Harris and FAST corner detection on the NVIDIA Jetson TX2 using OpenCV

Bas Bleijerveld

January 23, 2019

Abstract

3D modelling can be useful for a better diagnosis/prognosis in the health care. The NVIDIA Jetson TX2 is a system on a chip with intended usage for artificial intelligence and computer vision applications. The Jetson TX2 is a fairly new chip, a part of this research is exploring and explaining how to get started with computer vision applications on the NVIDIA Jetson TX2. The OpenCV library is an open source library for computer vision, image processing and machine learning. This library is used for the computer vision applications in this research. The relative orientation of images from different angles of the same object can be used to render a 3D model. Features can be detected on the images and these features can be used to determine the relative orientation. In this research two different corner detection methods are tested on their performance on the Jetson TX2, these are Harris corner detection and FAST corner detection. The illumination invariance of the methods will be compared and their performance of detecting corners from different angles.

Contents

1 Introduction

3D modelling in the health care can be useful to make a better diagnosis/prognosis. For example for the prognosis of a diabetic foot ulcer (Li et al., 2018). Diabetic foot ulcer is a disease caused by long-term increase in blood sugar. This often results in an ulcer on the foot of the patient. Before the development of digital imaging the most used method was the transparency tracing method, which is drawing the contours of the ulcer on a transparent film. With digital imaging the accuracy of the estimation of the ulcer was increased, but 2D imaging is not capable of dealing with depth changes of the ulcer. A better estimation of the ulcer can be made using a 3D modelling method, which is able to also consider the depth gradients.

With a sufficient amount of images from different angles around the patient (or a specific part of the body) it should be possible to render a 3D model. A single hand-held device for the doctor to take the images from different angles and which renders the 3D model will be practical.

The device uses computer vision, which is a discipline for processing digital images and it can be used to automate tasks and replace the need of human vision. Computer vision can be used in a lot of different working spaces. In this research the focus is the processing of medical data.

Computer vision includes methods for feature detection, which processes the images and it decides at every point in the image whether this point is a feature of a specific type or not. These features and patterns of multiple features can be used to detect the same instance in different images.

When the same instance can be detected in different images, the relative orientation of these images can be determined. This information is needed for rendering the 3D model using the images.

In this research the performance of feature detection using the OpenCV library (Bradski, 2000) on the NVIDIA Jetson TX2 is tested. The Jetson TX2 is a system on a chip (SoC) of the Tegra Series which is developed by NVIDIA. The Tegra SoCs are developed for mobile devices. They include a CPU, GPU and integrated memory. The Jetson TX2 is a SoC with intended usage for artificial intelligence and computer vision applications. So the Jetson TX2 is possibly a good SoC to use for the hand-held device to render a 3D model. For this reason the performance on the NVIDIA Jetson TX2 is tested. Harris corner detection and FAST corner detection are the two feature detection methods of which the performances on the Jetson TX2 are evaluated. The Jetson TX2 is a fairly new chip, this also means that there is not a lot of information and examples about working with the Jetson TX2 to be found on the internet. A part of this research is also exploring and explaining how to get started with computer vision applications on the NVIDIA Jetson TX2.

To make it easier for developers to get started with the Jetson TX2, NVIDIA has made a Jetson TX2 Developer Kit. The Jetson TX2 Developer Kit has the Jetson TX2 module installed on a developer board which gives it the desired hardware capabilities and an easy to use interface for the developer.

The Jetson TX2 does also come with a Software Development Kit (SDK) called JetPack which is created by NVIDIA. JetPack is an SDK for building artificial intelligence and computer vision applications. JetPack includes the operating system NVIDIA L4T, which is a Linux distribution for Tegra-devices. This Linux distribution is derived from Ubuntu. JetPack also contains libraries, APIs, developer tools, samples and documentation.

One of the libraries that is included in JetPack is OpenCV. OpenCV is an open source library for computer vision, image processing and machine learning. This is the library that will be used

for the computer vision applications in this research, with the focus on functions for Harris corner detection and FAST corner detection.

The main question of this research is which detection method of Harris and FAST has the most reliable results to work with in the health care on the Jetson TX2. This will be done by comparing the results under different conditions. One is under different lighting levels and the other one is with a rotating object. Besides that it also explains how to get started with OpenCV on the Jetson TX2.

2 Methods

2.1 Scope of methods/analysis

Harris corner detection and FAST corner detection on the Jetson TX2 have been evaluated on their performance. Two aspects of their performance have been tested. One of those was the illumination invariance of both detection methods, so how well do they perform with different levels of illumination. And the other one was the tracing of features while the object was turning to different angles. The object that was used for the experiments was a diabetic foot model which can be seen in Figure 2.1. The Jetson and the on-board camera that are used can be seen in Figure 2.2. All the materials that are used can be seen in Figure 2.3, this also includes a lamp, lux meter, rotation disc and screens to create an environment where the level of illumination can be controlled.

Figure 2.1: The diabetic foot model that is used for the experiments.

2.1.1 Illumination experiment

The illumination invariance was tested by comparing the detected features of frames taken under different lighting levels. Therefore an environment in which the lighting levels can be controlled was created. The setup with different levels of illumination can be seen in Figure 2.4 and Figure 2.5.

Both detection methods have been tested for lux values from 100 lux up to 1500 lux with steps of 100 lux. With a lux meter the amount of lux on the surface of the object has been measured, the placement of the lux meter for the measurements can be seen in Figure 2.6. To get a specific lux value the direction of the lamp could be slightly adjusted and the towel on the screens could be used as a curtain. For each lighting level a frame with the detected features has been saved.

2.1.2 Tracing / different angles experiment

The tracing of detected features from different angles of the object has been tested. A rotation disc was designed to make it possible to rotate the object in front of the camera, this rotation disc can be seen in Figure 2.7 and Figure 2.8. The object was placed on the rotation disc and

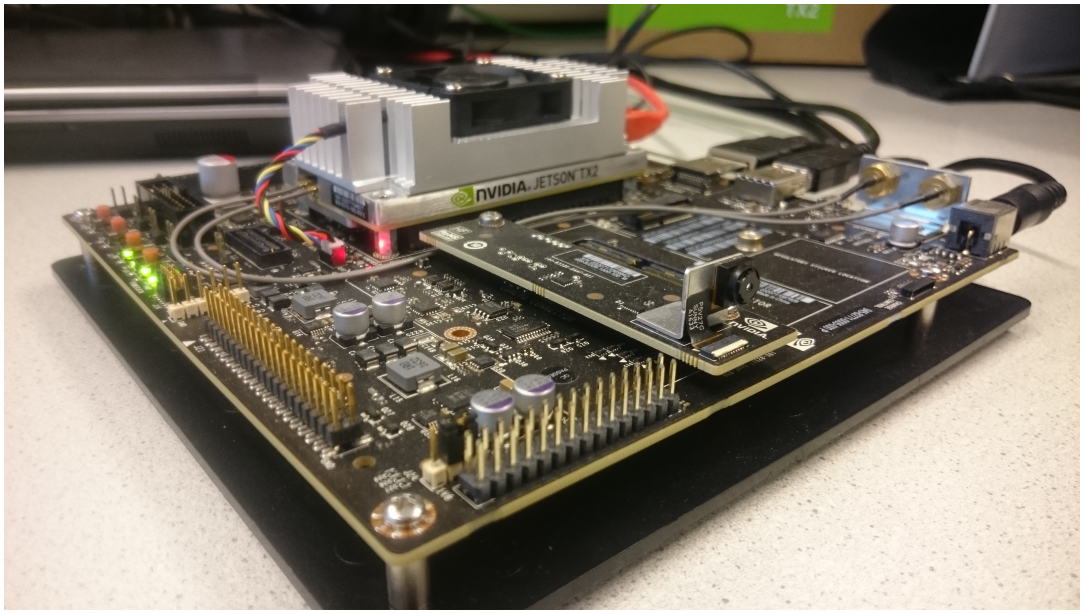

Figure 2.2: The Jetson TX2 with the on-board camera.

