NTNU
Norwegian University of
Science and Technology

Adaptive Gripping Technology

Development of a gripper interface for SCHUNK Dextrous Hand

Sølve Jonathan Monteiro

Master of Science in Engineering Cybernetics
Submission date: June 2010
Supervisor: Geir Mathisen, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

Use of robotic systems in industrial processes offers huge potential in economic benefits, performance, safety and precision. One of the most important aspects of robotic manipulation is the gripping process. Grippers range in complexity from the simplest single-actuated 1-movable part jaws, up to very advanced multi-fingered multi-jointed hand-like grippers. More complex gripping operations require advanced control systems in order to replace the dexterity, intuition and reflexes of human hands.

This thesis will explore the possibilities of the SCHUNK Dextrous Hand (SDH), a 3-fingered gripper with 7 degrees of freedom and tactile feedback sensors in the fingers. By creating a feedback control system, this gripper can be used to grip unknown and fragile objects.

Tasks:
1) Do a literature survey in the field of robotic gripping
2) Design a system for automatic gripping and holding objects, that involves both a manipulator arm and the SCHUNK Dextrous Hand.
3) Implement the gripper part of the above mentioned control system that can grip objects of various shapes and sizes and of a fragile nature.

Assignment given: 15. January 2010
Supervisor: Geir Mathisen, ITK

Abstract

The use of robotic grippers offers huge potential benefits in industrial settings. The more advanced a gripper is, the more uses it can have, thus offering large economic benefits. On the other hand, the more complex a gripper is, the more advanced its control system needs to be, in order to control it effectively and safely. This thesis will focus on controlling SCHUNK Dextrous Hand (SDH), a 3-fingered robotic gripper with 7 degrees of freedom and tactile sensors in the fingers. By creating a real-time control system the sensors in the fingers can be used to make a feedback loop that controls the fingers. This is the basis for creating an adaptive gripper that can grip objects of unknown shape, size or position. This control system in combination with a controller for a robotic manipulator arm lets the gripper attempt to grip objects even if they are out of reach. By passing requests up to an overall control system, the gripper can request a translation to a position that gives it a better chance at performing a successful grip on the targeted object. In this project, the controller for the gripper is created, and the communication to and from the manipulator control system is replaced with a simple user-interface. This user interface offers a way of testing the complete system without the use of a manipulator arm. The translations from the gripper is read out, and the target object is moved by hand, in the opposite direction. This solution offers a simple way to expand the system to include the manipulator and its control system in later editions.

Initial experiments were successful, with the gripper successfully able to pick up different objects. An apple, an empty soda can (both upright and lying down) and a chocolate egg were all picked up and held firmly without damaging the object. Complications arose with regards to the sensitivity of the sensors. They were generally unable to register any pressure when the fingers came in contact with lighter objects, and had to push the objects against the other two fingers. Another problem that arose was the stability of the application created. The program was based on multi-threading, and real-time sensor analysis. The application crashing did cause some objects to be damaged in the experiments, as the application logic could not halt the fingers despite pressure being registered. Future work should focus on restructuring the application logic to improve the stability, and the control system for the manipulator arm.

Preface

This thesis is the result of work done in the final semester of my Master of Science Degree in Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU). This thesis can be considered the continuation of the project done in the fall of 2009, under the same supervisors and using much of the same equipment.

I would like to thank my supervisor Associate Professor Geir Mathisen and co-supervisor Research Scientist Terje Mugaas for their continued help throughout the project. Thanks to my friends and family for ideas and input during the work, and especially Åshild Breivik without whom I would never have reached this far.

Sølve J. Monteiro
Trondheim, June 2010

Contents

1 Introduction

Figure 1: Typical industrial robots working in an auto plant. These robots can work at a pace and a precision unmatched by humans. Image ©www.harting-mitronics.ch

1.1 Background

Robotic gripping is a very important part of modern industry, and can be seen as a replacement for humans hands doing the same job. Working in hazardous conditions, long hours, and with extremely high precision is no problem for a robotic hand. With computerized systems, one avoids human mistakes and traits such as bad judgement, fatigue, carelessness and imprecision. With technology, the scope of their possible tasks has increased wildly, making possible tasks from moving red-hot iron bars, sorting bottles, to holding microscopic tools for the microchip industry. Robotic technology is especially relevant in high-income countries where manual labour is expensive. In such cases economic benefits can be made by replacing manual labour with autonomous robotic handling. Figure 1 shows such a typical scenario, a modern auto-manufacturing plant with robots doing most of the grunt work. However, as more and more tasks are considered for robotic manipulators, their control systems must become increasingly powerful in order to work effectively. Parallel to the benefits of using a computer for the task, there are also drawbacks. A computer does not have instinct, judgement or creativity to solve unknown problems. A robot can, and will always, only do as it is programmed.

Consider the handling of fragile objects. Figure 2 is a nice example of the usage of an adaptive gripping technology. As can be seen from the image, the target objects are of variable shape. Not only that, but the fragile nature of light bulbs requires the gripper to handle them with care. They must be held firmly enough to not slip out of the grip, while at the same time they must not be crushed by an overpowering grip. This can also relate to for example the food industry, where one would suffer economic losses if the handling of the product destroyed or damaged it's shape. Another aspect

to be considered is adaptive gripping. By this it is meant that the object to be gripped is not in a pre-programmed or pre-defined position, and the control system itself must figure out where to grip, how to close the grip and how firmly to grip. This can relate to almost any process where objects are not placed in an exact position and orientation. It is even more relevant when the objects to be picked up not necessarily are equal or even similar in shape. One example of this is sea-food processing plants, where the fish come in various sizes. On top of that, the objects are not necessarily of a firm nature and might bend, squash or deform otherwise. This makes it hard for automatic machinery to do the processing, and often results in people having to do the manual labour. In summary, by adaptive gripping technology, the focus is on gripping objects of unknown position, shape, orientation or firmness. The fragility of an object also needs an adaptive gripping mechanism to apply the least amount of pressure on the object without it slipping.

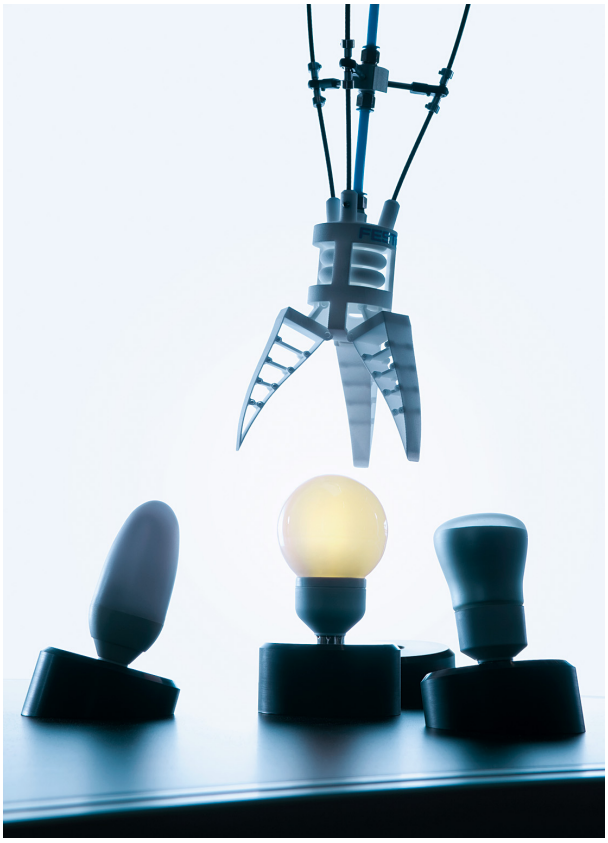


Figure 2: A possible scenario where an adaptive grip is needed. The varied shape and the fragility of the light bulbs requires a varying grasp and force. Image ©www.festo.com

1.2 Scope

In this paper the action of gripping a fragile object will be considered, where the position is not pre-defined. This will then fall under two of the mentioned criteria for adaptive gripping: The unknown position and the fragile nature of the object. For a robotic system

to perform a grip, picking up an object, moving it or processing it, it will normally be attached to some sort of robotic arm. This arm is outside the scope of this text and only the operation of the gripper will be considered. The gripper tool used is the three-fingered SCHUNK Dextrous Hand (SDH) [3]. This gripper has pressure feedback sensors in the finger tips, and these will be used to devise a gripping algorithm that will lift the object firmly, but not too hard. These pressure sensors are the only feedback system available, so they will also be used for finding the position of the object, by doing a grasping motion (imagine being blindfolded and trying to grip an unknown object). The gripper cannot actually move itself (only its fingers) so it needs a feedback system to the manipulator arm. Communication to and from this higher-level control over both arm and gripper will be abstracted for future implementation.

