Enable for math blocks	Control Unit	Data path	EN_CMP_THR
Results of comparisons	Data path	Control Unit	OP1_EQS_7
Results of comparisons	Data path	Control Unit	OP2_EQS_9
Results of comparisons	Data path	Control Unit	OP2_EQS_10
Results of comparisons	Data path	Control Unit	OP1_EQS_2
Results of comparisons	Data path	Control Unit	I_EQS_9
Results of comparisons	Data path	Control Unit	RESULT1_EQS_1

Table 4 List of signals exchanged between control unit and data path

For each point whose coordinates are kept on ROMx, ROMy and ROMz, the corresponding registers XI, YI and ZI are loaded to store the values. To cycle between the nine coordinates, the I_CNT counter is exploited and, when it reaches the 9 values, the I_EQS_9 signal becomes equal to '1'. In this way, the control unit understands that all the points have been taken into account and the following steps could be performed. Once XI, YI and ZI are loaded with the coordinates of the i^{th} point, it is performed the subtraction between the values store on registers XI, YI and ZI and the values stored on XC, YC, ZC. Since the subtractor needs for 10 clock cycles, the OP2_CNT is exploited to count till ten. Once the ten clock cycles are counted, the signal OP2_EQS_10 becomes equal to '1', so that the control unit understands that the subtractor can be disenabled. The results of these operations are stored on registers DIFFX1, DIFFY1 and DIFFZ1. Then, these differences have to be squared so it is performed the square of these values through a multiplication between the same inputs. Since the multiplication needs for 9 clock cycles, the OP2_CNT is exploited to count till nine and, once the needed clock cycles are counted, the signal OP2_EQS_9 becomes high so that the controller unit understands that the multiplier can be disenabled and the results of the operations are stored on registers DIFFX2, DIFFY2 and DIFFZ2. Then it is performed the sum between the elements contained on DIFFX2 and DIFFY2, whose result is stored on register SQR_XY. Successively, the value on SQR_XY is added to DIFFZ2 and the final result is stored on register SQR_XYZ. Both additions require 7 clock cycles to be completed so OP1_CNT is exploited to count 7 clock cycles and, once the needed clock cycles are counted, the signal OP1_EQS_7 becomes high so that the control unit understands that ADDxy before and ADDxyz after can be disenabled. At this point, it should be computed the square root of the Euclidean distance so that it could be compared to the given threshold, which is stored on CONSTANT_2. Since the square root is quite expensive from a point of view of clock latency, it is performed the comparison between the square Euclidean distance and the square of the given threshold. The comparison between the two values is performed by CMP_THR, which receives as input the value stored on SQR_XYZ and CONSTANT_2 and it returns ‘1’ if SQR_XYZ is less than CONSTANT_2 or it returns ‘0’ if SQR_XYZ is greater than CONSTANT_2. Since the comparator needs 2 clock cycles, the OP1_CNT is exploited to count them and the signal OP1_EQS_2 becomes equal to ‘1’ after two clock cycles. In this way, the controller unit understands that the comparator can be disenabled. The result of the computation is stored on the register RESULT_1 and it is fed as input to the comparator RESULT1_CMP_1, which provides as output the signal RESULT1_EQS_1, which is equal to ‘1’ if RESULT_1 = ‘1’ or ‘0’ if RESULT_1 = ‘0’. If RESULT1_EQS_1 is equal to ‘1’, the controller enables counter CNT_K to increase. In this case, the point p_i, whose Euclidean distance from the centroid has been computed, is exploited to compute the associated covariance matrix. Otherwise, if RESULT1_EQS_1 = ‘0’, CNT_I is increased and it is performed the computation of the Euclidean distance for the next points, but only if I_EQS_9 is different from ‘0’.

4.3.3.3 Computation of covariance matrix

At this stage, if point p_i is far from the centroid less than the given threshold, it can be involved in the computation of the covariance matrix. In particular, since it is symmetric, it is sufficient to compute just six entries of the matrix, instead of nine. The components that are interested in the computation of the covariance matrix are the following ones:

TYPE OF COMPONENT	NAME	ALREADY INSTANTIATED
RegN (32-bits)	DIFFX1	Yes
RegN (32-bits)	DIFFY1	Yes
RegN (32-bits)	DIFFZ1	Yes
RegN (32-bits)	DIFFX2	Yes
RegN (32-bits)	DIFFY2	Yes
RegN (32-bits)	DIFFZ2	Yes
RegN (32-bits)	XY	No
RegN (32-bits)	XZ	No
RegN (32-bits)	YZ	No
RegN (32-bits)	COV1	No
RegN (32-bits)	COV2	No
RegN (32-bits)	COV3	No
RegN (32-bits)	COV12	No
RegN (32-bits)	COV13	No
RegN (32-bits)	COV23	No
RegN (32-bits)	COVXX	No
RegN (32-bits)	COVYY	No
RegN (32-bits)	COVZZ	No

RegN (32-bits)	COVXY	No
RegN (32-bits)	COVXZ	No
RegN (32-bits)	COVYZ	No
Multiplier (IP Block)	MULTxy	No
Multiplier (IP Block)	MULTxz	No
Multiplier (IP Block)	MULTyz	No
Adder (IP Block)	SUMxx	No
Adder (IP Block)	SUMyy	No
Adder (IP Block)	SUMzz	No
Adder (IP Block)	SUMxy	No
Adder (IP Block)	SUMxz	No
Adder (IP Block)	SUMyz	No
Divider (IP Block)	DIVxx	No
Divider (IP Block)	DIVyy	No
Divider (IP Block)	DIVzz	No
Divider (IP Block)	DIVxy	No
Divider (IP Block)	DIVxz	No
Divider (IP Block)	DIVyz	No
Counter3	OP1_CNT	Yes
Counter4	OP2_CNT	Yes
Counter5	OP3_CNT	No
Counter4	I_CNT	Yes
Counter4	K_CNT	Yes
CMP_7	OP1_CMP_7	Yes
CMP_9	OP2_CMP_9	Yes
CMP_16	OP3_CMP_16	No
CMP_GT_6	K_CMP_6	No
CHANGE_K	CHANGE_K	No

Table 5 List of components in data path

The signals which are exchanged between the data path and the control unit are listed below:

TYPE OF SIGNAL	FROM	TO	NAME
Load signal	Control Unit	Data path	LOAD_XY
Load signal	Control Unit	Data path	LOAD_XZ
Load signal	Control Unit	Data path	LOAD_YZ
Load signal	Control Unit	Data path	LOAD_COV1
Load signal	Control Unit	Data path	LOAD_COV2
Load signal	Control Unit	Data path	LOAD_COV3
Load signal	Control Unit	Data path	LOAD_COV12
Load signal	Control Unit	Data path	LOAD_COV13
Load signal	Control Unit	Data path	LOAD_COV23
Load signal	Control Unit	Data path	LOAD_COVXX
Load signal	Control Unit	Data path	LOAD_COVYY
Load signal	Control Unit	Data path	LOAD_COVZZ
Load signal	Control Unit	Data path	LOAD_COVXY
Load signal	Control Unit	Data path	LOAD_COVXZ
Load signal	Control Unit	Data path	LOAD_COVYZ
Clear signal	Control Unit	Data path	CLEAR_XY
Clear signal	Control Unit	Data path	CLEAR_XZ
Clear signal	Control Unit	Data path	CLEAR_YZ
Clear signal	Control Unit	Data path	CLEAR_COV1
Clear signal	Control Unit	Data path	CLEAR_COV2
Clear signal	Control Unit	Data path	CLEAR_COV3
Clear signal	Control Unit	Data path	CLEAR_COV12
Clear signal	Control Unit	Data path	CLEAR_COV13
Clear signal	Control Unit	Data path	CLEAR_COV23
Clear signal	Control Unit	Data path	CLEAR_COVXX

Clear signal	Control Unit	Data path	CLEAR_COVYY
Clear signal	Control Unit	Data path	CLEAR_COVZZ
Clear signal	Control Unit	Data path	CLEAR_COVXY
Clear signal	Control Unit	Data path	CLEAR_COVXZ
Clear signal	Control Unit	Data path	CLEAR_COVYZ
Clear signal	Control Unit	Data path	CLEAR_OP1
Clear signal	Control Unit	Data path	CLEAR_OP2
Clear signal	Control Unit	Data path	CLEAR_OP3
Clear signal	Control Unit	Data path	CLEAR_I
Enable for counters	Control Unit	Data path	INC_OP1
Enable for counters	Control Unit	Data path	INC_OP2
Enable for counters	Control Unit	Data path	INC_OP3
Enable for math blocks	Control Unit	Data path	EN_MULT3
Enable for math blocks	Control Unit	Data path	EN_ADDER4
Enable for math blocks	Control Unit	Data path	EN_DIVIDER1
Results of comparisons	Data path	Control Unit	OP1_EQS_7
Results of comparisons	Data path	Control Unit	OP2_EQS_9
Results of comparisons	Data path	Control Unit	OP3_EQS_16
Results of comparisons	Data path	Control Unit	K_GT_6

Table 6 List of signals exchanged between control unit and data path

Three of the six components of the covariance matrix have been already computed:

- $(x_i - x_C)^2$, which is stored on DIFFX2.
- $(y_i - y_C)^2$, which is stored on DIFFY2.