Figure 2.3: The materials for the experiments.

multiple frames were taken while the object was turned around. A thin thread was used to pull on the handle of the disc in order to rotate the disc without being on camera. During the rotation of the object all the frames with the detected features have been saved.

2.1.3 User case scenario

An employee of the hospital measured the lux values in the room with a cell phone to give an indication of the level of lighting. The image of the measurement can be seen in Figure 2.9. Above the treatment table the level of lighting was around 1000 Lux. This data was used for comparing the user case scenario with results of the illumination experiment.

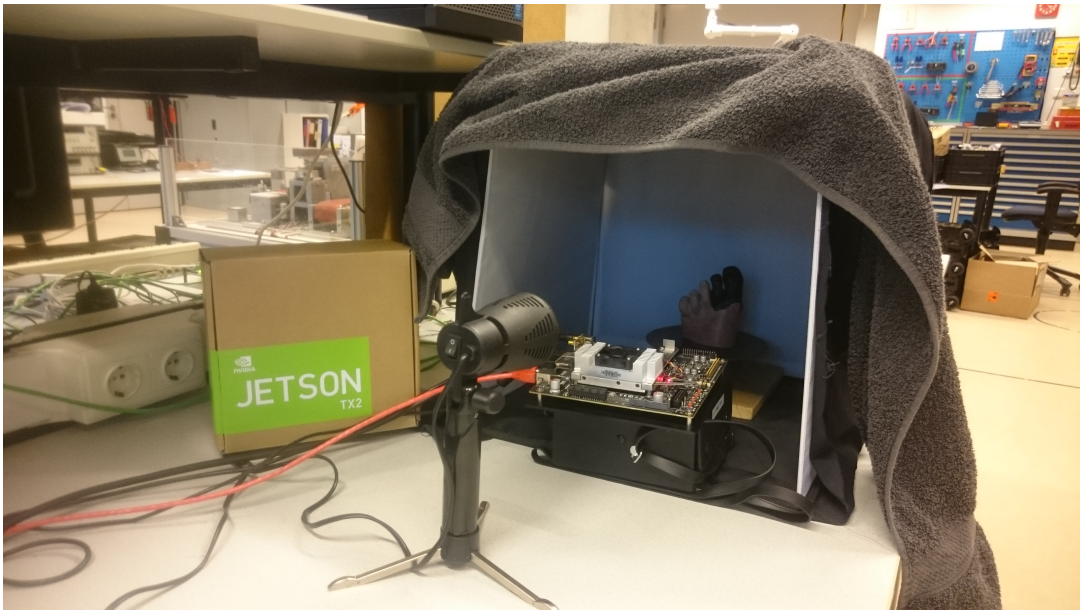

Figure 2.4: The setup for the illumination experiment with the lamp turned off.

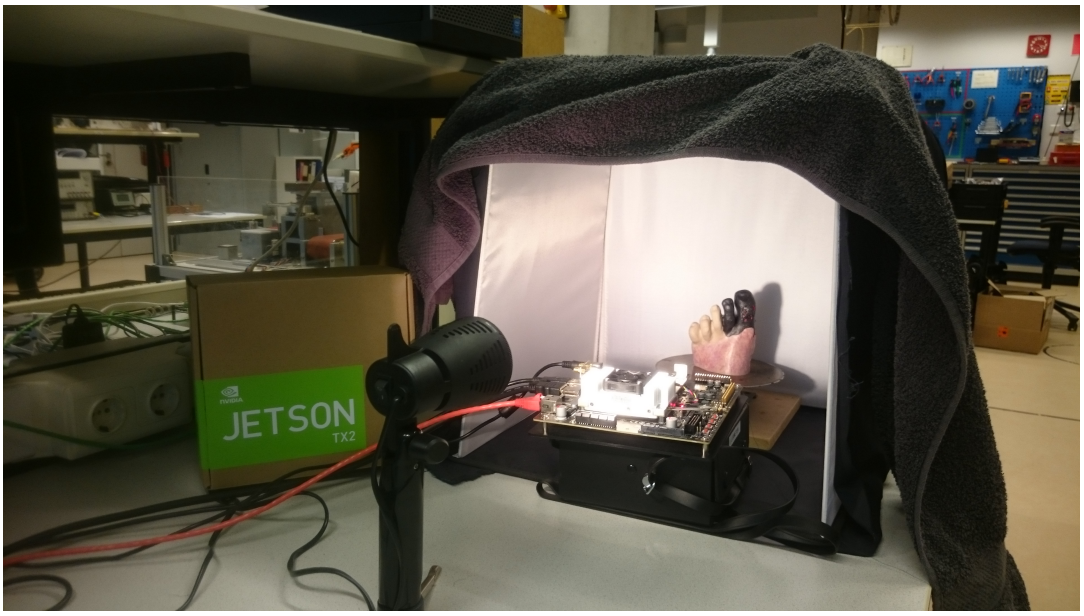

Figure 2.5: The setup for the illumination experiment with the lamp turned on.

2.2 Installation / Preparation Jetson TX2

2.2.1 Jetson install/flash Jetpack

First the Jetson TX2 needed to be prepared with JetPack.

To flash the operating system and to install JetPack 3.2.1 an extra computer is needed. This extra computer should run on Ubuntu Linux x64 Version 16.04. In the process of installing JetPack the extra computer is called the host and the Jetson is called the target. For this process it is important that the host and the target are both in the same network. This can be done by connecting them to the same router or network switch.
The JetPack installer should be downloaded first on the host. JetPack can be downloaded from

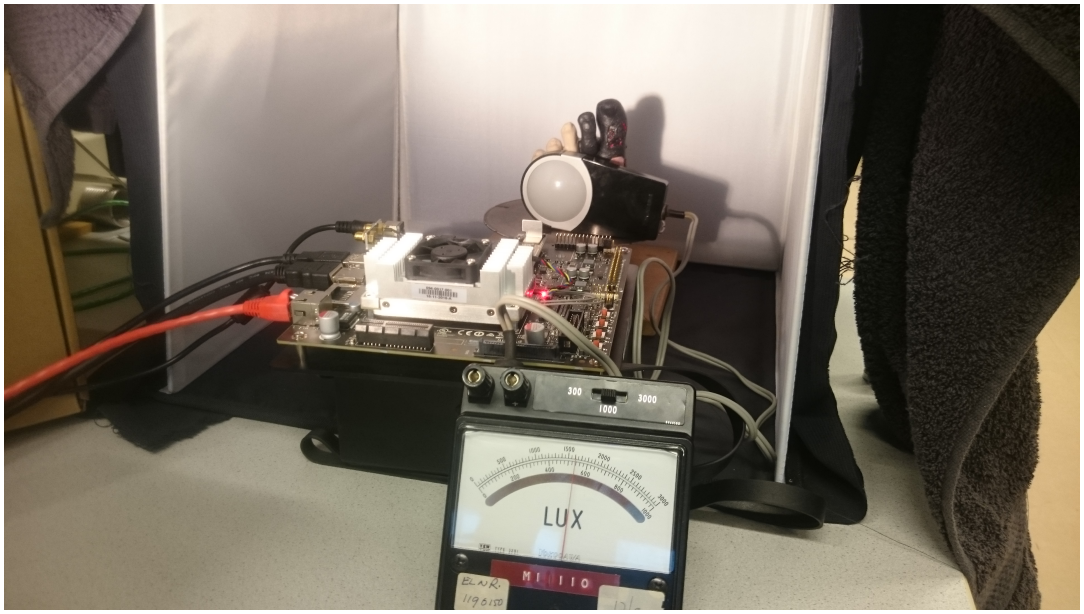

Figure 2.6: The placement of the lux meter for the measurements.