This project will experiment with picking up a fragile egg-like object, and see if it is possible to find and maintain a firm grip on it. The surrounding control system and some protocols will be discussed and if possible evaluated. A part of this paper will also look into previous work considered relevant in this field.

1.3 Thesis Outline

This paper is divided into 4 main parts: literature, design and implementation and conclusion. Chapter 2 will look at some earlier work done in the robotics field considered relevant. It will focus on object detection and gripping technology, and attempt to refine some useful ideas for this project. In chapters 3 and 4 an initial strategy for the design and implementation of the project will be created. By using parts of the theory from the first section, it will focus on the parts of this project that will be implemented (the gripper controller). Chapters 5 to 7 will revolve around the practical aspect of this project: The programming, the testing and the results of the testing. The ideas from chapter 3 and 4 will be refined and discussed throughout the thesis. The last section of this paper, chapters 7 and 8, will summarise the results of the testing and compare the solution to the one devised in chapter 4. The problems encountered and the potential for improvement in the future will be discussed.

2 Relevant previous work

This section explores some previous work done in the field of gripping and robotic object detection. Some examples have been included to demonstrate the wide variety of applications a mechanical gripper can have. Some work regarding object detection for gripping systems and gripping algorithms has also been reviewed.

2.1 Mechanical Grippers

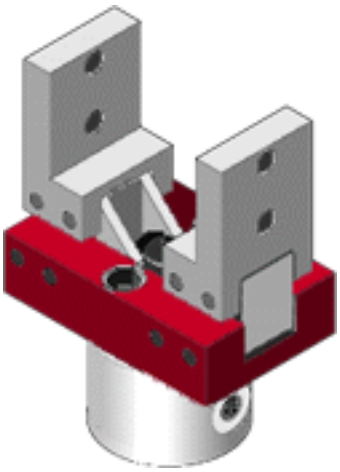


Figure 3: Barrington Automation B-0P Mini Gripper

Often called End Of Arm Tooling (EOT) or end-effectors, the gripper is the robot manipulators interface by which it performs most operations. Grippers are divided into four categories (Hardin 2005 [7]): pneumatic, vacuum, hydraulic and electric. While pneumatic grippers are the most common, hydraulics are generally more powerful. Electric grippers are more versatile than the pneumatic and hydraulic ones, and make up for the lack of power with versatility and controllability. Vacuum grippers operate by creating a very low pressure and funnelling this through a suction cup to grip objects. This vacuum can be achieved either by an *aspirator / Venturi Pump* (forcing a fluid through a narrower tube, causing speed increase and lower pressure) or a remote pump connected by tubing.

Grippers vary greatly in both size and function. The simplest grippers have only one movable joint and a single servo controlling it, allowing for a basic jaw motion. As an example, consider figure 3, a simple double-actuated gripper produced by Barrington Automation. This gripper is powered by pneumatics and is specified for loads up to about 230 grams. It is simple and robust, and serves a quite direct purpose. In the other end of the spectrum there are very advanced grippers, like Ishikawa Komuro Laboratory's high-speed hand [11], seen in figure 4.

This three-fingered hand-like gripper features an advanced combination of optical and tactile sensing, and very fast processing. It has demonstrated its impressive dexterity by performing feats such as spinning a pen around its fingers, throwing a ball with precision, dribbling a bouncing ball, tying knots and catching falling objects.

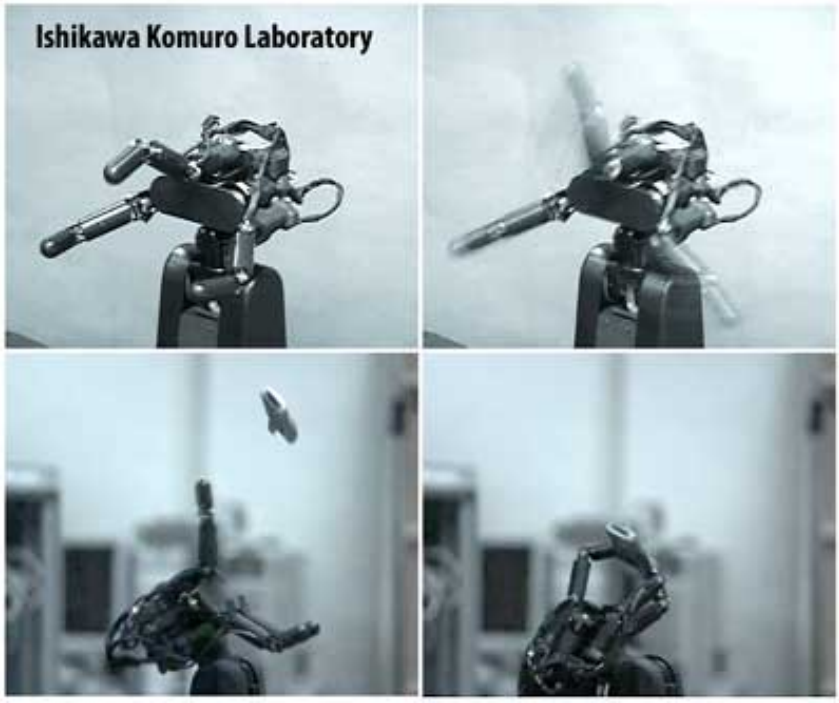


Figure 4: Ishikara Komuru Laboratory's Sensor Fusion Hand

The size of the grippers are wide ranging, from the smallest grippers like NanoEffector in figure 5 up to huge gripping systems like the ones applied in for example auto junk yards.

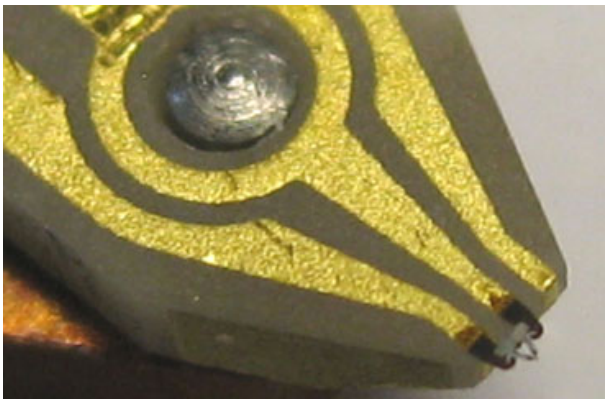


Figure 5: Zyvex' NanoEffector, capable of manipulating components from 1 to 500 μm in size. Image ©Zyvex Instruments.

Although definitely more impressive than most simple industrial grippers, the cost and complexity of a robotic gripper such as Sensor Fusion will make it overly advanced for most industrial and manufacturing processes. This does however make great contributions in the field of robotic gripping and expands the types of applications robots can be used for. With its precise and effective control, it is not hard to imagine how a tool like this could be used in applications such as medicine, advanced manufacturing and

prosthesis technology. Industries mostly utilize the much simpler and cheaper grippers, and employ them to relatively basic tasks, mainly because it is easier and cheaper to leave the complex tasks to humans. Examples of such a task can be complex movements, evaluating quality and sorting of objects on a conveyor belt. It is possible to make machines that can do the same tasks, but it is often harder to defend this cost. One way to reduce the cost of such operations, is to create more adaptive and simpler to use algorithms for the grippers. A system that could adapt itself to grip a wider variety of tasks would save both time and cost if implemented effectively.