- $(z_i - z_C)^2$, which is stored on DIFFZ2.

The remaining entries of the covariance matrix which have to be computed are:

- $(x_i - x_C)(y_i - y_C)$, which can be computed as the product between the value stored on DIFFX1 and DIFFY1.
- $(x_i - x_C)(z_i - z_C)$, which can be computed as the product between the value stored on DIFFX1 and DIFFZ1.
- $(y_i - y_C)(z_i - z_C)$, which can be computed as the product between the value stored on DIFFY1 and DIFFZ1.

These three products are executed in parallel and they are performed through the multipliers MULTxy, MULTxz and MULTyz. As explained before, the multiplication needs for 9 clock cycles to be completed, so the OP2_CNT is exploited to count 9 clock cycles and, once this value is reached, the signal OP2_EQS_9 becomes equal to '1'. In this way, the controller unit understands that the multipliers can be disenabled and the results of the operations are stored on registers XY, XZ and YZ. Then, each entry of the covariance matrix has to be summed up with the ones computed for previous points so, for each point p_i, its entries are added to the ones computed previously and stored on registers COV1, COV2, COV3, COV12, COV13 and COV23. The operation is performed through adders SUMxx, SUMyy, SUMzz, SUMxy, SUMxz, SUMyz. As explained for previous adders, these components require 7 clock cycles to complete the operation and, for this purpose, OP1_CNT counts up to 7 clock cycles. When its stored value is equal to seven, the control unit receives the input signal OP1_EQS_7 from the comparator OP1_CMP_7, which reports that the adders can be disactivated.

When all the nine points of the neighborhood have been considered, the counter K_CNT reports how many of those points have been used for the computation of the covariance matrix and consequently how many points are close to the centroid less than the chosen threshold. When the I_EQS_9 becomes equal to '1', the controller unit checks the value of K_CMP_6 output: if the signal K_GT_6 is equal to '1', it means that more than six points have been exploited for the computation of the covariance matrix so the successive steps can be considered. On the other side, if the signal K_GT_6 is equal to ‘0’, less than six or six points are sufficiently close to the centroid so the control unit directly jumps to the state where the three components of the normal vector are set equal to 0. On the other case, it is performed a division between the six elements of the covariance matrix stored on register COV1, COV2, COV3, COV12, COV13, COV23 and the value stored on K_CNT. Since the value stored on K_CNT is 4-bit long and it is an integer, it has to be converted as a 32-bit value and it has to be expressed in the floating-point format. For this reason, the K_CNT output is passed as input to the block CHANGE_K, which converts the number in the desired format and its output enters as second input in the six dividers, whose first input is the corresponding entry of the covariance matrix. The six divisions are executed in parallel but each of them require 16 clock cycles to be carried out. The 16 clock cycles are counted up through counter OP3_CNT and when it counts sixteen, the signal OP3_EQS_16 becomes equal to ‘1’. In this way, the control unit understands that the dividers can be disenabled and the results of the operations are stored on registers COVXX, COVYY, COVZZ, COVXY, COVXZ, COVYZ.

4.3.3.4 Estimation of normal vector

The next step is the estimation of the eigenvector associated to the smallest eigenvalue of the covariance matrix. As highlighted in Chapter 3, there are six sub-steps to perform in order to compute the desired quantities. Moreover, the mathematical computations behind the realization of this part of the data path and the controller unit are explained in Appendix A.

<u>Compute the corrected covariance matrix.</u>

First, it is necessary to compute the shifted covariance matrix, $\hat{\Sigma}_i - \sigma I$. This matrix differs from the covariance matrix computed at the previous step only in the diagonal entries: indeed, the shift value must be subtracted to the elements in the diagonal, while the others remain unchanged. The tables below list the components which are involved in this part of the architecture:

TYPE OF COMPONENT	NAME	ALREADY INSTANTIATED
RegN (32-bits)	COVXX	Yes
RegN (32-bits)	COVYY	Yes
RegN (32-bits)	COVZZ	Yes
RegN (32-bits)	COVXXs	No
RegN (32-bits)	COVYYs	No
RegN (32-bits)	COVZZs	No
Subtractor (IP Block)	SUB_shiftXX	No
Subtractor (IP Block)	SUB_shiftYY	No
Subtractor (IP Block)	SUB_shiftZZ	No
Counter4	OP2_CNT	Yes
CMP_10	OP2_CMP_10	Yes

Table 7 List of components in data path

The signals which are exchanged between the data path and the control unit are listed below:

TYPE OF SIGNAL	FROM	TO	NAME
Load signal	Control Unit	Data path	LOAD_COVXXs
Load signal	Control Unit	Data path	LOAD_COVYYs
Load signal	Control Unit	Data path	LOAD_COVZZs
Clear signal	Control Unit	Data path	CLEAR_COVXXs
Clear signal	Control Unit	Data path	CLEAR_COVYYs
Clear signal	Control Unit	Data path	CLEAR_COVZZs
Enable for counters	Control Unit	Data path	INC_OP2
Clear signal	Control Unit	Data path	CLEAR_OP2
Enable for math blocks	Control Unit	Data path	EN_SUB3
Results of comparisons	Data path	Control Unit	OP2_EQS_10

Table 8 List of signals exchanged between control unit and data path

The values stored on registers COVXX, COVYY and COVZZ are fed as input to subtractors SUB_shiftXX, SUB_shiftYY and SUB_shiftZZ, together with the value of the shift, which is stored on CONSTANT_6. Since the subtraction needs for ten clock cycles to be completed, the counter OP2_CNT counts up to ten and, when this value is reached, the signal OP2_EQS_10 becomes equal to '1'. In this way, the control unit understands that subtraction operation is complete and set EN_SUB3 = '0'. The final results of subtractions are stored on registers COVXXs, COVYYs, COVZZs.

Computation of the inverse shifted – covariance matrix.

This part of the algorithm is one of the most computationally heavy, so the mathematical operations involved in the computation of the entries of the inverse matrix and those involved in the computation of the determinant have been fully parallelized. First, the involved components are shown in the table below:

TYPE OF COMPONENT	NAME	ALREADY INSTANTIATED
RegN (32-bits)	COVXXs	Yes
RegN (32-bits)	COVYYs	Yes
RegN (32-bits)	COVZZs	Yes
RegN (32-bits)	COVXY	Yes
RegN (32-bits)	COVXZ	Yes
RegN (32-bits)	COVYZ	Yes
RegN (32-bits)	AUX1_MULT6	No
RegN (32-bits)	AUX2_MULT6	No
RegN (32-bits)	AUX3_MULT6	No
RegN (32-bits)	AUX4_MULT6	No
RegN (32-bits)	AUX5_MULT6	No
RegN (32-bits)	AUX6_MULT6	No
RegN (32-bits)	AUX7_MULT6	No
RegN (32-bits)	AUX8_MULT6	No