Figure 2.7: The rotation disc with diabetic foot model from the side.

the NVIDIA developer website, see the following link:
"https://developer.nvidia.com/embedded/jetpack"
Once it is downloaded the execution properties must be changed to allow execution. This can be done by going to the properties of the file, switch to the Permissions tab and check the box which allows the file to execute as a program.
To execute the installer open a terminal (CTRL + ALT + T) and switch to the directory. Switching to the directory can be done by typing in *cd* followed by the directory of the file and hit enter. In this directory the installer can be run by typing ./ followed by the filename. The installer should start and a window will pop up.
The JetPack installer will introduce what it is and indicates the directory where it will be in-

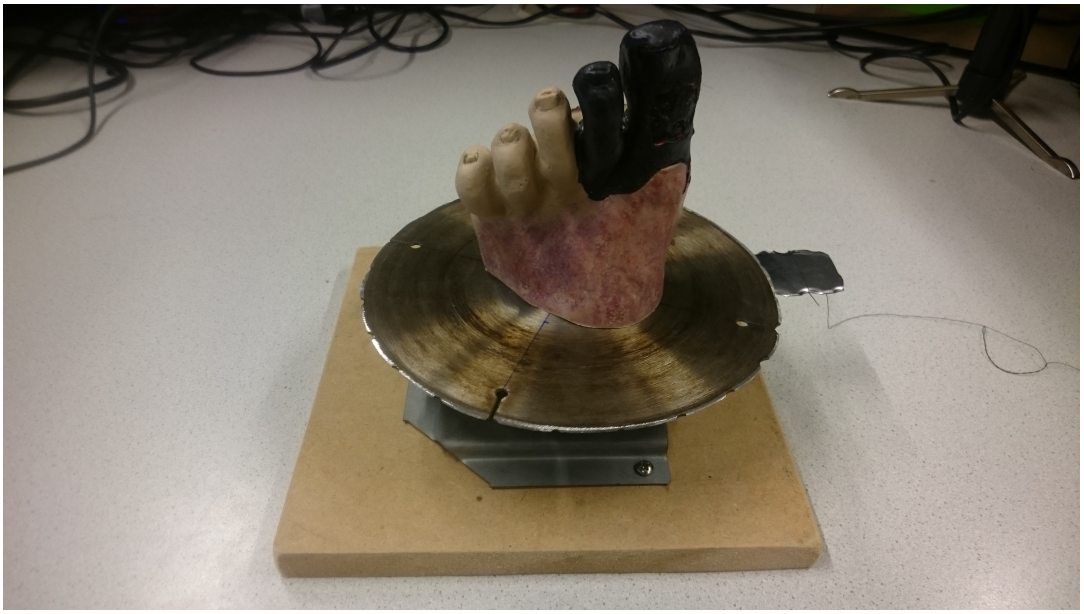

Figure 2.8: The rotation disc with diabetic foot model from above.

Figure 2.9: The lux measurement in the hospital.

stalled on the host. After that the user has to select the development environment, so which version of the Jetson is used. In the component manager can be selected which packages of JetPack must be installed. A full install can be selected in the top right corner, which will install all the packages for the host and all the packages for the target.
After the host installation is completed select the configured network layout and which interface is used on the host.
When the host is installed the installer will ask to put the target in recovery mode. If the Jetson is already running it must be shutdown first. When the Jetson is turned off disconnect the AC adapter from the board. Use the Micro USB to USB A cable to connect the Micro USB port from the Jetson with a USB port on the host. Now reconnect the power adapter to the Jetson.

Press down the POWER button. After that press and hold the FORCE RECOVERY button, while pressing the FORCE RECOVERY button press and release the RESET button and wait for two seconds to release the FORCE RECOVERY button. The Jetson should now be in recovery mode. This can be seen on the host by using the command *lsusb*, this will list the connected USB devices and the Jetson will be displayed as "NVidia Corp". Now the Jetson is in recovery mode the installation of the target can be done. Start installing JetPack on the target by pressing enter in the post installation window on the host. After installation the Jetson will boot and a Linux distribution which is derived from Ubuntu will be running with the chosen software packages installed.

2.2.2 Build OpenCV with CUDA support and GStreamer

The latest version of JetPack that was available and which has been installed for this research was JetPack 3.2.1. The Jetson and JetPack are tools in a development environment, so not everything might be completely finished and there may still be some problems. The version of OpenCV that was included in this version of JetPack did not have CUDA support to enable GPU acceleration and GStreamer was not enabled in this version. The GStreamer pipeline can be used to open the Jetson camera and enables it to actually use the camera. Both of these features were important for this research, so for that reason the OpenCV version that was included in JetPack had been removed and another version has been built. This version of OpenCV did have CUDA support and enabled GStreamer.
The OpenCV version that was built for this research was installed by using installation scripts. These scripts were made by JetsonHacks, which is a community that reports news and information around the NVIDIA Jetson and they also provide information and programs that can be used for developing on a Jetson.
To build the OpenCV version by using the JetonHacks scripts the scripts must first be downloaded. This can be done by using Git, with the command *git clone https* : *//github.com/jetsonhacks/buildOpenCVTX*2 the directory with needed files can be copied to the Jetson. Switch to the copied directory by using the *cd* command. In this directory is a file called buildOpenCV.sh, this file can be executed with the command *./buildOpenCV.sh* and it will start the build process. After this OpenCV has been built with the default settings of the scripts and this will enable CUDA support and GStreamer. OpenCV can also be build with custom settings and these settings can be changed in the /opencv/build directory by executing the command *ccmake* .. in this directory. Now all the options can be changed and a new build with other options can be generated. But for this research the default options were used.

2.3 Code of the programs

Python has been used as the programming language.

2.3.1 Open and display camera

The code in Appendix A will open the on-board camera on the Jetson TX2 developer kit and it will display the captured frames in a window.

On the first line the OpenCV library is imported. Then the function open_onboard_camera() is defined, which will open the camera on the Jetson TX2. Important to mention is that it uses a GStreamer pipeline to access the on-board camera. The pipeline accesses the camera, determines the resolution and frame rate and it converts it to the right format so OpenCV handles the input correctly.

The function read_cam() takes the opened camera as input. The function uses a loop that will constantly get the current frame that the camera captures by using .read() and it will display this in a window on screen by using cv2.imshow(). It will check for key 27 which is the escape key,

if this key is pressed down the function will break out the loop and the program will terminate and all the windows will be closed.

2.3.2 Harris

Code

The code in Appendix B will open the camera, capture frames and it will mark corners that are detected using the Harris corner detection method.

The main function is at the bottom of the code and from there the other functions will be called. With the function open_onboard_camera() the camera on the Jetson TX2 will be opened, now it can be used to capture frames. The function read_cam uses a loop for capturing, processing and displaying the frames.

In the function read_cam a few variables are set at the beginning. In the while loop a timer will be started, this will be used for calculating the frames per second of the program. After that the frames of the camera will be captured, converted to gray scale and a median blur will be applied. This blur will remove noise from the images. Then the cornerHarris function will calculate the Harris value for each individual pixel. These values will be thresholded and if they are above 0.001 times the maximum Harris value in that frame they will be considered as a keypoint. This will result in areas of keypoints close to each other. For these areas the centers will be determined and the centers of these areas will be marked with a circle.

Also the total amount of detected keypoints of that frame and the average frames per second over the last 10 frames will be written on the displayed frame.

Two if statements will be used for saving the images, one is for recording multiple consecutive images as separate frames and as a video. The other if statement will result in only saving the current frame.

After that the image will be displayed in a opened window.

Then a few statements will be used to check if specific keys are pressed, these will control the different methods of saving the images and it will be able to terminate the program.

Harris feature detection

According to the OpenCV documentation (OpenCV, 2018b), Harris corner detection looks for parts (windows) of an image that give a large variation when moved over the image in different directions. So it calculates the difference between the original window and the moved window and if the difference is above a threshold it is considered a corner.