The SCHUNK Dextrous Hand enters in a middle ground in between these two extremities described above. It is a 3-fingered electrically powered gripper with tactile sensor matrices in the fingers. It is not close to as fast as the Sensor Fusion, but it is still far too complicated and costly to be useful in many standard industrial processes. However because of its feedback system and flexibility it has the potential to become a very versatile tool that could excel in areas where the standard gripper could not, and the extra cost of the SDH could cancel out the cost of using several different grippers for tasks that could be done with a single gripper.

2.2 Object detection

The first step of adaptive gripping should necessarily be the object detection. Wether it be optically, by radar, sonar, laser or tactile sensing, the gripper needs to know where to grip. A few different methods to detect objects before the actual gripping begins will be discussed.

Contact sensing: Robert D. Howe describes two main methods to sense contact (Howe 1993 [8]); *kinaesthetic* sensing measures limb motion and forces with internal receptors. One example of this type of sensing can be a torque sensor in an electric motor, or the tendons in our body. *Cutaneous* sensing, on the other hand, uses receptor in the skin or contact surface to detect contact. Based on An, Atkeson, Hollerbach 1988 ([6]) it is relatively hard to sense and control positions and forces based on actuator signals alone, and a cutaneous feedback element will give a much higher precision. There are several ways to measure cutaneous activity, the following are the most common: Piezoelectric, Piezoresistive, capacitive and inductive. The Piezoelectric crystal is a sensor that converts pressure or strain into an electric signal. Of the four technologies mentioned, this is the most sensitive and robust. It is insensitive to electromagnetism and radiation, and therefore enables measuring in harsh conditions. The main disadvantage of this is that it measures a change in force, not actual force, and is thus not optimal for static measurements (this can still be achieved by integrating the changes over time). The piezoresistive effect differs from the piezoelectric in that it does not produce any potential itself, only changes its resistive properties relative to the force applied. Capacitive sensing is a sensor technology that does not rely on contact to the sensors, but uses the conductivity of the object in question to sense position and change of position. This has the obvious disadvantage of not being able to sense non-conductive objects directly. The inductive sensor (also known as the Eddy Current Displacement Sensor) shares many of the properties of the capacitive, but instead of using an electric field it uses a magnetic

field. It is slightly less sensitive than the capacitive sensor, but is better suited for harsh environments.

Detecting contact by measuring the force, either in separate sensors in the joints, or by torque measurements in the actuators, can also help in identifying contact. The Jacobian matrix describes the kinematic chain of positions, velocities and acceleration from a static point to an end-point, and by looking at the transpose of this matrix, the force of contact can be determined. The obvious problem with this method is that it will only measure forces in the actuating directions.

One very important aspect of contact sensing is the detection of slippage. This is a good indicator for when to tighten or improve the grip. If the sensor is in the form of a matrix (e.g. the Weiss Robotic sensors in the SDH) a **pressure distribution** is useful for determining sliding. If the slip is translational (like sliding out of a grip), the force required to slip is given by the friction coefficient of the materials multiplied by the total normal force. Using Coulomb friction this is independent of the details of pressure distribution. For slippage in rotation, pressure information is required. Generally, the smaller the pressure area is, the less torque can be sustained before slippage. For both translational and rotational slippage, relation between force and torque is complex.

Object shape may be determined grossly by letting the robot fingers explore the surface in a *groping mode* (Allen 1987, Bay 1989). This data can be correlated by non-contact sensors (optical, etc.) and help provide surface normals (vectors) to prevent slip, since the ratio of tangential force to surface normal will determine if slipping will occur. Curvature of the object can be determined effectively by matrix sensors, and also by small movement of finger tips at the contact location. In general smaller tasks require sensors closer to the contact point since this gives less interference from inertia and parts of the manipulator.

The **Contact conditions** can help specify transition between different phases of a task. The events can also help determine if the objects starts slipping from the grip. The contact events can separate different phases of the overall grip, for example before contact one strategy may be used, and after object contact, the controller might have to switch to another strategy. Initial contact events can typically be determined by sudden pressure on contact sensors, cessation of motion from motor-torques, finger tip sensors, or vibration from the impact.

2.3 Other object detection methods

Structured light (Figure 6, left) paints a light pattern on the scene, and cameras pick up the resulting pattern. One or more cameras observe the scene from slightly different angles, and through analysis the curvature of the light pattern will describe the 3D scene. The precision and resolution of the scene is dependant on the width of the light strips and their optical quality. For extreme cases when detecting very small objects it will depend on the wavelength of the light used, but more relevant drawbacks are typically camera and display resolution.

Time-of-flight uses laser light to probe or *pick* a point in the scene, and measures the time before the reflection is returned to a sensor. Since the speed of light is known, the distance to the pick can be determined by the formula $d = c*t/2$, where c is the speed of light and t the measured time. This method of 3D scanning depends on the accuracy of the sensor, since only a few picoseconds define a depth of a millimetre. Other limitations to this method is the surface to be scanned. The reflectiveness and diffusiveness can have an effect on the reflected laser beams. One application of this method is the LIDAR (Light Detection And Ranging) (Figure 6, right).

Triangulation also uses a laser or light beam on the surface, with a camera mounted at a slightly different angle. The camera looks for the location of the laser/light dot, and with simple geometry the distance to the dot can be determined.

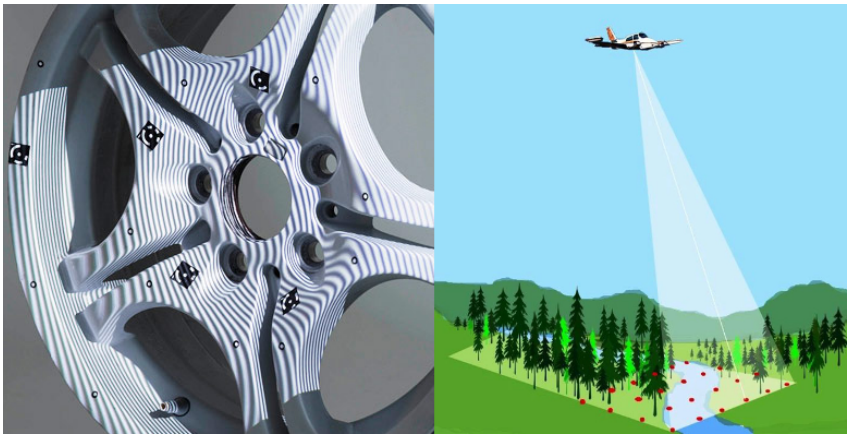


Figure 6: The left image shows structured light used for creating a 3D image from a wheel-hub. A typical application of LIDAR is aerial scanning of terrain to make height-maps (right).

2.4 Gripper Technology

The process of gripping an object can roughly be divided into two separate types of tasks: Pre-programmed and adaptive tasks. A pre-programmed task is where the gripper executes a precise and known series of movements and actions. This type of gripping is most relevant for factories and industrial processes, where for example a conveyor belt will contain objects that are positioned at a precisely given location. In this case the feedback of the gripping process is less relevant (although it should not be neglected) than the second case for gripping. In the second case, the object to be gripped is either unknown or the size or position of it is. To manage such objects, an adaptive gripping system is needed. For the robot to be able to pick up this object, it is not enough with a preprogrammed set of specific actions - it will need to detect the object and use a robust feedback system to be able to hold on to it.