RegN (32-bits)	AUX9_MULT6	No
RegN (32-bits)	AUX10_MULT6	No
RegN (32-bits)	AUX11_MULT6	No
RegN (32-bits)	AUX12_MULT6	No
RegN (32-bits)	AUX13_MULT6	No
Multiplier (IP Block)	MULT6_1	No
Multiplier (IP Block)	MULT6_2	No
Multiplier (IP Block)	MULT6_3	No
Multiplier (IP Block)	MULT6_4	No
Multiplier (IP Block)	MULT6_5	No
Multiplier (IP Block)	MULT6_6	No
Multiplier (IP Block)	MULT6_7	No
Multiplier (IP Block)	MULT6_8	No
Multiplier (IP Block)	MULT6_9	No
Multiplier (IP Block)	MULT6_10	No
Multiplier (IP Block)	MULT6_11	No
Multiplier (IP Block)	MULT6_12	No
Multiplier (IP Block)	MULT6_13	No
RegN (32-bits)	AUX1_MULT7	No
RegN (32-bits)	AUX2_MULT7	No
RegN (32-bits)	AUX3_MULT7	No
RegN (32-bits)	AUX4_MULT7	No
RegN (32-bits)	AUX5_MULT7	No
RegN (32-bits)	AUX6_MULT7	No
RegN (32-bits)	AUX7_MULT7	No
Multiplier (IP Block)	MULT7_1	No
Multiplier (IP Block)	MULT7_2	No
Multiplier (IP Block)	MULT7_3	No
Multiplier (IP Block)	MULT7_4	No

Multiplier (IP Block)	MULT7_5	No
Multiplier (IP Block)	MULT7_6	No
Multiplier (IP Block)	MULT7_7	No
RegN (32-bits)	AUX1_ADDER9	No
RegN (32-bits)	AUX2_ ADDER9	No
RegN (32-bits)	AUX3_ ADDER9	No
Adder (IP Block)	ADDER9_1	No
Adder (IP Block)	ADDER9_2	No
Adder (IP Block)	ADDER9_3	No
RegN (32-bits)	AUX_ADDER10	No
Adder (IP Block)	ADDER10	No
RegN (32-bits)	AUX1_SUB3	No
RegN (32-bits)	AUX2_SUB3	No
RegN (32-bits)	AUX3_SUB3	No
RegN (32-bits)	AUX4_SUB3	No
RegN (32-bits)	AUX5_SUB3	No
Subtractor (IP Block)	SUB3_1	No
Subtractor (IP Block)	SUB3_2	No
Subtractor (IP Block)	SUB3_3	No
Subtractor (IP Block)	SUB3_3	No
Subtractor (IP Block)	SUB3_4	No
RegN (32-bits)	INV_COVXX	No
RegN (32-bits)	INV_COVYY	No
RegN (32-bits)	INV_COVZZ	No
RegN (32-bits)	INV_COVXY	No
RegN (32-bits)	INV_COVXZ	No
RegN (32-bits)	INV_COVYZ	No
Divider (IP Block)	DIV4_1	No
Divider (IP Block)	DIV4_2	No

Divider (IP Block)	DIV4_3	No
Divider (IP Block)	DIV4_5	No
Divider (IP Block)	DIV4_5	No
Divider (IP Block)	DIV4_6	No
Counter3	OP1_CNT	Yes
Counter4	OP2_CNT	Yes
Counter5	OP3_CNT	Yes
CMP_7	OP1_CMP_7	Yes
CMP_9	OP2_CMP_9	Yes
CMP_10	OP2_CMP_10	Yes
CMP_16	OP3_CMP_16	Yes

Table 9 List of components in data path

The signals which are exchanged between the data path and the control unit are listed below:

TYPE OF SIGNAL	FROM	TO	NAME
Load signal	Control Unit	Data path	LOAD_COVXXs
Load signal	Control Unit	Data path	LOAD_COVYYs
Load signal	Control Unit	Data path	LOAD_COVZZs
Load signal	Control Unit	Data path	LOAD_AUX_MULT6
Load signal	Control Unit	Data path	LOAD_AUX_MULT7
Load signal	Control Unit	Data path	LOAD_AUX_ADDER9
Load signal	Control Unit	Data path	LOAD_AUX_ADDER10
Load signal	Control Unit	Data path	LOAD_AUX_SUB3
Load signal	Control Unit	Data path	LOAD_INV_COV
Clear signal	Control Unit	Data path	CLEAR_COVXXs
Clear signal	Control Unit	Data path	CLEAR_COVYYs
Clear signal	Control Unit	Data path	CLEAR_COVZZs

Clear signal	Control Unit	Data path	CLEAR_AUX_MULT6
Clear signal	Control Unit	Data path	CLEAR_AUX_MULT7
Clear signal	Control Unit	Data path	CLEAR_AUX_ADDER9
Clear signal	Control Unit	Data path	CLEAR_AUX_ADDER10
Clear signal	Control Unit	Data path	CLEAR_AUX_SUB3
Clear signal	Control Unit	Data path	CLEAR_INV_COV
Enable for counters	Control Unit	Data path	INC_OP1
Enable for counters	Control Unit	Data path	INC_OP2
Enable for counters	Control Unit	Data path	INC_OP3
Clear signal	Control Unit	Data path	CLEAR_OP1
Clear signal	Control Unit	Data path	CLEAR_OP2
Clear signal	Control Unit	Data path	CLEAR_OP3
Enable for math blocks	Control Unit	Data path	EN_MULT6
Enable for math blocks	Control Unit	Data path	EN_MULT7
Enable for math blocks	Control Unit	Data path	EN_ADDER9
Enable for math blocks	Control Unit	Data path	EN_ADDER10
Enable for math blocks	Control Unit	Data path	EN_SUB3
Enable for math blocks	Control Unit	Data path	EN_DIV4
Results of comparisons	Data path	Control Unit	OP1_EQS_7
Results of comparisons	Data path	Control Unit	OP2_EQS_9
Results of comparisons	Data path	Control Unit	OP2_EQS_10

Results of comparisons	Data path	Control Unit	OP3_EQS_16

Table 10 List of signals exchanged between control unit and data path

These components and the corresponding signals implement the computations explained in Appendix A. The multiplications involved in the computation of the determinant and the multiplications involved in the inversion of the matrix have been computed in parallel as much as possible, in such a way to speed up the computing time. For this reason, the command signals are in common for the blocks that perform the same kind of operation in parallel.

Computation of the smallest eigenvector

The remaining part is the computation of the eigenvector associated to the smallest eigenvector by means of the inverse power method. The components which are necessary to perform the computations are:

TYPE OF COMPONENT	NAME	ALREADY INSTANTIATED
RegN (32-bits)	EV0_1	No
RegN (32-bits)	EV0_2	No
RegN (32-bits)	EV0_3	No
RegN (32-bits)	NORM_EV0	No
RegN (32-bits)	EV_1	No
RegN (32-bits)	EV_2	No
RegN (32-bits)	EV_3	No
RegN (32-bits)	INV_COVXX	Yes
RegN (32-bits)	INV_COVYY	Yes
RegN (32-bits)	INV_COVZZ	Yes
RegN (32-bits)	INV_COVXY	Yes
RegN (32-bits)	INV_COVXZ	Yes
RegN (32-bits)	INV_COVYZ	Yes
RegN (32-bits)	A	No

RegN (32-bits)	B	No
RegN (32-bits)	C	No
RegN (32-bits)	D	No
RegN (32-bits)	E	No
RegN (32-bits)	F	No
RegN (32-bits)	G	No
RegN (32-bits)	H	No
RegN (32-bits)	J	No
RegN (32-bits)	AB	No
RegN (32-bits)	DE	No
RegN (32-bits)	GH	No
RegN (32-bits)	NEW_EV0_1	No
RegN (32-bits)	NEW_EV0_2	No
RegN (32-bits)	NEW_EV0_3	No
RegN (32-bits)	THETA1	No
RegN (32-bits)	THETA2	No
RegN (32-bits)	THETA1	No
RegN (32-bits)	THETA_NF	No
RegN (32-bits)	THETA	No
RegN (32-bits)	PART_VEC_CHECK1	No
RegN (32-bits)	PART_VEC_CHECK2	No
RegN (32-bits)	PART_VEC_CHECK3	No
RegN (32-bits)	VEC_CHECK1	No
RegN (32-bits)	VEC_CHECK2	No
RegN (32-bits)	VEC_CHECK3	No
RegN (32-bits)	NORM_VEC_CHECK	No
RegN (32-bits)	ABS_THETA	No
RegN (32-bits)	THETAxDELTA	No
Multiplier (IP Block)	MULTa	No
Multiplier (IP Block)	MULTb	No
Multiplier (IP Block)	MULTc	No
Multiplier (IP Block)	MULTd	No
Multiplier (IP Block)	MULTe	No