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2 \tag{2.1}$$

This will result in equation 2.1, where E is the difference between the windows, u and v are the displacement of the window in x and y direction, $w(x,y)$ is the window at the original position, $I(x+u, y+v)$ is the intensity at the moved window and $I(x,y)$ is the intensity at the original window.

Since the detection method is looking for windows with a large variation in intensity the following term 2.2 of the equation 2.1 has to be maximized.

$$\sum_{x,y} [I(x+u, y+v) - I(x,y)]^2 \tag{2.2}$$

This equation can be rewritten using the Taylor series. By using only the first three terms of the Taylor series it translates to the following equation 2.3.

$$E(u,v) \approx \sum_{x,y} [I(x+y) + uI_x + vI_y - I(x,y)]^2 \tag{2.3}$$

Now the square can be expanded which results in equation 2.4, where some terms cancelled each other out.

$$E(u,v) \approx \sum_{x,y} u^2 I_x^2 + 2uvI_xI_y + v^2 I_y^2 \tag{2.4}$$

This equation 2.4 can now be written in matrix form which can be seen in equation 2.5.

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} (\sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}) \begin{bmatrix} u \\ v \end{bmatrix} \tag{2.5}$$

The summed-matrix can be renamed M as in equation 2.6.

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \tag{2.6}$$

This M placed into the original equation 2.5 will give the following equation 2.7.

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} \tag{2.7}$$

The eigenvalues of the matrix can be used to determine if a window holds a corner. A score R can be calculated with the following equation 2.8 in which k is a free parameter.

$$R = det(M) - k(trace(M))^2 \tag{2.8}$$

In this equation 2.8 the $det(M)$ and $trace(M)$ mean the following as in equations 2.9 and 2.10 with λ being the eigenvalues.

$$det(M) = \lambda_1 \lambda_2 \tag{2.9}$$

$$trace(M) = \lambda_1 + \lambda_2 \tag{2.10}$$

If the score R is greater than a certain threshold value the window should be considered as a corner.

2.3.3 FAST

Code

The code in Appendix C will open the camera, capture frames and it will mark corners that are detected using the FAST corner detection method.

The main function is at the bottom of the code and from there the other functions will be called. With the function open_onboard_camera() the camera on the Jetson TX2 will be opened, now it can be used to capture frames. The function read_cam uses a loop for capturing, processing and displaying the frames.

In the function read_cam a few variables are set at the beginning and the FAST feature detector is created. In the while loop a timer will be started, this will be used for calculating the frames per second of the program. After that the frames of the camera will be captured and a median blur will be applied. This blur will remove noise from the images. Then the FAST feature detector will detect the FAST keypoints and it will store them in variable kp. These keypoints will be drawn on the frame by the drawKeypoints function.

Also the total amount of detected keypoints of that frame and the average frames per second over the last 10 frames will be written on the displayed frame.

Two if statements will be used for saving the images, one is for recording multiple consecutive images as separate frames and as a video. The other if statement will result in only saving the current frame.

After that the image will be displayed in a opened window.

Then a few statements will be used to check if specific keys are pressed down, these will control the different methods of saving the images and it will be able to terminate the program.

FAST feature detection

According to the OpenCV documentation (OpenCV, 2018a), the FAST (Features from Accelerated Segment Test) detection method is intended to be fast so it can be used in real time applications. For each pixel is checked if it is a keypoint or not. Every individual pixel has an intensity and this intensity will be compared with surrounding intensities to define if it is a keypoint. Around the pixel a circle of 16 pixels will be taken and a threshold value will be chosen. The intensity of the pixels in the circle will be considered brighter than the center pixel if the intensity is more than the intensity of the center pixel plus the threshold. The intensity of the pixels in the circle will be considered darker than the center pixel if the intensity is less than the intensity of the center pixel minus the threshold. When 12 or more pixels are brighter and/or darker then the center pixel will be determined as a keypoint.

To make it faster not always all 16 pixels of the circle are checked. First only four pixels are tested, the pixels that are centered at the top, bottom, left and right of the circle. At least three of these four must all be brighter or darker, if that is not the case it cannot be a keypoint. But if it is the case then all the other pixels in the circle will also be tested.

2.3.4 Example image with Harris and FAST corner detection

An example image of the diabetic foot model that is used for the experiments is given in Figure 2.10. This same is image is used for an example of Harris corner detection, the result can be seen in Figure 2.11. It is also used for an example of FAST corner detection, the result can be seen in Figure 2.12. The FAST corner detection detects more keypoints on the same image than the Harris corner detection with the default threshold values. The keypoints detected by the Harris method seems to be more specific at corners, while the FAST method also has a lot of keypoints on locations that seems more to be edges/lines than corners. At first glimpse the Harris corner detection seems to be more reliable, because it seems to be more accurate with detecting corners. The reliability of both methods will further be tested on illumination invariance and tracking keypoints from different angles.

2.3.5 Analysis

The code in Appendix D will be used for analyzing the frames and evaluating the corner detection methods.

The basic principle of the analysis is that the program compares keypoints in consecutive frames. It looks if each keypoint is also present in the other frame, it compares the keypoints based on their coordinates in the image. The keypoints are allowed to be a small amount of

Figure 2.10: Example image of diabetic foot model.

Figure 2.11: Example image of diabetic foot model with detected Harris corners.

pixels shifted from each other. This variable amount of pixels can be chosen when the analysis program is executed. The amount of keypoints that are equal between the two frames is divided by the total amount of keypoints present in the two frames. This will give a value which represents the equality between the keypoints of two frames. It is a value between 0 and 1, a closer value to 1 means more equality between the keypoints of the two frames.

First it will ask the user for input, it needs the number of the last frame that needs to be analyzed and the amount of pixels that the keypoints are allowed to be shifted. Then a few dictionaries are created in which the data will be stored and it will add two zero's to the list with coordinates for every keypoint. If a keypoint is already compared and linked with a keypoint in the other frame, than this zero will be set to 1 and the keypoint can not be used for another keypoint

Figure 2.12: Example image of diabetic foot model with detected FAST corners.

anymore. The comparison works in two ways, from the first frame to the second frame and from to second frame to the first frame, for that reason two zero's are added so one for each way of comparison. Then a for loop will start that starts with 0 difference in pixels up to the maximum amount of pixels that is allowed which is chosen by the user. It loops over every keypoint in the first frame and it checks if it is close to a keypoint from the second frame. If it is it will change the 0 for comparison to a 1 and the keypoint can not be used for another keypoint. If the keypoint is close to a keypoint in the other frame it will add the keypoint to a list. The same comparison happens the other way around, so the keypoints from the second frame will be compared with the keypoints from the first frame. After the comparison it will calculate the values for the equality between consecutive frames and it will plot the values of all comparisons in a graph. This graph will be shown and is the final result of the analysis.

3 Results

3.1 Results illumination experiment

The lux values for the different frames are given in 3.1. Both detection methods were applied on the same situation.

Frame number	Lux
0	100
1	200
2	300
3	400
4	500
5	600
6	700
7	800
8	900
9	1000
10	1100
11	1200
12	1300
13	1400
14	1500

Table 3.1: Lux values for the different frames.

3.1.1 Harris

In total 15 frames have been taken. The first frame with an illumination of 100 lux and the detected Harris corners can be seen in Figure 3.1. The last frame with an illumination of 1500 lux and the detected Harris corners can be seen in Figure 3.2.

Figure 3.1: Harris corner detection with an illumination of 100 lux.