In their work, Jørgensen and Petersen (Petersen, [9]) used a 2D image to calculate a rough bounding volume around the object to be picked up. They made a simulation

engine consisting of 3 different physics engines, ODE[2], Bullet[1] and RWPhysics, and created their own abstracted software layer. Their system used a 3-fingered SCHUNK Dextrous Hand mounted on a SCHUNK Lightweight Arm. The calculated bounding volume of the object gave an indication of where to apply the grip. The image was provided by an AVT Guppy Firewire camera, and the control of the robot was done with a Nintendo Wii controller. The purpose of their work was to pick up a random unknown object placed on a table, and hand it to a spectator. For this to work, they came up with some ideas that were considered reasonable for an optimal grip.

- Grasp the object close to table to achieve larger vertical grip region
- Attempt to adjust ϕ (the angle of the fingers) to be orthogonal to the 2D boundary of the object
- Grasp force and torque should be low to avoid slippage.
- Make all 3 fingers touch the bounding box at the same time

A couple of the points in this list should be clarified. When contact is detected, the fingers are rotated slightly to potentially get more contact points on the same finger. This slight change in angle is the adjustment of ϕ. The argument for lower grasp force to reduce slipping is perhaps not immediately clear. Normally, the harder one grasps an object, the less likely it is to slip. However, if the grasp force is not perpendicular to the surface of the object, a higher force makes it more likely to slip along this surface after a certain attack angle. They also found that if the contact point is deep in the hand (closer to the palm, and further from the table surface) an attempt at closing the fingers inwards at the bottom (a grasping motion) would allow more lifting with less force on the object.

Their experiment was based on grasping an unknown object from a 3D database. This was simulated in a physics environment, and tested. Their algorithm resulted in a stable grasp in about one out of 3 attempts. To improve on this, they discussed that a better image processing would result in a more precise bounding volume. This combined with better information about the 3D simulation, especially at the contact points, would improve the gripping algorithm.

Allen and Bajcsy (1984 [5]) described an interesting idea, detecting objects by both touch and vision. The tactile sensing provided information in areas that were occluded from the stereoscopic cameras, and touch sensing was also used to describe the surface of the object better. This setup used a "finger" with a tactile sensor, attached to a robotic arm. This sensor was moved to the position of an object in an attempt to trace the surface, giving a better 3D description of the object than what the vision alone can accomplish. The objects detected were then matched against candidate objects in an internal database, and attempted verified or discarded. Their work was not complete, but in a follow up paper (Allen 1995 [4]) more experiments were done with the same setup. The results showed that detection by vision alone is noisier, uses more bandwidth and gives more errors (especially due to lighting conditions) than touch. The surface material can also have a large impact on optical detection due to reflectiveness, transparency and

colour. Touch sensing is generally less noisy, provides less data (and thus consumes less bandwidth), has more degrees of freedom than the cameras, and can give a more precise shape detection. The largest problem with touch sensing is that it requires a guided or clever adapting system to control the touch sensor for meaningful results. The combination of the two sensor types provided a more robust object detection than single sensor systems.

The project by Petersen and Jørgensen (2008 [9]) is especially relevant for this project since they used the same gripper hardware, the SDH. Their approaches for an optimal grip are very relevant, with the exception that they have some predefined boundaries found with optical detection. This project does not currently have this, but by building an expandable control system it is easy to incorporate more object detection features. These can then assist in the positioning of the gripper and the gripping. One example of this is the use of structured light, described in section 2.3. This method would create a 3D image of the target object, which could be analysed for boundaries and an optimal gripping approach.

Figure 7: SCHUNK Dextrous Hand

3 Solution Outline

This chapter will focus on the goals for the project and how the gripping should be performed. It will look at the entire system, consisting of both gripping module, manipulator arm, their control systems and the interaction between them. The gripper module will be explained and the relevant variables and coordinate systems to be used later will be defined. Possible expansion and potential use is also considered.

3.1 SCHUNK Dextrous Hand (SDH)

This project will focus on working with the SCHUNK Dextrous Hand (SDH), seen in figure 7. In order to work effectively with the hardware and avoid future confusion, it is useful to define some terms to be used for the hand. These terms are illustrated in figure 8. The SDH is a three-fingered robot with two tactile sensors on the inside of each finger. Two of the fingers can rotate relative to the base of the hand (the palm), and all three fingers have two joints: A proximal and a distal. This totals to 7 degrees of freedom, all of which should be controlled real-time. On the inside of each finger there are two rubbery pads. These matrices with contact sensors (produced by Weiss Robotics) can measure the force applied to them. The movement of the fingers can be controlled both directly with positions and with velocities. A controlled gentle grip will have to rely on both of these in order to make an effective grip. The terms are illustrated in figure 8. The largest piece of the gripper is the **base** or **palm** of the hand. All three fingers are attached to this, and it has a universal connector to be fitted on to several robotic manipulators. It also contains the hardware controller for the gripper. Each of the six mobile parts of the gripper are defined as a **links**, and the connections between them as the **joints**. When

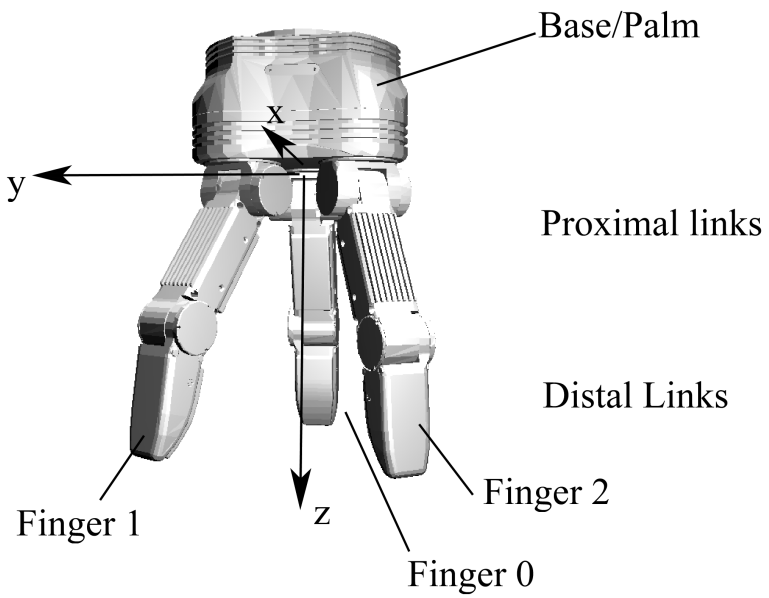


Figure 8: Terms used to define the parts of the SDH gripper

the gripping takes place, the base of the hand will be pointing its connector sky-ward and the fingers towards the ground or table. The non-rotating finger is denoted as the **thumb** or **finger 0**, and in counter clockwise direction seen from above, **finger 1** and **finger 2**. The links closest to the palm are denoted as **proximal** and the ones at the end of the fingers as **distal**. Since the gripper will be communicating with an overall control system, it needs to be able to specify directions in a manner that is understandable. A coordinate system is thus defined for the gripper. This coordinate system will always be fixed to the palm of the gripper, with the intersection of the three axis in the centre of the palm. The **x-axis** is pointing out between the two rotating fingers (the thumb "opens" in a negative-x direction). The **y-axis** will point in the opening direction of finger 1, and **z-axis** will point down towards the ground/table. The z-axis will also be referred to as the "palm-axis".

3.2 A rough gripping solution

The intended purpose of this project is to attach the SDH to a robotic arm, and have them communicate via a control system. If something is beyond the reach of the gripper, the control system must be such that it will move the hand to an improved position, allowing the gripper to hold on to the target. Both pieces of hardware have their separate control systems, and the overall control system must create the needed communication channels between them. The layout of the control systems and their communication channels are illustrated in figure 9.