Multiplier (IP Block)	MULTf	No
Multiplier (IP Block)	MULTg	No
Multiplier (IP Block)	MULTh	No
Multiplier (IP Block)	MULTj	No
Adder (IP Block)	ADDab	No
Adder (IP Block)	ADDde	No
Adder (IP Block)	ADDgh	No
Multiplier (IP Block)	MULTtheta1	No
Multiplier (IP Block)	MULTtheta2	No
Multiplier (IP Block)	MULTtheta3	No
Adder (IP Block)	ADDtheta_nf	No
Adder (IP Block)	ADDtheta	No
Multiplier (IP Block)	MULTpartcheck	No
Multiplier (IP Block)	MULTpartcheck	No
Multiplier (IP Block)	MULTpartcheck	No
Subtractor (IP Block)	SUBcheck1	No
Subtractor (IP Block)	SUBcheck2	No
Subtractor (IP Block)	SUBcheck3	No
Absolute_value (IP Block)	ABStheta	No
Multiplier (IP Block)	MULTfinal_thr	No
Comparator (IP Block)	EV_CMP_THR	No
Counter8	N_CNT	No
Counter5	OP3_CNT	Yes
Counter4	OP2_CNT	Yes
Counter3	OP1_CNT	Yes
CMP_16	OP3_CMP_16	Yes
CMP_9	OP2_CMP_9	Yes
CMP_7	OP1_CMP_7	Yes
CMP_10	OP2_CMP_10	Yes
CMP_2	OP1_CMP_2	No
CMP_255	N_CMP_255	No
MUX_2TO1	MUX_EV0_1	No
MUX_2TO1	MUX_EV0_2	No
MUX_2TO1	MUX_EV0_3	No

Table 11 List of components in data path

TYPE OF SIGNAL	FROM	TO	NAME
Load signal	Control Unit	Data path	LOAD_EV0_1
Load signal	Control Unit	Data path	LOAD_EV0_2
Load signal	Control Unit	Data path	LOAD_EV0_3
Load signal	Control Unit	Data path	LOAD_NORM_EV0
Load signal	Control Unit	Data path	LOAD_EV_1
Load signal	Control Unit	Data path	LOAD_EV_2
Load signal	Control Unit	Data path	LOAD_EV_3
Load signal	Control Unit	Data path	LOAD_A
Load signal	Control Unit	Data path	LOAD_B
Load signal	Control Unit	Data path	LOAD_C
Load signal	Control Unit	Data path	LOAD_D
Load signal	Control Unit	Data path	LOAD_E
Load signal	Control Unit	Data path	LOAD_F
Load signal	Control Unit	Data path	LOAD_G
Load signal	Control Unit	Data path	LOAD_H
Load signal	Control Unit	Data path	LOAD_J
Load signal	Control Unit	Data path	LOAD_AB
Load signal	Control Unit	Data path	LOAD_DE
Load signal	Control Unit	Data path	LOAD_GH
Load signal	Control Unit	Data path	LOAD_NEW_EV0_1
Load signal	Control Unit	Data path	LOAD_NEW_EV0_2
Load signal	Control Unit	Data path	LOAD_NEW_EV0_3
Load signal	Control Unit	Data path	LOAD_THETA1
Load signal	Control Unit	Data path	LOAD_THETA2
Load signal	Control Unit	Data path	LOAD_THETA3
Load signal	Control Unit	Data path	LOAD_PART_VEC_CHECK1
Load signal	Control Unit	Data path	LOAD_PART_VEC_CHECK2
Load signal	Control Unit	Data path	LOAD_PART_VEC_CHECK3
Load signal	Control Unit	Data path	LOAD_VEC_CHECK1

Load signal	Control Unit	Data path	LOAD_VEC_CHECK2
Load signal	Control Unit	Data path	LOAD_VEC_CHECK3
Load signal	Control Unit	Data path	LOAD_NORM_VEC_CHECK
Load signal	Control Unit	Data path	LOAD_ABS_THETA
Load signal	Control Unit	Data path	LOAD_FINAL_THR
Clear signal	Control Unit	Data path	CLEAR_EV0_1
Clear signal	Control Unit	Data path	CLEAR_EV0_2
Clear signal	Control Unit	Data path	CLEAR_EV0_3
Clear signal	Control Unit	Data path	CLEAR_NORM_EV0
Clear signal	Control Unit	Data path	CLEAR_EV_1
Clear signal	Control Unit	Data path	CLEAR_EV_2
Clear signal	Control Unit	Data path	CLEAR_EV_3
Clear signal	Control Unit	Data path	CLEAR_A
Clear signal	Control Unit	Data path	CLEAR_B
Clear signal	Control Unit	Data path	CLEAR_C
Clear signal	Control Unit	Data path	CLEAR_D
Clear signal	Control Unit	Data path	CLEAR_E
Clear signal	Control Unit	Data path	CLEAR_F
Clear signal	Control Unit	Data path	CLEAR_G
Clear signal	Control Unit	Data path	CLEAR_H
Clear signal	Control Unit	Data path	CLEAR_J
Clear signal	Control Unit	Data path	CLEAR_AB
Clear signal	Control Unit	Data path	CLEAR_DE
Clear signal	Control Unit	Data path	CLEAR_GH
Clear signal	Control Unit	Data path	CLEAR_NEW_EV0_1
Clear signal	Control Unit	Data path	CLEAR_NEW_EV0_2
Clear signal	Control Unit	Data path	CLEAR_NEW_EV0_3
Clear signal	Control Unit	Data path	CLEAR_THETA1
Clear signal	Control Unit	Data path	CLEAR_THETA2
Clear signal	Control Unit	Data path	CLEAR_THETA3
Clear signal	Control Unit	Data path	CLEAR_PART_VEC_CHECK1
Clear signal	Control Unit	Data path	CLEAR_PART_VEC_CHECK2
Clear signal	Control Unit	Data path	CLEAR_PART_VEC_CHECK3
Clear signal	Control Unit	Data path	CLEAR_VEC_CHECK1

Clear signal	Control Unit	Data path	CLEAR_VEC_CHECK2
Clear signal	Control Unit	Data path	CLEAR_VEC_CHECK3
Clear signal	Control Unit	Data path	CLEAR_NORM_VEC_CHECK
Clear signal	Control Unit	Data path	CLEAR_ABS_THETA
Clear signal	Control Unit	Data path	CLEAR_FINAL_THR
Enable for counters	Control Unit	Data path	INC_N
Enable for counters	Control Unit	Data path	INC_OP3
Enable for counters	Control Unit	Data path	INC_OP2
Enable for counters	Control Unit	Data path	INC_OP1
Clear signal	Control Unit	Data path	CLEAR_N
Clear signal	Control Unit	Data path	CLEAR_OP3
Clear signal	Control Unit	Data path	CLEAR_OP2
Clear signal	Control Unit	Data path	CLEAR_OP1
Enable for math blocks	Control Unit	Data path	EN_MULT4
Enable for math blocks	Control Unit	Data path	EN_ADDER5
Enable for math blocks	Control Unit	Data path	EN_ADDER6
Enable for math blocks	Control Unit	Data path	EN_MULT5
Enable for math blocks	Control Unit	Data path	EN_ADDER7
Enable for math blocks	Control Unit	Data path	EN_ADDER8
Enable for math blocks	Control Unit	Data path	EN_MULT6
Enable for math blocks	Control Unit	Data path	EN_SUB4
Enable for math blocks	Control Unit	Data path	EN_ABS1
Enable for math blocks	Control Unit	Data path	EN_MULT7
Enable for math blocks	Control Unit	Data path	EN_CMP2

Selection signal	Control Unit	Data path	SEL_EV0
Results of comparisons	Data path	Control Unit	OP3_EQS_16
Results of comparisons	Data path	Control Unit	OP2_EQS_9
Results of comparisons	Data path	Control Unit	OP1_EQS_7
Results of comparisons	Data path	Control Unit	OP2_EQS_10
Results of comparisons	Data path	Control Unit	OP1_EQS_2
Results of comparisons	Control Unit	Data path	NORM_VEC_CHECK_LT_THR
Results of comparisons	Data path	Control Unit	N_EQS_255