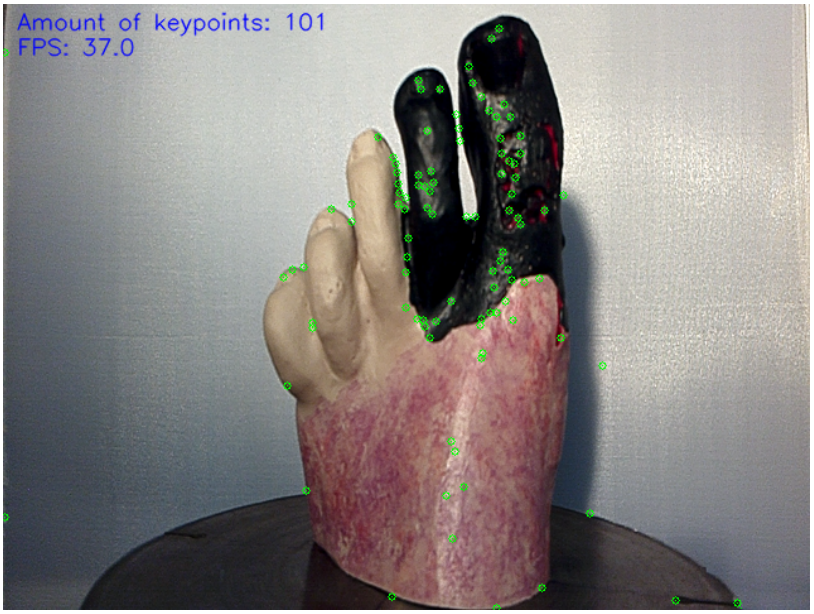

Figure 3.2: Harris corner detection with an illumination of 1500 lux.

In Figure 3.3 a graph with the analyzed data of the Harris corner detection with different illumination is shown. The amount of pixels that the keypoints were allowed to be shifted was 3.

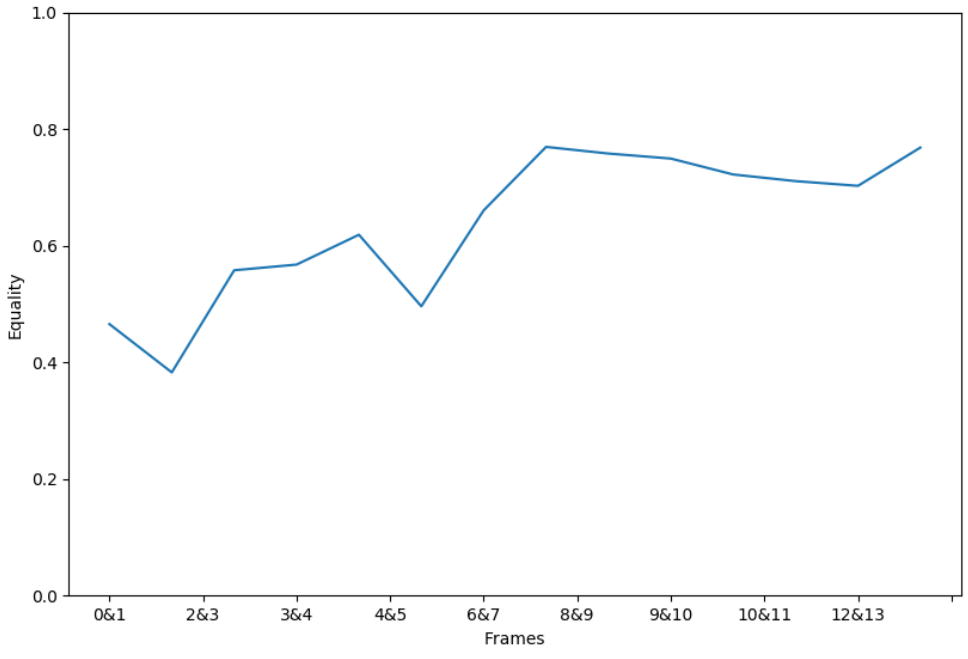

Figure 3.3: Graph with equality of frames for Harris with different illumination.

3.1.2 FAST

In total 15 frames have been taken. The first frame with an illumination of 100 lux and the detected FAST corners can be seen in Figure 3.4. The last frame with an illumination of 1500 lux and the detected FAST corners can be seen in Figure 3.5.

Figure 3.4: FAST corner detection with an illumination of 100 lux.

Figure 3.5: FAST corner detection with an illumination of 1500 lux.

In Figure 3.6 a graph with the analyzed data of the FAST corner detection with different illumination is shown. The amount of pixels that the keypoints were allowed to be shifted was 3.

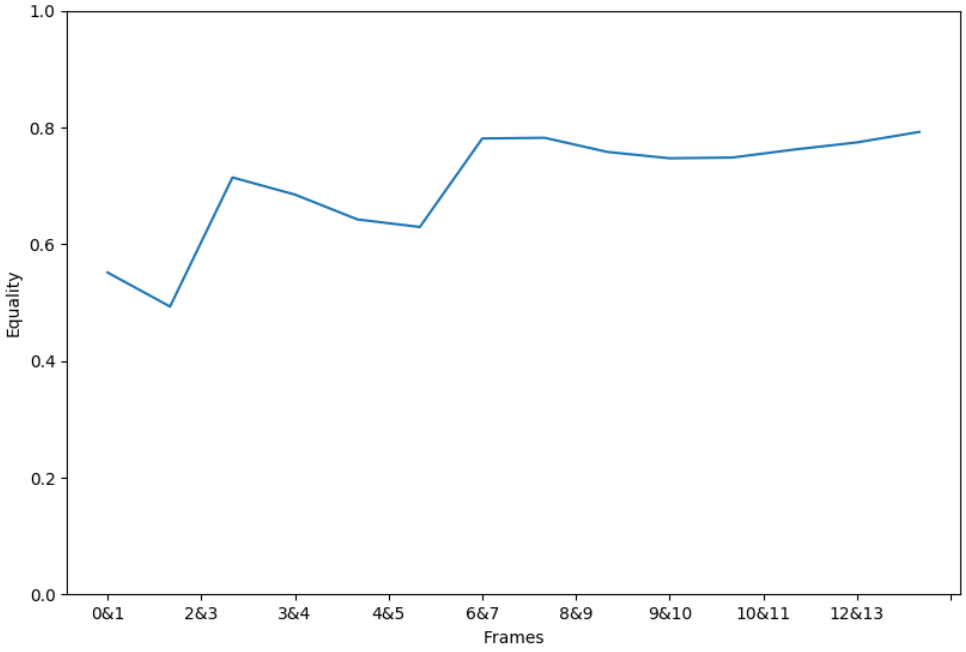

Figure 3.6: Graph with equality of frames for FAST with different illumination.

3.2 Results rotation experiment

The foot model was rotated approximately 180 degrees. This was done in approximately 3 seconds. Both detection methods had their own rotation, so the real-time detection was not done simultaneous on the same rotation. The frames per second were based on their performance and not fixed, this causes that the detection methods have a different amount of frames that were taken during the rotation of the foot model.

3.2.1 Harris

In total 109 frames have been taken. The first frame of the rotation can be seen in Figure 3.7 and the last frame of the rotation can be seen in Figure 3.8.

In Figure 3.9 a graph with the analyzed data of the Harris corner detection during the rotation is shown. The amount of pixels that the keypoints were allowed to be shifted was 3.

3.2.2 FAST

In total 173 frames have been taken. The first frame of the rotation can be seen in Figure 3.10 and the last frame of the rotation can be seen in Figure 3.11.

In Figure 3.12 a graph with the analyzed data of the FAST corner detection during the rotation is shown. The amount of pixels that the keypoints were allowed to be shifted was 3.