The overall control system will need to initialise the other two, and before anything else happens, set them in a waiting state. One of two things can then happen: it can determine that the gripper is in a good place to start gripping, or it can decide that the gripper needs to move. Initially the only available tool to determine whether there is

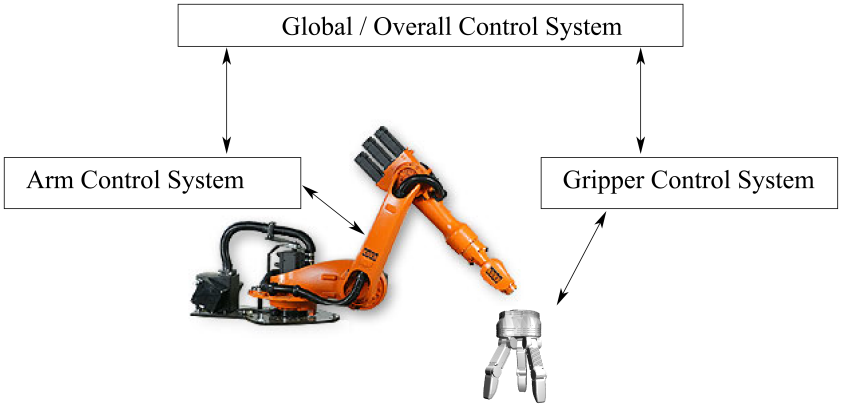


Figure 9: Illustrates the setup of the overall system, the controllers and their communication channels.

anything to pick up or not is the gripper. However by creating an open and expandable global control system, later editions can easily include other detection methods such as cameras, radars and lasers. Without that (and without human intervention) the control system could create a search pattern. For example, if there is a given work area it could start in one corner, and make a grasp. If nothing is found, the gripper control will return a signal, and the overall control will move to a new position. Consider the system seen from above: When the overall control system moves the hand, it needs to move it an offset equal to the maximum grasp radius of the gripper. Before the arm commences the movement, it needs to first reset the gripper to an "open" configuration, and after this put it in a waiting mode. When the movement is deemed complete, it can re-initialise the gripping algorithm, and wait for the results.

It should be clarified that the discussed search by grasping is not a very realistic situation. It is not especially effective, and in most situations the target object will either be placed at a more or less known location, or there will be other means for achieving a rough location estimate. It should be considered more of a possible addendum to the intended capabilities, namely grasping objects, and reporting the need for a better location. If the overall system were to have another sensor attached at a later time, this grasp-and-search approach could easily be exchanged for a much more effective method.

The gripping process is designed in such a way that the movement of the arm is abstracted out of the control system for the gripper. If the gripper calculates the position of the object to be such that it will not be able to successfully hold on to it, it will calculate an offset needed for a better grasp. This offset is returned to the overall control system, and after the move is complete, a new search-and-grasp is initialised. The gripping control does not need to know that it is in a more optimal position than before, it runs through

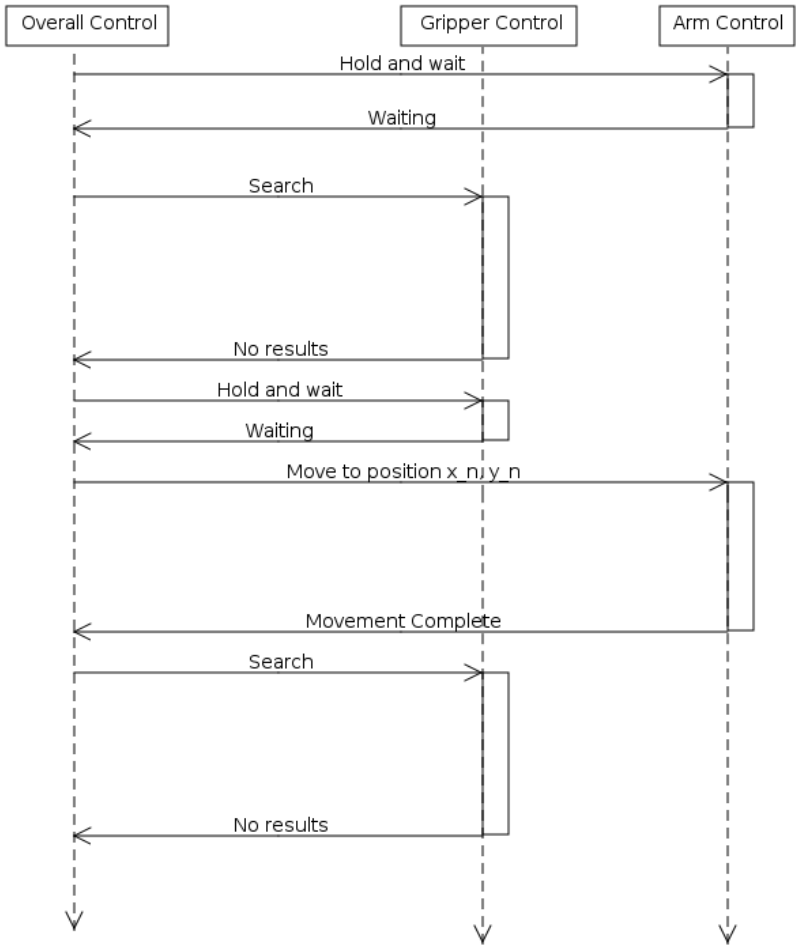


Figure 10: This sequence diagram shows an example of the search-and-move process and the messages sent between the control systems.

the same procedure as a normal search; detect, estimate object position, determine if it is within grasp, grasp if it is, and return an offset if not.

Figures 10 and 11 show examples of the gripping scenarios and the communication between the gripper control and the parent overall control system. In the first sequence diagram the gripper does not find anything, returns this message to the overall control and waits for further instructions. In figure 11 an object is detected. However it is too far from the centre of the gripper to be able to hold on to it. From the contact position it estimates a needed translation to get a better position on the object. This is the scenario that will be focused more on in the rest of this paper.

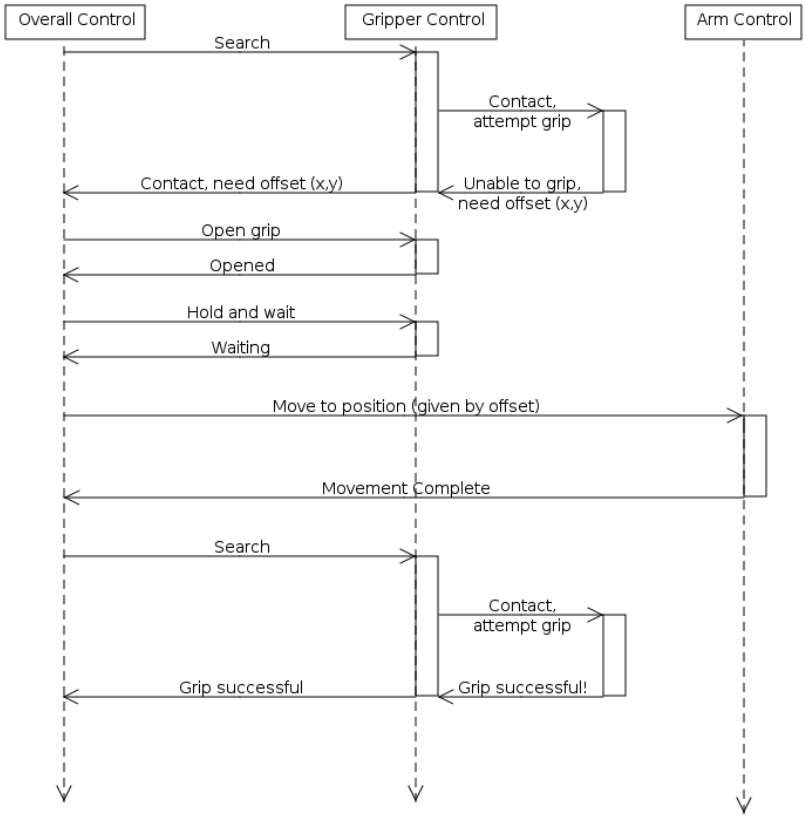


Figure 11: The sequence of events leading up to a successful grasp. When contact is detected in the gripper, it will attempt to determine if it is able to grasp the object. If it calculates the object to be too far off for a successful grip, it will return with a request to move in the direction required for a better grip. After the overall control is done moving the arm, the same iterative process is started.