Table 12 List of signals exchanged between control unit and data path

At this point, the iterative procedure to compute the eigenvector associated to the smallest eigenvalue can start. If it is the first iteration, the selection signal sel_EV0, which commands multiplexers MUX_EV0_1, MUX_EV0_2, MUX_EV0_3, is equal to '0' and the three components of registers EV0_1, EV0_2 and EV0_3 are set equals to one. If it is not the first iteration, the last estimate of the eigenvector associated to the smallest eigenvalue is stored on registers EV0_1, EV0_2 and EV0_3. The first operation of the algorithm is the normalization of vector EV0, so it is computed its norm by means of the block NORM_COMPUTATION. The end of the norm computation is indicated by NORM_COMPUTATION's output signal, called DONE, which becomes equal to '1'. The three components EV0_1, EV0_2 and EV0_3 can be divided by the norm, stored on register NORM_EV0. Since the division needs for 16 clock cycles to be completed, the OP3_CNT counts up to sixteen. Therefore, when this value is reached, the control unit checks the status to OP3_EQS_16 to be equal to '1'. In this way, it set equals to '0' the enable for the divider and the results of the operations are stored in the three registers EV_1, EV_2, EV_3. At this point, the product between the inverse – shifted covariance matrix and the vector EV0 should be computed, so the following operations have to be performed:

$$EV_0 = \hat{\Sigma}_{i,INV} \cdot EV = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12} & A_{22} & A_{23} \\ A_{13} & A_{23} & A_{33} \end{pmatrix} \begin{pmatrix} EV_1 \\ EV_2 \\ EV_3 \end{pmatrix}$$

$$EV_{0,1} = A_{11}EV_1 + A_{12}EV_2 + A_{13}EV_3$$

$$EV_{0,2} = A_{12}EV_1 + A_{22}EV_2 + A_{23}EV_3$$

$$EV_{0,3} = A_{13}EV_1 + A_{23}EV_2 + A_{33}EV_3$$

The coefficients of the inverse shifted-covariance matrix, A_{11}, A_{22}, A_{33}, A_{12}, A_{13}, A_{23} are respectively stored on registers INV_COVXX, INV_COVYY, INV_COVZZ, INV_COVXY, INV_COVXZ, INV_COVYZ. Before, the multiplications between the coefficients and the elements of vector EV are performed and, to simplify the notation, the results of these operations are stored on register A, B, C, D, E, F, G, H, J. These nine multiplications are performed in parallel in such a way to speed up the estimation of the normal vector. However, each multiplication requires 9 clock cycles to be completed so the OP2_CNT counts up to 9 and, when this value is reached, the comparator OP2_CMP_9 set the signal OP2_EQS_9 equals '1'. In this way, the control unit understands that the operation has been carried out and it activates the load signals of the corresponding registers to save the results of the operations. Then, it has to be performed the sum between A, B, C and, at the same way the sum between D, E, F and the sum between G, H, J. To complete this task, it has to be first performed the sum between A and B, D and E, G and H and the results are respectively stored on registers AB, DE, GH. Then, the values stored on these registers are added to those stored on C, F, J. In both steps, the three adding operations are carried out in parallel but the OP1_CNT is exploited to count the 7 clock cycles needed to complete the sum. The results of the last sums are stored on registers NEW_EV0_1, NEW_EV0_2, NEW_EV0_3. At this point, θ has to be estimated, which is given by the product between the new estimate EV_0, whose components are stored on NEW_EV0_1, NEW_EV0_2, NEW_EV0_3, and vector EV, whose components are stored on EV_1, EV_2, EV_3. The following operations have to be performed:

$$\theta = EV_0 \cdot EV^T = \begin{pmatrix} EV_{0,1} \\ EV_{0,2} \\ EV_{0,3} \end{pmatrix} (EV_1 \quad EV_2 \quad EV_3)$$
$$= EV_{0,1}EV_1 + EV_{0,2}EV_2 + EV_{0,3}EV_3$$

The first step consists of computing the products between NEW_EV0_1 and EV_1, NEW_EV0_2 and EV_2, NEW_EV0_3 and EV_3, which are respectively stored on the new registers called THETA_1, THETA_2, THETA_3. Then, the value stored on registers THETA_1 and THETA_2 are added together and the result is stored on registers THETA_NF. Finally, the result on THETA_NF is added to the value stored on THETA_3. The output of these two sum operations is the final result value of θ and it is saved on the register called THETA. The next action consists of verifying the difference between the previous estimate and the new one. The check is performed on vector $EV_0 - \theta EV$, so the first action to take is to compute the three products, θEV_1, θEV_2 and θEV_3. Then, these three products have to be subtracted to the values stored on NEW_EV0_1, NEW_EV0_2, NEW_EV0_3. The results of these subtractions are stored on registers VEC_CHECK1, VEC_CHECK2, VEC_CHECK3, which are fed as input to the block NORM_COMPUTATION. When the block completes the computation of the norm, the output signal DONE becomes equal to '1' so it is performed the product between the absolute value of THETA and the threshold δ, whose value is stored on CONSTANT_5. The absolute value of THETA is computed by means of the IP Block Absolute_value, which needs only one clock cycle to perform the computation. The comparison between this product, stored on register FINAL_THR, and the norm of vector VEC_CHECK, which is stored on the register called NORM_VEC_CHECK, is performed by the block EV_CMP_THR. The result of the comparison is analyzed by the control unit: if it is equal to '1', it means that the condition $||EV_0 - \theta EV||_2 \leq |\theta|\delta$ is verified, so the algorithm can stop; otherwise, through the initial mux, the values NEW_EV0_1, NEW_EV0_2, NEW_EV0_3 are stored in EV0_1, EV0_2 and EV0_3 and the steps are performed again. There is a counter, called N_CNT, that counts how many times the iterations are repeated and, when it reaches the value 255, even if the condition on the norm is not respected, the last computed values for EV0 are saved. If the algorithm finishes the computation of the three components of EV0, the division by THETA can be performed and the final results are stored on final registers NX, NY, NZ. At this point, the data path returns the three components of the normal vector as output to the common interface between data path and control unit.

4.3.3.5 NORM_COMPUTATION

During the explanation of the data path structure, it has been mentioned the component NORM_COMPUTATION. It is a fully – personalized basic component, which has been implement with the same architecture that has be exploited in the design of the normal estimation hardware unit. It means there is a data path, where the data are processed and a control unit, which enable the registers and mathematical blocks in the data path, in such a way to control the flow of data into it. Moreover, there is a common interface that links the two computational blocks. First, the list of the components involved in the data path are listed below:

TYPE OF COMPONENT	NAME	ALREADY INSTANTIATED
RegN (32-bits)	X	No
RegN (32-bits)	Y	No
RegN (32-bits)	Z	No
RegN (32-bits)	X2	No
RegN (32-bits)	Y2	No
RegN (32-bits)	Z2	No
RegN (32-bits)	SUMxy	No
RegN (32-bits)	SUMxyz	No
RegN (32-bits)	NORM	No
Multiplier (IP Block)	MULT1	No
Multiplier (IP Block)	MULT2	No
Multiplier (IP Block)	MULT3	No
Adder (IP Block)	ADDER1	No
Adder (IP Block)	ADDER2	No
Square root (IP Block)	SQR_ROOT1	No
Counter4	OP2_CNT	No
Counter3	OP1_CNT	No
CMP9	OP2_CMP_9	No
CMP7	OP1_CMP_7	No

Table 13 List of components in data path

The signals exchanged between the data path and the control unit are listed below:

TYPE OF SIGNAL	FROM	TO	NAME
Load signal	Control Unit	Data path	LOAD_X
Load signal	Control Unit	Data path	LOAD_Y
Load signal	Control Unit	Data path	LOAD_Z
Load signal	Control Unit	Data path	LOAD_X2
Load signal	Control Unit	Data path	LOAD_Y2
Load signal	Control Unit	Data path	LOAD_Z2
Load signal	Control Unit	Data path	LOAD_SUMxy
Load signal	Control Unit	Data path	LOAD_SUMxyz
Load signal	Control Unit	Data path	LOAD_NORM
Clear signal	Control Unit	Data path	CLEAR_X
Clear signal	Control Unit	Data path	CLEAR_Y
Clear signal	Control Unit	Data path	CLEAR_Z
Clear signal	Control Unit	Data path	CLEAR_X2
Clear signal	Control Unit	Data path	CLEAR_Y2
Clear signal	Control Unit	Data path	CLEAR_Z2
Clear signal	Control Unit	Data path	CLEAR_SUMxy
Clear signal	Control Unit	Data path	CLEAR_SUMxyz
Clear signal	Control Unit	Data path	CLEAR_NORM
Enable for counters	Control Unit	Data path	INC_OP1
Enable for counters	Control Unit	Data path	INC_OP2
Clear signal	Control Unit	Data path	CLEAR_OP1
Clear signal	Control Unit	Data path	CLEAR_OP2
Enable for math blocks	Control Unit	Data path	EN_MULT1
Enable for math blocks	Control Unit	Data path	EN_ADDER1
Enable for math blocks	Control Unit	Data path	EN_ADDER2
Enable for math blocks	Control Unit	Data path	EN_SQR_ROOT1
Results of comparisons	Data path	Control Unit	OP1_EQS_7
Results of comparisons	Data path	Control Unit	OP2_EQS_9

Table 14 List of signals exchanged between control unit and data path

First, there are two key aspects to highlight:

- The multiplier blocks, provided by Intel®, have a latency of nine clock cycles. It means that the final result is available only after nine clock cycles. For this reason, the counter called OP2_CNT is instantiated: indeed, it counts up to nine and, when this value is attained, the comparator OP2_CMP_9 returns the output signal OP2_EQS_9 equals '1' to the control unit. In this way, the control unit turns off the enable signals for the multiplier block. The same considerations can be made for component SQR_ROOT, instantiation of the IP block square_root, which needs for nine clock cycles to produce the final result.
- The adder blocks have a latency of seven clock cycles. It means that the final result is available only after seven clock cycles. For this reason, the counter called OP1_CNT is instantiated: indeed, it counts up to seven and, when this value is attained, the comparator OP1_CMP_7 returns the output signal OP1_EQS_7 equals '1' to the control unit. In this way, the control unit turns off the enable signals for the adder block.

The norm computation consists of computing the Euclidean norm of a vector $v = [v_1\ v_2\ v_3]^T$. In this case, it is considered a vector having three components, so the steps to perform are:

- To square the single components of vector v, in such a way to obtain v_1^2, v_2^2, v_3^2;
- To sum these three squared components, in such a way to have $v_1^2 + v_2^2 + v_3^2$;
- To perform the square root of the last computed sum. The final norm is given by:

$$\|v\|_2 = \sqrt[2]{v_1^2 + v_2^2 + v_3^2}$$

The state S0 of control unit is entirely dedicated to set the load signals of all registers and counter equal to '0', the clear signals of all registers and counters equal to '1' and the enable signals of all arithmetic blocks are set equal to '0'. After this first setup state, the first operation to be performed is the square of the input values, which represent the components of the vector whose norm has to be computed. The input components are stored on registers X, Y, Z. The square operation is carried out by multipliers MULT1, MULT2, MULT3, one for each component. When the operation is completed, the result is stored on registers X2, Y2, Z2. Then, these three squared values shall be added together, so it is first performed the sum between the values stored on X2 and Y2 through component ADDER1 and the result of this sum is stored on register SUMxy. Successively, the sum between the content of register SUMxy and the quantity stored on Z2 are added together through component ADDER2 and the result is stored on register SUMxyz. The last step to perform is the square root of the sum of the squared components, so the value stored on SUMxyz is fed as input to the block SQR_ROOT1. Finally, the result produced by this block is stored on register NORM, which represents the output of the whole NORM_COMPUTATION block. Moreover, to indicate that the whole activity of performing the norm computation is completed, the common interface between the data path and the control produces an output signal, called DONE, which becomes high to indicate the termination of operations.

Chapter 5

Conclusions and future works

5.1 Results of the synthesis process

The module was synthesized using Intel's Quartus Prime suite. The available synthesis tool generates a complete report after the compilation, to inform the user about the hardware resources needed to implement the desired functionalities. The hardware resources necessary to implement the present algorithm are listed in the table below.

Total logic elements	39513/55856 (71%)
Total registers	30330
Total pins	100/322 (31%)
Total virtual pins	0
Total memory bits	405152/2396160 (17%)
Embedded Multiplier 9-bit elements	260/312 (83%)
Device	Cyclone 10 LP 10CL055YF484C6G

Table 15 Allocated hardware resources for the synthesis

As it is possible to observe, the number of registers is considerable, which makes sense if it is considered the fact that the design has been carried out by means of the register transfer level technique, so by means of a technique that makes use of registers to store the computed values at the end of each operation.

5.2 Numerical simulations

After the synthesis, the simulation was performed by means of the ModelSim HDL Simulator. For this purpose, a simple testbench was developed in VHDL. Since the designed circuit implements the hardware unit computing the normal vector for a single point, the data points (so point p_i and its k – neighbors) for each simulation are written inside the components ROMx, ROMy, ROMz instantiated in the data path.

5.2.1 First test – Clear blue region

This region is flat and the slope is everywhere the same. In particular, the coordinate z is equal to the highest value it could attain, that is $z_{HI} = 0.5\, mt$. For example, a point of this region and its neighbors are defined as follow:

$x = 0$ $y = 0$ $z = 0.5$	$x = 0$ $y = 0.2083$ $z = 0.5$	$x = 0$ $y = 0.4167$ $z = 0.5$
$x = 0.2083$ $y = 0$ $z = 0.5$	$x = 0.2083$ $y = 0.2083$ $z = 0.5$	$x = 0.2083$ $y = 0.4167$ $z = 0.5$
$x = 0.4167$ $y = 0$ $z = 0.5$	$x = 0.4167$ $y = 0.2083$ $z = 0.5$	$x = 0.4167$ $y = 0.4167$ $z = 0.5$

Table 16 Data input for region 1

In this case, the following results are obtained:

$$N_X = -5.7253 \cdot 10^{-8}$$

$$N_Y = -5.7253 \cdot 10^{-8}$$

$$N_Z = 0.9994$$

5.2.2 Second test – Dark green region

This region is flat and the slope is everywhere the same. In particular, the coordinate z is equal to the lowest value it could attain, that is $z_{LO} = 0\, mt$. For example, a point of this region and its neighbors are defined as follow:

$x = 3.9583$ $y = 2.2917$ $z = 0.5$	$x = 3.9583$ $y = 2.5$ $z = 0.5$	$x = 3.9583$ $y = 2.7083$ $z = 0.5$

$x = 4.1667$ $y = 2.2917$ $z = 0.5$	$x = 4.1667$ $y = 2.5$ $z = 0.5$	$x = 4.1667$ $y = 2.7083$ $z = 0.5$
$x = 4.3750$ $y = 2.2917$ $z = 0.5$	$x = 4.3750$ $y = 2.5$ $z = 0.5$	$x = 4.3750$ $y = 2.7083$ $z = 0.5$

Table 17 Data input for region 2

In this case, the following results are obtained:

$$N_X = -5.7253 \cdot 10^{-8}$$

$$N_Y = -5.7336 \cdot 10^{-8}$$

$$N_Z = 0.9987$$

5.2.3 Third test – Clear green region

Also this region is flat but the coordinate z takes on the values between z_{HI} and a middle value, $z_{MID} = 0.25\,mt$. The set of values between z_{HI} and z_{MID} are obtained by uniformly sampling the interval between the two extremes. For example, a point of this region and its neighbors are defined as follow:

$x = 3.5417$ $y = 0$ $z = 0.4318$	$x = 3.5417$ $y = 0.2083$ $z = 0.4318$	$x = 3.5417$ $y = 0.4167$ $z = 0.4318$
$x = 3.75$ $y = 0$ $z = 0.4091$	$x = 3.75$ $y = 0.2083$ $z = 0.4091$	$x = 3.75$ $y = 0.4167$ $z = 0.4091$
$x = 3.958$ $y = 0$ $z = 0.3864$	$x = 3.958$ $y = 0.2083$ $z = 0.3864$	$x = 3.958$ $y = 0.4167$ $z = 0.3864$