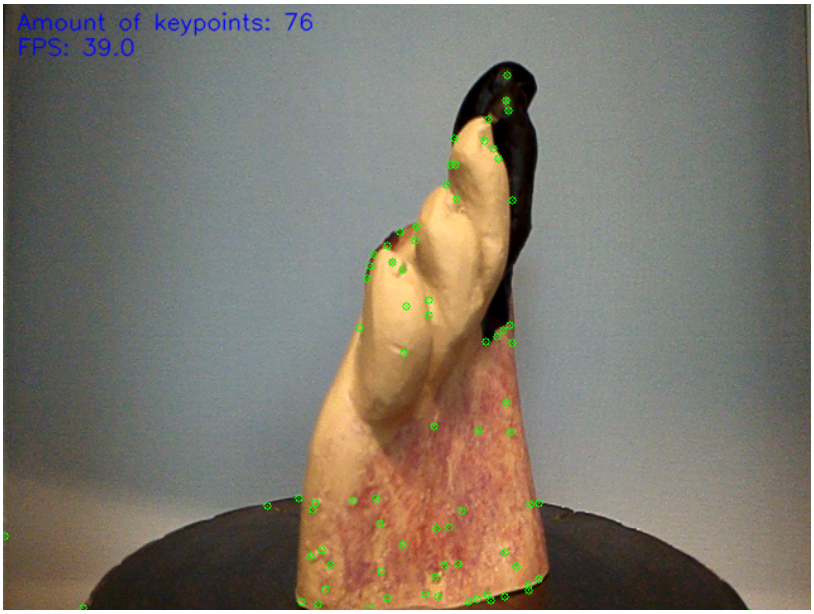

Figure 3.7: Harris corner detection at the first frame of the rotation.

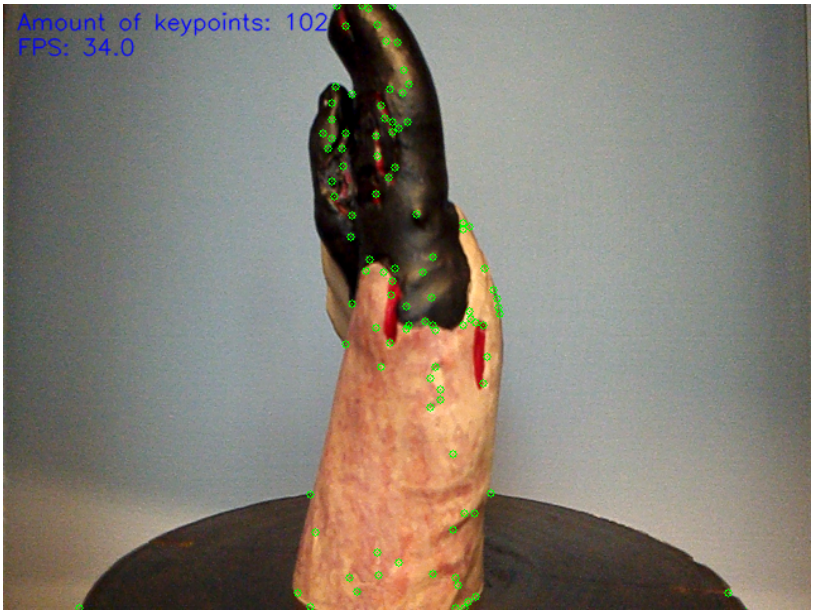

Figure 3.8: Harris corner detection at the last frame of the rotation.

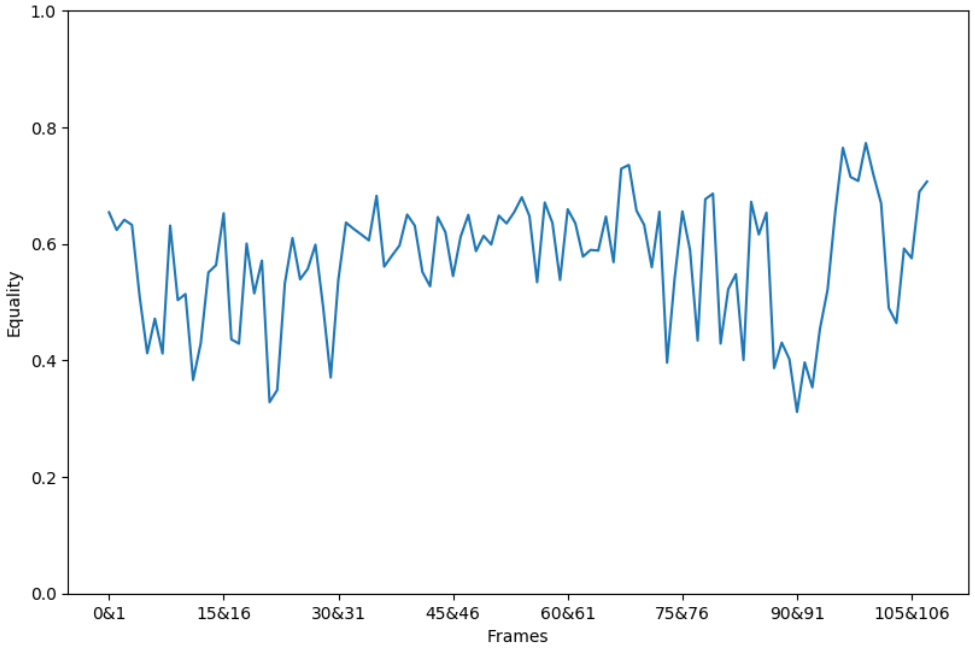

Figure 3.9: Graph with equality of frames for Harris during the rotation.

Figure 3.10: FAST corner detection at the first frame of the rotation.

Figure 3.11: Fast corner detection at the last frame of the rotation.

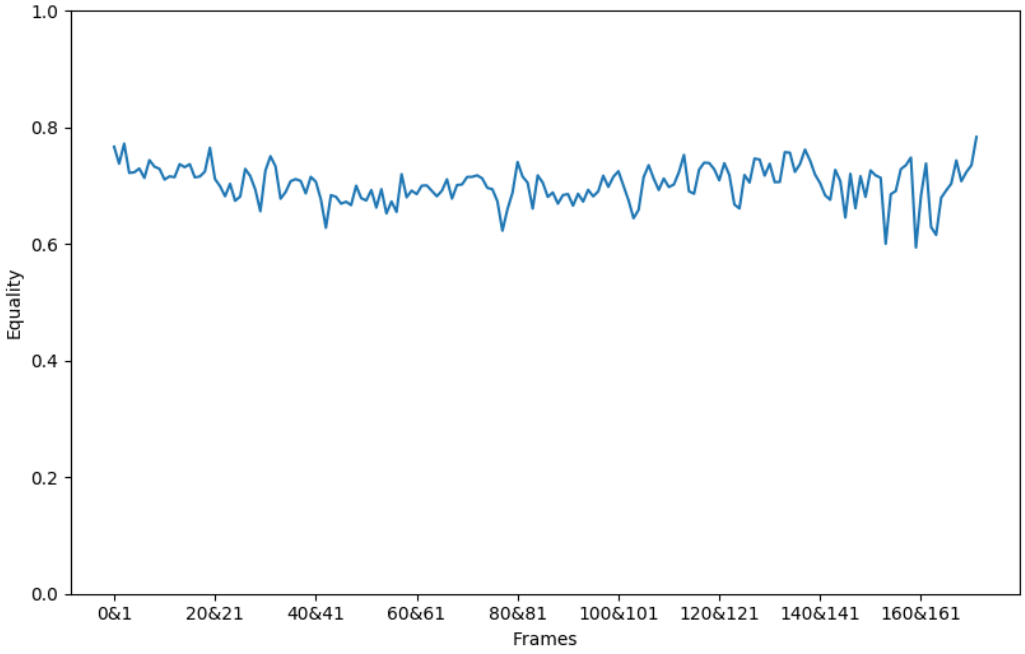

Figure 3.12: Graph with equality of frames for FAST during the rotation.