4 Design Outline

This chapter will focus on the parts of the project that will be implemented, the gripper control. The solution will be designed with the overall controller still in mind, but with the functions abstracted. This way a future project or continuation of this work will easily be able to pick up where this one ends, and implement the communication to an overall controller system.

4.1 Goals for the project

This project aims to implement one part of the previously mentioned system, the gripping control. The communication protocols will be implemented, but abstracted in such a way that one can manually read and write requests and also manually move the hand/object. The gripper control will pick up objects, holding firmly, but gently enough to not destroy the object in question. The first step in this process is to somehow detect the objects location, size and shape. A more advanced system could possibly determine what type of object this is (glass or steel?) by comparing a detected object to a database. This database could possibly hold data about the material of the object, and provide information about durability, weight and texture. This project will use detection based only on tactile sensing, using the SCHUNK Dextrous Hand. The challenge will be creating a system that is versatile enough to pick up fragile objects, in this case chocolate eggs, and holding them firmly without dropping or crushing them. The gripping mechanism relies on feedback from the sensors.

Given that the objects to be picked up are not in a specified position there is another problem that is not trivial. There is need for a two-way communication system between the gripper and the manipulator arm (or rather their control systems). In case the gripper does not find anything within its reach, the overall control will need to instruct the arm to move, perhaps in some sort of pattern, to try a new area. If there is another sensor present, for example a camera, this could also be used to give a rough estimate of the location of the target object. Even in the case that it does detect something, the object might be too far from the centre of the gripper to get a good grasp on, and again the controller for the gripper must instruct the arm to move in a given vector (relative to the hand). The communication from the arm to the hand will be simpler, and consist generally of messages telling the gripper controller that it is moving, stationary, in a new position, and similar. Since this project is not directly concerning the control of the manipulator arm, it might be sensible to abstract these actions, and handle the implementation at a later time, perhaps with an overall controller, able to operate both the arm and the gripper.

4.2 An adaptive gripping solution

The suggested objects to be gripped are chocolate eggs, and if these are approximated to a sphere, a slightly crude method of gripping the object can be devised. Assume that there is an upper limit for the size of the egg, and that a touch by the gripper does not move the egg. If one can determine in a 3D space exactly where the contacts happen, they can

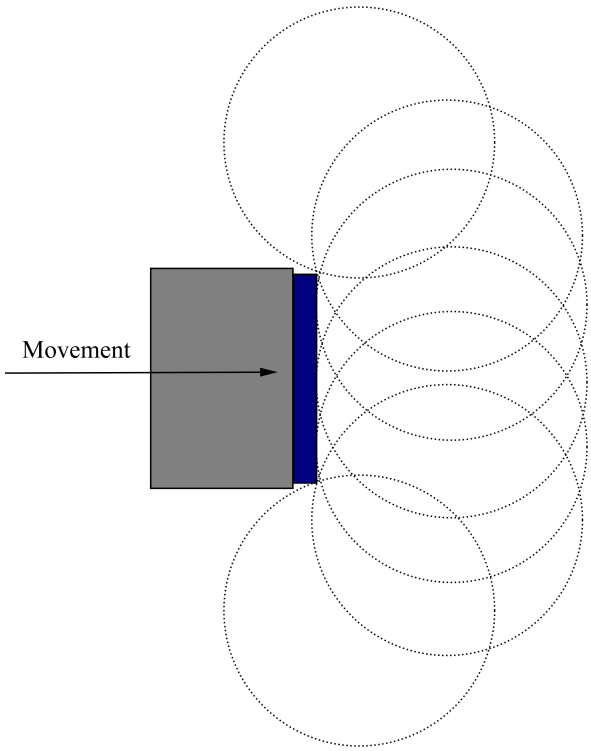


Figure 12: A finger (grey) with its contact pad (blue) seen from a distal position, and the possible locations of the egg based on the contact point.

be aggregated into a 3D image of the egg. Assume the gripper makes a motion where the movement of a pressure pad is always perpendicular to the surface of the pad (Figure 12). Furthermore, assume that the egg is always standing up, with its narrower end pointing up towards the palm of the gripper. If a closing motion of the fingers produce a contact event, the position of the contact relative to the width of the finger gives some insight about the position of the egg. This simplification can be used in another dimension as well. Consider a simpler gripper with vertical fingers with contact pads pointing in towards the centre. The point of contact in the vertical direction can help determine the position of the egg, since either the widest part of the egg will contact, or the end of the finger will touch first (see figure 13). In both the cases, the only complicated scenario is when the contact happens at the very edge of the contact pad. This means that the egg can be in any position pivoting this edge. The simplest solution to this would be to move the finger (either by gripper control or by manipulator arm control) in the direction of the contact, relative to the finger, and attempt to get a more central contact.

A simple approach to the first step in detecting the location of the egg can be to have the fingers straightened out while grasping (the distal angle on all fingers = 0), and this way a simple transformation from spherical into Cartesian coordinates can determine the position of the contact, relative to the palm or a desired position. Since the egg is of a fragile nature (both real eggs and chocolate eggs), some care must be taken in not relying

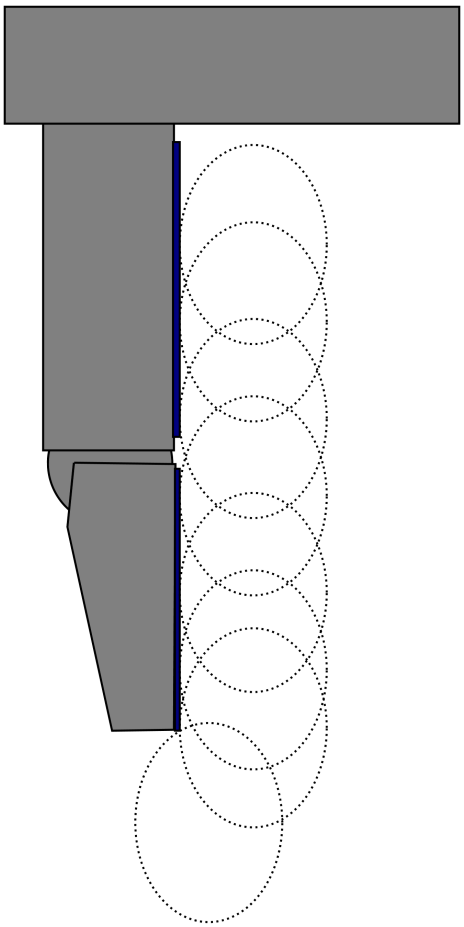


Figure 13: A side view of a finger, stretched out. Again, given the contact point in a vertical direction, a rough estimate of the location of the egg can be made.

too much on grip pressure to maintain the grip. One possible solution to this could be to let the proximal links do the gripping, and then curl the distal links around the bottom of the egg to hold it up instead of squeezing it. This would rely on the position of the egg being relative close to the palm, and the egg not being too large. If the egg is positioned directly on the table (or another flat surface) care must be taken to not collide the distal links with the table.

Images 12 and 13 show a finger, from the bottom and from the side respectively. The dotted lines are the possible positions of an egg, given that there is contact detected on the finger. The idea is to use the symmetry of the egg to estimate the position based only on one contact point. Unless this contact happens on the edge of the pressure pad, the pressure pad will always be a tangent to the egg surface. If the contact does happen on the edge of the pad, the possible locations are many more, and another approach is needed to determine the location of the egg exactly.