Table 18 Data input for region 3

In this case, the following results are obtained:

$$N_X = -0.1084$$

$$N_Y = 0$$

$$N_Z = -0.9941$$

5.2.4 Fourth test – Purple region

Also this region is flat but the coordinate z takes on the values between z_{LO} and a middle value, $z_{MID} = 0.25\ mt$. The set of values between z_{LO} and z_{MID} are obtained by uniformly sampling the interval between the two extremes. For example, a point of this region and its neighbors are defined as follow:

$x = 3.5417$ $y = 3.9583$ $z = 0.1364$	$x = 3.5417$ $y = 4.1667$ $z = 0.1364$	$x = 3.5417$ $y = 4.3750$ $z = 0.1364$
$x = 3.75$ $y = 3.9583$ $z = 0.1591$	$x = 3.75$ $y = 4.1667$ $z = 0.1591$	$x = 3.75$ $y = 4.3750$ $z = 0.1591$
$x = 3.958$ $y = 3.9583$ $z = 0.1818$	$x = 3.958$ $y = 4.1667$ $z = 0.1818$	$x = 3.958$ $y = 4.3750$ $z = 0.1818$

Table 19 Data input for region 4

In this case, the following results are obtained:

$$N_X = -0.1084$$

$$N_Y = 0$$

$$N_Z = 0.9941$$

5.2.5 Analysis of the results

The obtained results confirm the correctness of the algorithm as well as the fact that the designed circuit effectively implements the desired functions. Obviously, it has been reported only one point for each region, to give an idea of the results in the different areas. For what regards the results, they reflect the correct orientation of the normal: the fact that the rounding settings have been set in the IP blocks as well as the fact that the involved matrix is not the original covariance one but the shift let the result be slightly different from the expected values. However the difference is quite small, so these results can be perfectly accepted. Another consideration to point out is the fact that there is a irregularity in signs: indeed, for the last region, the signs reflect the correct orientation of the normal vector, while it is not true for the third region. It means that, in a future work, it is crucial to orient all the normals according to the viewpoint. Finally, as expected, no result was produced for the points on the corner.

5.3 Timing results

From a point of view of timing, the resulting module has a high latency of more than 925 clocks latency (depending on the number of needed iterations N). However, there is the chance of improving the throughput of the designed architecture: indeed, the circuit needs for 104 clock cycles to return the first result about the centroid computation. It means that a good pipelining realization of the circuit could allow for a throughput of 104 clock cycles to be achieved. Moreover, the clock used in the simulation has 5 ns period or 200MHz frequency. Considering the latency of the module, this means that it would take about 5 μs until the first output is computed but exploiting the pipelining implementation the throughput could consistently decrease.

5.4 Future works

The aim of the thesis was to demonstrate the feasibility of implementing a well – know computer vision algorithm on a FPGA board. It has been shown how to correct minor problems raised during the definition of the algorithm and the corresponding hardware implementation for each step of the algorithm. Future works shall concentrate on optimizing the designed circuit, by decreasing the number of allocated resources and shared signals. At the same way, the design should be pipelined in such a way to increase the throughput. Finally, it could be extended the use of the hardware unit, in such a way to compute the normal vector for all points in the point cloud.

Appendix A

Mathematical computations behind data path realization of eigenvector estimation

Given a generic matrix $A \in \mathbb{R}^{3,3}$, whose entries are the following ones:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Its inverse matrix can be computed only if $\det(A) \neq 0$, since its expression is given by:

$$A^{-1} = \frac{1}{\det(A)} \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

Where each element of position (i,j) is obtained as:

$$A_{i,j} = (-1)^{i+j} C_{ij}$$

Taking an element $a_{i,j}$ of the matrix A, C_{ij} is the complementary minor relative to the element $a_{i,j}$ and it is the determinant of the submatrix which is obtained from A by eliminating the i-th row and the j-th column. To compute the inverse matrix, it is then necessary to divide each entries for the determinant of matrix A, whose formula is:

$$\det(A) = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{31}a_{22}a_{13} - a_{32}a_{23}a_{11} - a_{33}a_{21}a_{12}$$

If matrix A is symmetric, its inverse matrix is:

$$A^{-1} = \frac{1}{\det(A)}\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12} & A_{22} & A_{23} \\ A_{13} & A_{23} & A_{33} \end{pmatrix}$$

Each element of the inverse matrix is computed as:

$$A_{11} = (-1)^{1+1}\begin{pmatrix} a_{22} & a_{23} \\ a_{23} & a_{33} \end{pmatrix} = a_{22}a_{33} - a_{23}^2$$

$$A_{22} = (-1)^{2+2}\begin{pmatrix} a_{11} & a_{13} \\ a_{13} & a_{33} \end{pmatrix} = a_{11}a_{33} - a_{13}^2$$

$$A_{33} = (-1)^{3+3}\begin{pmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{12}^2$$

$$A_{12} = A_{21} = (-1)^{1+2}\begin{pmatrix} a_{12} & a_{23} \\ a_{13} & a_{33} \end{pmatrix} = -(a_{12}a_{33} - a_{13}a_{23})$$

$$A_{13} = A_{31} = (-1)^{1+3}\begin{pmatrix} a_{12} & a_{22} \\ a_{13} & a_{23} \end{pmatrix} = a_{12}a_{23} - a_{13}a_{22}$$

$$A_{23} = A_{32} = (-1)^{2+3}\begin{pmatrix} a_{11} & a_{13} \\ a_{12} & a_{23} \end{pmatrix} = -(a_{11}a_{23} - a_{13}a_{12})$$

Finally, the simplified formula for the determinant is the following one:

$$\det(A) = a_{11}a_{22}a_{33} + 2a_{12}a_{23}a_{31} - a_{13}^2a_{22} - a_{23}^2a_{11} - a_{12}^2a_{33}$$

In the considered case, the matrix that has to be inverted is the shifted covariance matrix:

$$\hat{\Sigma} = \begin{pmatrix} COV_{xx} & COV_{xy} & COV_{xz} \\ COV_{xy} & COV_{yy} & COV_{yz} \\ COV_{xz} & COV_{yz} & COV_{zz} \end{pmatrix}$$

$$\hat{\Sigma} - \sigma I_{3,3} = \begin{pmatrix} COV_{xx} - \sigma & COV_{xy} & COV_{xz} \\ COV_{xy} & COV_{yy} - \sigma & COV_{yz} \\ COV_{xz} & COV_{yz} & COV_{zz} - \sigma \end{pmatrix}$$

The determinant of this new matrix is:

$$\begin{aligned} \det\left(\hat{\Sigma} - \sigma I_{3,3}\right) &= (COV_{xx} - \sigma)(COV_{yy} - \sigma)(COV_{zz} - \sigma) + 2COV_{xy}COV_{yz}COV_{xz} \\ &\quad - COV_{xz}^2(COV_{yy} - \sigma) - COV_{yz}^2(COV_{xx} - \sigma) - COV_{xy}^2(COV_{zz} - \sigma) \end{aligned}$$

While the elements of inverse matrix $(\hat{\Sigma} - \sigma I_{3,3})^{-1}$ are:

$$\begin{aligned} A_{11} = (-1)^{1+1} &\begin{pmatrix} COV_{yy} - \sigma & COV_{yz} \\ COV_{yz} & COV_{zz} - \sigma \end{pmatrix} \\ &= (COV_{yy} - \sigma)(COV_{zz} - \sigma) - COV_{yz}^2 \end{aligned}$$

$$\begin{aligned} A_{22} = (-1)^{2+2} &\begin{pmatrix} COV_{xx} - \sigma & COV_{xz} \\ COV_{xz} & COV_{zz} - \sigma \end{pmatrix} \\ &= (COV_{xx} - \sigma)(COV_{zz} - \sigma) - COV_{xz}^2 \end{aligned}$$

$$\begin{aligned} A_{33} = (-1)^{3+3} &\begin{pmatrix} COV_{xx} - \sigma & COV_{xy} \\ COV_{xy} & COV_{yy} - \sigma \end{pmatrix} \\ &= (COV_{xx} - \sigma)(COV_{yy} - \sigma) - COV_{xy}^2 \end{aligned}$$

$$\begin{aligned} A_{12} = A_{21} = (-1)^{1+2} &\begin{pmatrix} COV_{xy} & COV_{yz} \\ COV_{xz} & COV_{zz} - \sigma \end{pmatrix} \\ &= -[COV_{xy}(COV_{zz} - \sigma) - COV_{xz}COV_{yz}] \end{aligned}$$

$$A_{13} = A_{31} = (-1)^{1+3}\begin{pmatrix} COV_{xy} & COV_{yy} - \sigma \\ COV_{xz} & COV_{yz} \end{pmatrix}$$

$$= COV_{xy}COV_{yz} - COV_{xz}(COV_{yy} - \sigma)$$

$$A_{23} = A_{32} = (-1)^{2+3}\begin{pmatrix} COV_{xx} - \sigma & COV_{xz} \\ COV_{xy} & COV_{yz} \end{pmatrix}$$

$$= -[(COV_{xx} - \sigma)COV_{yz} - COV_{xz}COV_{xy}]$$

Bibliography and sitography

[1] Shaukat, Affan, Peter Blacker, Conrad Spiteri, e Yang Gao. «Towards Camera-LIDAR Fusion-Based Terrain Modelling for Planetary Surfaces: Review and Analysis». Sensors 16, fasc. 11 (20 novembre 2016): 1952. https://doi.org/10.3390/s16111952.