4 Discussion

4.1 Significance illumination experiment

By comparing Figure 3.3 and Figure 3.6 a few things can be noticed. With a lower level of lighting the equality of the keypoints with FAST corner detection is better than with Harris corner detection. With a higher level of lighting the equality of keypoints increases for both detection methods. Still has FAST a bit more equality than Harris. So with different lighting conditions FAST seems to be a more reliable detection method than Harris. FAST does detect more keypoints than Harris with the current set of variables. This can have as a result that FAST keypoints have more chance to be closer to each other, because there are more keypoints detected. Although every keypoint can only be linked once to another keypoint, this might still give FAST an advantage over Harris.

4.2 Significance rotation experiment

By comparing Figure 3.9 and Figure 3.12 it can be noticed that the equality of FAST is higher than Harris most of the time. With this result it seems that FAST corner detection is more reliable in detecting keypoints from different angles than Harris corner detection. But in similarity with the illumination experiment does FAST detect more keypoints than Harris, which can also give it an advantage in this experiment.

4.3 Comparing illumination experiment results with user case scenario

It is useful to compare the results of the illumination experiment with the user case scenario. The measurement in the hospital that can be seen in Figure 2.9 indicates that above the treatment table the level of lighting was around 1000 lux. For the illumination experiment for frame number 9 the lighting level was 1000 lux. In Figure 3.3 it shows that from around 900 lux until 1500 lux the results of Harris corner detection are at its highest and fairly constant. So for Harris corner detection the user case scenario of 1000 lux is a good condition according to the results of the experiment. In Figure 3.6 it shows that from around 800 lux until 1500 lux the results of FAST corner detection are at its highest and fairly constant. So also for FAST corner detection the user case scenario of 1000 lux is a good condition according to the results of the experiment.

5 Conclusion

The illumination results show that FAST corner detection was more reliable with different levels of illumination than Harris corner detection. This suggests that the illumination invariance of the FAST corner detection method is better. Illumination invariance is an important feature for a detection method, because it will make the results of the method more reliable due to less influence of change of illumination. FAST also seems to be more reliable for a wider range of lux values than Harris is. It was more reliable for lower lux values than Harris. The level of illumination of the user case scenario was also part of this range. So for the user case in the hospital FAST seems to have a better reliability and has a bigger margin around the user case were the reliability stays high.

The rotation results show that FAST corner detection was more reliable in detecting keypoints from different angles than Harris corner detection. FAST had more equality in the keypoints of different frames which suggests that the same keypoints were better detected from different angles. This feature will give the FAST detection method a better reliability than Harris when the orientation of the camera is changed in relation to the object.

An additional advantage of the FAST corner detection method is that the workload is lower than the workload of the Harris corner detection method. In case of these experiments it resulted in a higher rate of the frames per second for the output of the FAST corner detection than the output of Harris. For further development the lower workload of FAST can be useful. By limiting the frame rate, FAST can have a sufficient performance with less load on the hardware. This can then be used for other calculations of additions that the software will get in further development.

A single 2D image is not capable of dealing with the depth changes of the ulcer on the diabetic foot. For a better estimation of the shape and size of the ulcer a 3D model can be a solution. With the detected corners it should be possible to recognize patterns in different images. A further research can focus on the recognition of equal patterns of keypoints in different images. These patterns can be used to determine the relative orientation of different images. Multiple frames from different points of view and their relative orientation will possibly be very useful information for the reconstruction of a 3D image.

The OpenCV library was used for this research. It is a popular library for computer vision applications and it has a lot of documentation and examples. This was a useful library to get started with computer vision on the Jetson TX2 and exploring the capabilities of the Jetson TX2. The OpenCV library does have CUDA support to enhance the performance on the Jetson TX2. But NVIDIA also has their own toolkit for computer vision and image processing called VisionWorks. This toolkit includes OpenVX, which is an open, royalty-free standard for cross platform acceleration of computer vision applications. It is complementary to the OpenCV library and in some applications it offers a better optimized graph management than OpenCV (Wikipedia contributors, 2018). So for better optimization and to use the Jetson TX2 more efficiently it can be useful to do further research in the implementation of OpenVX.

A Appendix 1

A.1 Open on-board camera and display frames

This code will open the on-board camera on the Jetson TX2 developer kit and it will display the captured frames in a window.

```
import cv2

def open_onboard_camera():
    return cv2.VideoCapture("nvcamerasrc ! video/x-raw(memory:NVMM)
        , width=(int)640, height=(int)480, format=(string)I420,
        framerate=(fraction)60/1 ! nvvidconv ! video/x-raw, format
        =(string)BGRx ! videoconvert ! video/x-raw, format=(string)
        BGR ! appsink")

def read_cam(video_capture):
    if video_capture.isOpened():
        while True:
            retval, frame = video_capture.read()
            cv2.imshow("Window", frame)
            key=cv2.waitKey(10)
            if key == 27:
                break;
    else:
        print("camera open failed")

if __name__ == '__main__':
    print("OpenCV version: {}".format(cv2.__version__))
    video_capture=open_onboard_camera()
    read_cam(video_capture)
    video_capture.release()
    cv2.destroyAllWindows()
```

B Appendix 2

B.1 Open on-board camera and display frames with marked Harris corners

This code will open the camera, capture frames and it will mark corners that are detected using the Harris corner detection method.

```
import cv2
import numpy as np
import time
import collections

def open_onboard_camera():
    return cv2.VideoCapture("nvcamerasrc ! video/x-raw(memory:NVMM)
        , width=(int)640, height=(int)480, format=(string)I420,
        framerate=(fraction)60/1 ! nvvidconv ! video/x-raw, format
        =(string)BGRx ! videoconvert ! video/x-raw, format=(string)
        BGR ! appsink")


def read_cam(video_capture):
    if video_capture.isOpened():
        windowName = "Harris"
        cv2.namedWindow(windowName, cv2.WINDOW_NORMAL)
        cv2.resizeWindow(windowName,640,480)
        cv2.moveWindow(windowName,0,0)
        cv2.setWindowTitle(windowName,"Harris Feature Detection")
        queue = collections.deque(maxlen = 10)
        fourcc = cv2.VideoWriter_fourcc(*'MJPG')
        out = cv2.VideoWriter('output.avi',fourcc, 20.0, (640,480))
        record = False
        save = False
        framenumb = 0
        while True:
            start = time.time()
            if cv2.getWindowProperty(windowName, 0) < 0: # Check to
                see if the user closed the window
                # This will fail if the user closed the window;
                    Nasties get printed to the console
                break;
            retval, frame = video_capture.read()
            gray=cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            gray=np.float32(gray)
            gray = cv2.medianBlur(gray, 5)
            dst=cv2.cornerHarris(gray,2,3,0.04)

            ret, dst = cv2.threshold(dst,0.001*dst.max(),255,0)
            dst = np.uint8(dst)

            ret, labels, stats, centroids = cv2.
                connectedComponentsWithStats(dst)
```

```
            points = np.int0(centroids)

            frame[points[:,1],points[:,0]]=[0,255,0]
            for i in range(0,len(points)-1):
                cv2.circle(frame, (points[i,0],points[i,1]), 3,
                    [0,255,0])

            textkp = "Amount of keypoints: " + str(len(points))
            cv2.putText(frame, textkp, (11,20), cv2.
                FONT_HERSHEY_SIMPLEX, 0.6, (255,0,0), 1, cv2.
                LINE_AA)

            end = time.time()
            seconds = end - start
            queue.append(seconds)
            textfps = "FPS: " + str(round(1/(sum(queue)/10)))
            cv2.putText(frame, textfps, (11,40), cv2.
                FONT_HERSHEY_SIMPLEX, 0.6, (255,0,0), 1, cv2.
                LINE_AA)

            if record == True:
                out.write(frame)
                cv2.imwrite('image'+str(framenumb)+'.png', frame)
                np.savetxt('array'+str(framenumb)+'.txt', points)
                framenumb += 1

            if save == True:
                cv2.imwrite('image.png', frame)
                np.savetxt('array.txt', points)
                save = False

            cv2.imshow(windowName,frame)
            key=cv2.waitKey(10)

            if key == 27: # Check for ESC key
                cv2.destroyAllWindows()
                break ;
            elif key == 49: #1 key
                record = True
                print("Recording ON")
            elif key == 50: #2 key
                record = False
                print("Recording OFF")
            elif key == 51: #3 key
                save = True
    else:
     print ("camera open failed")


if __name__ == '__main__':
```

```
print("OpenCV version: {}".format(cv2.__version__))
video_capture=open_onboard_camera()
read_cam(video_capture)
video_capture.release()
cv2.destroyAllWindows()
```