The entire control system is designed to work around the principle of a control feed-

back to the controller for the robot manipulator. If this is abstracted, it can easily be expanded and improved in the future to provide support for different detection mechanisms from the grippers perspective, as well as auxiliary systems from the manipulator. These two directions for the information flow are best explained by an example. In the first case, assume that an optical detection device is equipped, with image recognition algorithms and feedback to the manipulator control. This device will then work parallel to the gripper, feeding the control system with information and updates regarding the location of the egg relative to the hand. In the second example, assume a future version of the SDH gripper comes with a radar or sonar, and can find the direction of the egg relative to itself. This information can then again be passed up to the manipulator control system, to move closer to the target.

The control of the gripper itself is the main task of this paper, and as such an effective and safe (both for the egg and the hardware) method of locating and picking up the egg must be made. This algorithm will be divided into separate steps, and attempted separated to simplify future implementation;

1. Reset the hand to open position
2. Wait for ready signal from manipulator controller
3. Slowly close the fingers, while checking the contact sensors
4. If no contact is made after the proximal links reach an angle α, send signal to overall control that no positive contact was found. Return to point 1
5. If there is contact in the grasp, send immediate stop to actuators
6. From the location of the contact point, attempt to calculate the centroid of the egg.
7. If the centroid is within a previously defined area (close enough for a good grasp), begin grasping motion.
8. Else, reset hand to open position, send signal to manipulator control to move hand in a vector given by [**desired centroid position**] - [**calculated centroid position**].
9. For the grasping, keep the distal joint at an angle of 0 deg, while slowly closing in the proximal links. When contact is made by one of the fingers, this should immediately stop, and the other two carry on until they also have contact.
10. When contact is made by a finger, the contact point should optimally be close to the centre of the contact pad of the proximal link. If it is very close to the edge, attempt to rotate this finger gently in order to improve the contact position.
11. When all three proximal links are in contact with the egg, begin to close the distal links, until each one has contact. When one distal contact is made, it stops and tells the gripper control system it is done, waiting for the others to complete.

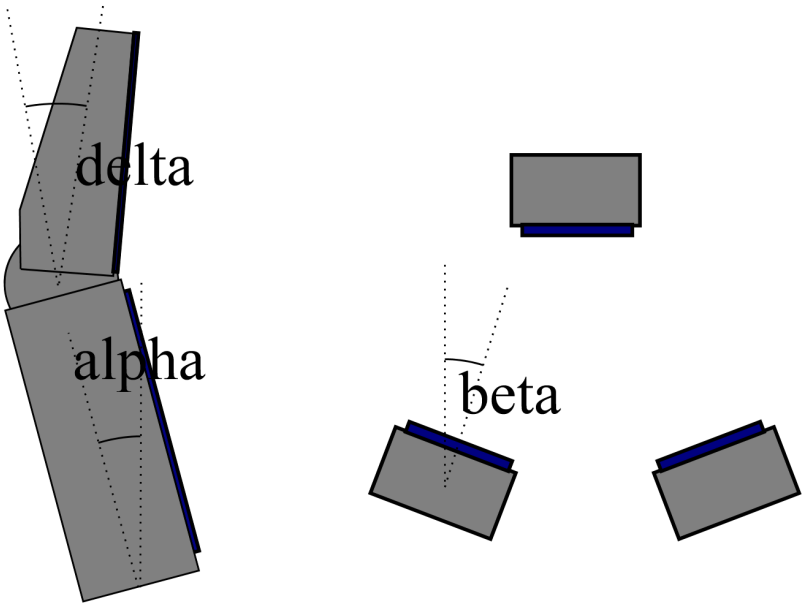


Figure 14: Left diagram shows proximal and distal α angles for a finger, right diagram shows angle β

This algorithm is also illustrated in the state machine in figure 15. From this algorithm it is apparent that some more terms need definitions. The **open position** of the hand is defined as proximal links outstretched (gripper is as open as possible). This way, the grasping algorithm can cover a large volume, even if it is unable to grip objects in all of this volume, a detection will help improve the position. In this open position, the values of the proximal joint angles are $-\pi/2$.

The angle $\alpha_{[0..2]}$ of a link is defined such that in upright position $\alpha = 0$, positive angle is closing in towards the palm, and negative angle is opening outwards. The subtext term defines the finger. The limits for α is $\pm 2\pi$. The two fingers that can rotate in the palm (finger 1 and 2) are mechanically coupled, and as such only one variable is needed to define the rotation; β. Rotation in towards the palm is defined as positive, and $\beta = 0$ is defined as the rotation needed to place the pads of finger 1 and 2 pointing in direct opposite direction of finger 0. Given the maximum size of an egg, the angle α_{max} can be determined. Beyond this angle no egg will fit inside the grip, and it is therefore not necessary to close the grip further. This grip will be tight enough to detect any egg with the centroid in the boundaries of the hand. A failed detection after all three proximal angles have reached α_{max} means there is no egg in the reachable area, and the manipulator arm must be instructed to move to a new position. Note that this α_{max} should only be used if there is no detection. If one of the fingers make contact, the others will continue closing in, even beyond α_{max} to attempt to reach the same target. This however, only applies if the egg is assumed to be in a reachable area. If for example the contact is made very far from the palm (with a proximal angle close to $-\pi/2$ it is definitely not reachable, and the program goes to point 8 of the list. The angles are shown in figure 14. Figure 16

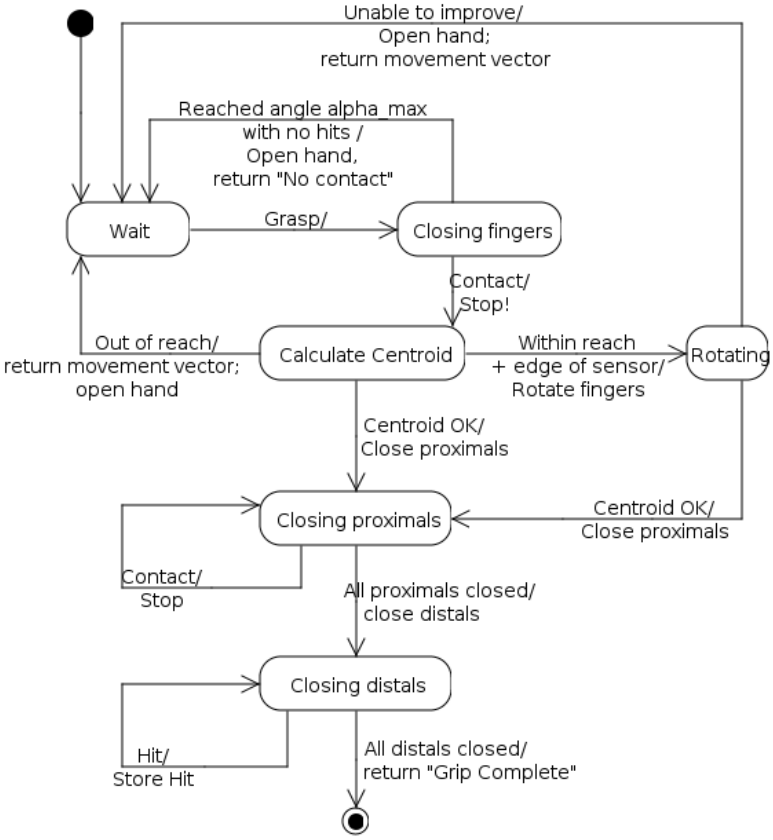


Figure 15: This state diagram illustrates the algorithm for gripping unknown objects

shows a configuration of the hand, where it is not possible for an object to fit in between the fingers. If the fingers reach such an angle, there is no need to continue closing them further. Alternatively, α_{max} could be such that the tips of the fingers just about touch.