[2] Ellery, Alex. Planetary Rovers: Robotic Exploration of the Solar System. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. https://doi.org/10.1007/978-3-642-03259-2.

[3] Matthies, Larry, Mark Maimone, Andrew Johnson, Yang Cheng, Reg Willson, Carlos Villalpando, Steve Goldberg, Andres Huertas, Andrew Stein, e Anelia Angelova. «Computer Vision on Mars». International Journal of Computer Vision 75, fasc. 1 (18 luglio 2007): 67–92. https://doi.org/10.1007/s11263-007-0046-z.

[4] Berger, Matthew, Andrea Tagliasacchi, Lee M. Seversky, Pierre Alliez, Gaël Guennebaud, Joshua A. Levine, Andrei Sharf, e Claudio T. Silva. «A Survey of Surface Reconstruction from Point Clouds». Computer Graphics Forum 36, fasc. 1 (gennaio 2017): 301–29. https://doi.org/10.1111/cgf.12802.

[5] Cazals, Frédéric, e Joachim Giesen. «Delaunay Triangulation Based Surface Reconstruction». In Effective Computational Geometry for Curves and Surfaces, a cura di Jean-Daniel Boissonnat e Monique Teillaud, 231–76. Berlin, Heidelberg: Springer, 2006. https://doi.org/10.1007/978-3-540-33259-6_6.

[6] Remondino, Fabio. «From Point Cloud to Surface: The Modeling and Visualization Problem». Application/pdf, 2003, 11p. https://doi.org/10.3929/ETHZ-A-004655782.

[7] Castejón, Cristina, Beatriz L. Boada, Dolores Blanco, e Luis Moreno. «Traversable Region Modeling for Outdoor Navigation». Journal of Intelligent and Robotic Systems 43, fasc. 2–4 (24 dicembre 2005): 175–216. https://doi.org/10.1007/s10846-005-9005-5.

[8] Seraji, H. «Traversability Index: A New Concept for Planetary Rovers». In Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C), 3:2006–13. Detroit, MI, USA: IEEE, 1999. https://doi.org/10.1109/ROBOT.1999.770402.

[9] Gennery, Donald B. «Traversability Analysis and Path Planning for a Planetary Rover».

[10] Villalpando, Carlos. «Acceleration of Stereo Correlation in Verilog».

[11] Buell, D., El-Ghazawi, T., Gaj, K., Kindratenko, V.: 'High-performance reconfigurable computing', IEEE Comput., 2007, 40, (3), pp. 23–27.

[12] Herbordt, M.C., et al.: 'Achieving high performance with FPGA-based computing', IEEE Comput., 2007, 40, (3), pp. 50– 57.

[13] El-Ghazawi, T., El-Araby, E., Huang, M., Gaj, K., Kindratenko, V., Buell, D.: 'The promise of high-performance reconfigurable computing', IEEE Comput., 2008, 41, (2), pp. 69– 76.

[14] VanCourt, T., Herbordt, M.C.: 'Elements of high-performance reconfigurable computing', Adv. Comput., 2009, 75, pp. 113 –157.

[15] Underwood, K.: 'FPGAs vs. CPUs: trends in peak floating-point performance'. Proc. ACM/SIGDA 12th Int. Symp. on FieldProgrammable Gate Arrays (FPGA), Monterey, CA, USA, February 2004, pp. 171– 180.

[16] Jovanović, Ž., e V. Milutinović. «FPGA Accelerator for Floating-Point Matrix Multiplication». IET Computers & Digital Techniques 6, fasc. 4 (2012): 249. https://doi.org/10.1049/iet-cdt.2011.0132.

[17] Woods, Roger, John McAllister, Gaye Lightbody, e Ying Yi. FPGA-Based Implementation of Signal Processing Systems. Second edition. Hoboken, NJ: John Wiley & Sons Inc, 2017.

[18] Gokhale, Maya B., and Paul S. Graham. Reconfigurable computing: Accelerating computation with field-programmable gate arrays. Springer Science & Business Media, 2006.

[19] Todman, T.J., G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, e P.Y.K. Cheung. «Reconfigurable Computing: Architectures and Design Methods». IEE Proceedings - Computers and Digital Techniques 152, fasc. 2 (2005): 193. https://doi.org/10.1049/ip-cdt:20045086.

[20] Woods, Roger, John McAllister, Gaye Lightbody, e Ying Yi. FPGA-Based Implementation of Signal Processing Systems. Second edition. Hoboken, NJ: John Wiley & Sons Inc, 2017.

[21] Brown, Stephen, e Jonathan Rose. «Architecture of FPGAs and CPLDs: A Tutorial».

[22] Kuon, Ian, Russell Tessier, e Jonathan Rose. «FPGA Architecture: Survey and Challenges». Foundations and Trends® in Electronic Design Automation 2, fasc. 2 (2007): 135–253. https://doi.org/10.1561/1000000005.

[23] https://www.generatecnologias.es/en/space-fpga.html

[24] Biesiadecki, J., Maimone, M., and Morrison, J. (2001) ‘‘Athena SDM rover: A testbed

for Mars rover mobility,’’ Proceedings International Symposium Artificial Intelligence

and Robotics in Space, Montreal, Canada.

[25]https://www.esa.int/ESA_Multimedia/Images/2006/09/The_Lunar_Robotic_Mockup_LRM

[26] https://www.nasa.gov/content/what-are-smallsats-and-cubesats

[27] Varnavas, Kosta, e Dr William Herbert Sims. «The Use of Field Programmable Gate Arrays (FPGA) in Small Satellite Communication Systems».

[28] Hoppe, Hugues, Tony DeRose, Tom Duchamp, John McDonald, e Werner Stuetzle. «Surface Reconstruction from Unorganized Points».

[29] Rusu, Radu Bogdan. «Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments». KI - Künstliche Intelligenz 24, fasc. 4 (novembre 2010): 345–48. https://doi.org/10.1007/s13218-010-0059-6.

[30] Bazazian, Dena, Josep R. Casas, e Javier Ruiz-Hidalgo. «Fast and Robust Edge Extraction in Unorganized Point Clouds». In 2015 International Conference on Digital Image Computing: Techniques and Applications (DICTA), 1–8. Adelaide, Australia: IEEE, 2015. https://doi.org/10.1109/DICTA.2015.7371262.

[31] Rodriguez-Cuenca, B., S. Garcia-Cortes, C. Ordonez, e M. C. Alonso. «A Study of the Roughness and Curvature in 3D Point Clouds to Extract Vertical and Horizontal Surfaces». In 2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS), 4602–5. Milan, Italy: IEEE, 2015. https://doi.org/10.1109/IGARSS.2015.7326853.

[32] Hsieh, Cheng-Tiao. «An Efficient Development of 3D Surface Registration by Point Cloud Library (PCL)». In 2012 International Symposium on Intelligent Signal Processing and Communications Systems, 729–34. Tamsui, New Taipei City, Taiwan: IEEE, 2012. https://doi.org/10.1109/ISPACS.2012.6473587.

[33] The Elements of Statistical Learning. Data Mining, Inference and Prediction – J. Friedman.

[34] M. Gu – Single- and Multiple- Vector Iterations (Section 4.3). In Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors, Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, SIAM, Philadelphia, 2000.