C Appendix 3

C.1 Open on-board camera and display frames with marked FAST corners

This code will open the camera, capture frames and it will mark corners that are detected using the FAST corner detection method.

```
import cv2
import numpy as np
import time
import collections

def open_onboard_camera():
    return cv2.VideoCapture("nvcamerasrc ! video/x-raw(memory:NVMM)
        , width=(int)640, height=(int)480, format=(string)I420,
       framerate=(fraction)60/1 ! nvvidconv ! video/x-raw, format
       =(string)BGRx ! videoconvert ! video/x-raw, format=(string)
       BGR ! appsink")


def read_cam(video_capture):
    if video_capture.isOpened():
        windowName = "FAST"
        cv2.namedWindow(windowName, cv2.WINDOW_NORMAL)
        cv2.resizeWindow(windowName,640,480)
        cv2.moveWindow(windowName,0,0)
        cv2.setWindowTitle(windowName,"FAST Feature Detection")
        queue = collections.deque(maxlen = 10)
        fourcc = cv2.VideoWriter_fourcc(*'MJPG')
        out = cv2.VideoWriter('output.avi',fourcc, 20.0, (640,480))
        record = False
        save = False
        framenumb = 0
        fast=cv2.FastFeatureDetector_create()
        while True:
            start = time.time()
            if cv2.getWindowProperty(windowName, 0) < 0: # Check to
                see if the user closed the window
                # This will fail if the user closed the window;
                    Nasties get printed to the console
                break;
            retval, frame = video_capture.read()
            frame = cv2.medianBlur(frame, 5)
            kp=fast.detect(frame,None)
            frame = cv2.drawKeypoints(frame,kp,None,color=(0,0,255)
                )

            textkp = "Amount of keypoints: " + str(len(kp))
            cv2.putText(frame, textkp, (11,20), cv2.
                FONT_HERSHEY_SIMPLEX, 0.6, (255,0,0), 1, cv2.
                LINE_AA)
```

```
            end = time.time()
            seconds = end - start
            queue.append(seconds)
            textfps = "FPS: " + str(round(1/(sum(queue)/10)))
            cv2.putText(frame, textfps, (11,40), cv2.
                FONT_HERSHEY_SIMPLEX, 0.6, (255,0,0), 1, cv2.
                LINE_AA)

            if record == True:
                out.write(frame)
                cv2.imwrite('image'+str(framenumb)+'.png', frame)
                points = cv2.KeyPoint_convert(kp)
                np.savetxt('array'+str(framenumb)+'.txt', points)
                framenumb += 1

            if save == True:
                cv2.imwrite('image.png', frame)
                points = cv2.KeyPoint_convert(kp)
                np.savetxt('array.txt', points)
                save = False

            cv2.imshow(windowName,frame)
            key=cv2.waitKey(1)

            if key == 27: # Check for ESC key
                cv2.destroyAllWindows()
                break ;
            elif key == 49: #1 key
                record = True
                print("Recording ON")
            elif key == 50: #2 key
                record = False
                print("Recording OFF")
            elif key == 51: #3 key
                save = True


    else:
     print ("camera open failed")


if __name__ == '__main__':
    print("OpenCV version: {}".format(cv2.__version__))
    video_capture=open_onboard_camera()
    read_cam(video_capture)
    video_capture.release()
    cv2.destroyAllWindows()
```

D Appendix 4

D.1 Analyze the frames and evaluate the corner detection methods

This code will be used for analyzing the frames and evaluating the corner detection methods.

```
import numpy as np
from collections import defaultdict
import matplotlib.pyplot as plt

print("Enter the number of the last frame")
lastframe = input("Number of last frame: ")

print("Enter the amount of pixels that the keypoints can be
    different")
inputdif = input("Amount of pixels: ")

allframes = {}
closekpftof1 = defaultdict(list)
closekpf1tof = defaultdict(list)
ratios = {}
plotvals = []

for i in range(0, lastframe+1):
    keypoints = np.loadtxt("array"+str(i)+".txt")
    listkp= []
    for j in range(0, keypoints.size/2):
        listkp.append(np.append(keypoints[j],[0, 0]).tolist())
    allframes['frame{0}'.format(i)] = np.asarray(listkp)

for dif in range(0, inputdif+1):
    for i in range(0, lastframe):
        for j in range(0,(allframes['frame'+str(i)]).size/4):
            for k in range(0,(allframes['frame'+str(i+1)]).size/4):
                if np.isclose((allframes['frame'+str(i)][j][0]), (
                    allframes['frame'+str(i+1)][k][0]), atol=dif):
                    if np.isclose((allframes['frame'+str(i)][j][1])
                        , (allframes['frame'+str(i+1)][k][1]), atol
                        = dif):
                        if allframes['frame'+str(i)][j][2] == 0:
                            allframes['frame'+str(i)][j][2] = 1
                            closekpftof1['frame{0}'.format(i)].
                                append(allframes['frame'+str(i)][j]
                                .tolist())
    for i in range(1, lastframe+1):
        for j in range(0,(allframes['frame'+str(i)]).size/4):
            for k in range(0,(allframes['frame'+str(i-1)]).size/4):
                if np.isclose((allframes['frame'+str(i)][j][0]), (
                    allframes['frame'+str(i-1)][k][0]), atol = dif)
                    :
```

```
                        if np.isclose((allframes['frame'+str(i)][j][1])
                            , (allframes['frame'+str(i-1)][k][1]), atol
                             = dif):
                             if allframes['frame'+str(i)][j][3] == 0:
                                 allframes['frame'+str(i)][j][3] = 1
                                 closekpf1tof['frame{0}'.format(i)].
                                     append(allframes['frame'+str(i)][j]
                                     .tolist())

for i in range(0, lastframe):
    totalkps = (allframes['frame'+str(i)].size/4)+(allframes['frame
        '+str(i+1)].size/4)
    equalkps = len(closekpftof1['frame'+str(i)])+len(closekpf1tof['
        frame'+str(i+1)])
    ratio = float(equalkps/float(totalkps))
    ratios['ratio{0}&{1}'.format(i, i+1)] = ratio

for i in range(0, lastframe):
    plotvals.append(ratios['ratio'+str(i)+'&'+str(i+1)])

a = plotvals
b = range(0,len(plotvals))
for i in b:
    b[i] = str(i)+'&'+str(i+1)

plt.plot(b, a)
plt.show()
```

Bibliography

Bradski, G. (2000), The OpenCV Library, *Dr. Dobb's Journal of Software Tools*.

Li, J., L. Jiang, T. Li and X. Liang (2018), Application of 3D reconstruction system in diabetic foot ulcer injury assessment, **vol. 1955**, no.1, p. 040119, doi:10.1063/1.5033783.
https://aip.scitation.org/doi/abs/10.1063/1.5033783

OpenCV (2018a), FAST Algorithm for Corner Detection - OpenCV 3.4.5 documentation.
https://docs.opencv.org/3.4.5/df/d0c/tutorial_py_fast.html

OpenCV (2018b), Harris Corner Detection - OpenCV 3.4.5 documentation.
https://docs.opencv.org/3.4.5/dc/d0d/tutorial_py_features_harris.html

Wikipedia contributors (2018), OpenVX — Wikipedia, The Free Encyclopedia, [Online; accessed 21-January-2019].
https://en.wikipedia.org/w/index.php?title=OpenVX&oldid=865189217