An optimal position for the centroid of the egg can be defined (given minimum and maximum boundaries) such that the egg will be gripped from the side by the proximal links and held underneath by the distal links. Caution must be taken by the manipulator arm control to not lower the gripper so low that the palm of the hand can collide with the egg. If a contact is detected too close to the palm or too far out (only the distal links can grip), instructions must be sent to the manipulator control to move the hand into a more optimal height. The desired position of the egg referenced in point **8** of the list, is a point in between the fingers where the largest egg will not collide with the palm, centred in between all three fingers, and at such a vertical position that only the proximal links will grip it.

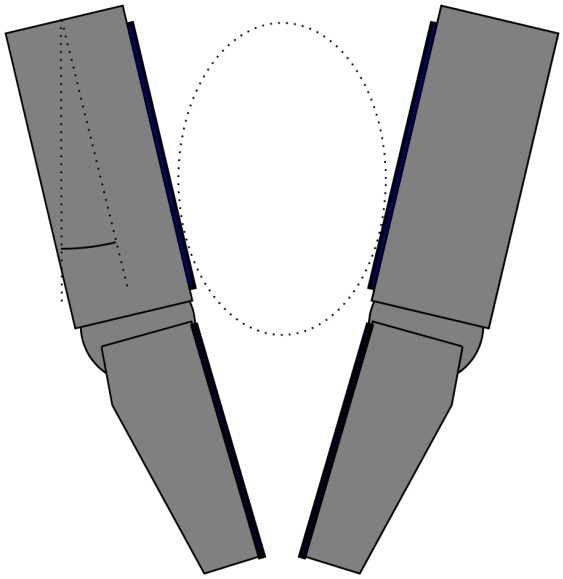


Figure 16: Given the minimum size of an egg, after a certain α there can be no egg in between the fingers

5 Implementation

This section will cover the implementation of the algorithm designed in the previous chapter. Some requirements in the application are discussed. This chapter will not include source code for the implementation, please see Appendix A for attached source. During the implementation of the algorithm some changes were made. These changes and their reasoning will be made clear in this chapter.

The SDH comes shipped with libraries for connection to the hardware in C++ and Python. Since previous work (Monteiro 2009 [10]) was done with C++ and .NET, the same programming language and environment will be used to implement the gripping algorithm. Visual Studio and .NET provide a simple and intuitive interface for creating a graphical Windows User Interface. For this project, not much is needed besides a few buttons to trigger actions, and an output window to read system messages. In addition to this, it will be writing data to a file for later analysis of the forces.

From the list given in chapter 4.2 one could implement a state machine, typically a large switch-case statement. However, this is not really necessary for this system. The states are mostly sequential, and the transitions between them are rather simple. The gripper alternates between waiting, searching, grasping and maintaining grasp. From the "wait" state, it will always move on to the "search" state, and from the search state there are only two possible new states, "wait" and "grasp". This might be simpler to implement and be more efficient performance wise.

One very important aspect of this system is that it needs to be able to detect collision and call for a stop very fast. Therefore it is likely wise to keep the contact detection in a separate thread that can poll the hardware at as high rate as possible. The likely bottleneck here will be the bandwidth to the hardware, or the hardware itself. This implies that the searching fingers should not move at maximum speed. By moving them gently enough, the largest delay in the contact-sensing thread will have time to call for a full stop before the finger has moved the egg too much.

In the algorithm for detecting and gripping objects one point that was brought up was determining the centroid of the egg relative to the contact position. This is not quite trivial because the position of the egg is not always easy to determine based only on one contact point. At first glance what seems to be a rather simple solution is to define the surface of the pad as a tangent to the egg. However, given the shape of the egg, and that the finger might not approach the centre, this is not true. It is possible that the contact will happen at the edge of the pad (see figure 12). If this occurs, the finger must be rotated, if possible while still maintaining contact with the egg. This is done to see if a better position can be achieved. By better position it is meant a position where the contact will move away from the edge of the pad towards the centre. When this has been achieved, it is more correct to say that the pad will be a tangent to the egg, and the gripping can continue. This of course does not work if the contact is done by the thumb (finger 0) that cannot rotate, or even if the contact is made far from the centre of the palm. A change in the angle β will then cause more of a translation than a

rotation for the pads. If this is the case, the only solution is to send a message back to the manipulator control system to move the gripper to a better position.

5.1 The application

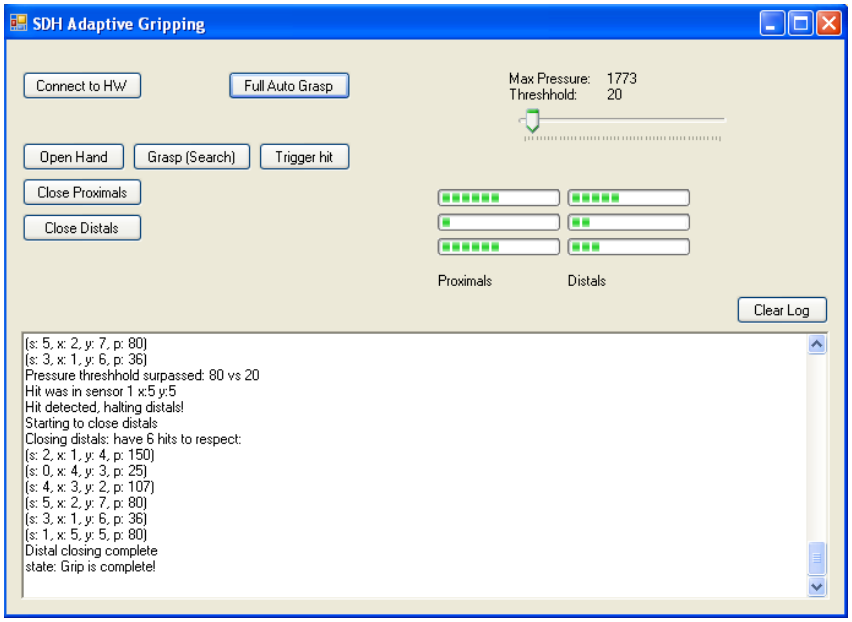


Figure 17: GUI for the application. The buttons in the interface are a way to simulate the communication between the controller for the gripper and the overall controller for the entire system (gripper and arm). The progress bars show the maximum pressure detected for each pressure sensor.

Figure 17 shows the application created in Microsoft Visual C++ 2010. It has only a set of simple controls that mimic the communication between the overall control system and the gripper control system. This application instances an object containing all the main parts of the gripping algorithm. However, in order to not lock down the user interface, almost all the methods are started as separate threads. The first thing the user must do after initialising the application is connect it to the hardware via the button **Connect to HW**. If the SDH is not connected or not powered on, an error message will be displayed in the log window. Once it is connected to the hardware, it will immediately start checking the sensors on the hardware. It cycles through each of the 6 sensor pads, and for each of those pads it checks every pressure point. In this phase (the *Wait* state) it does not do anything with these sensor read-outs. However, it will still display the maximum pressure detected (seen above the Threshold slider in figure 17). When the **Grasp (Search)** button is hit, the system moves to the next state, *Grasp*. Now it will begin to test the max pressure read from the sensors. It will then compare this against the threshold set by the user via the slider in the user interface. By default the threshold value is 20. Whenever a readout exceeds the set threshold, an object of the type *SensorHit* is created and added to a global collection. In the grasp state, all movement is halted, and an analysis of the sensor hit is done. The SensorHit object contains infor-

mation about which sensor was hit, where on the sensor and how much pressure was put on the sensor. This is used in the next stage of the algorithm, where the centroid of the target is estimated. The centroid calculation can result in three possible outcomes: 1 - the egg is within the reach of the hand, in which case the system can proceed with the next phase, *close grasp*. 2 - The egg is within reach, but the contact occurred on the side of a sensor (the x position is either close to 0 or close to the max-x for that sensor). In this case the system will move to the *rotate* state. 3 - The target object is estimated to be out of reach. From the estimated position, a vector is created that is the position of the centroid - the position of the optimal placement of the egg. This vector is relative to an absolute coordinate system for the hand, and returned up to the overall control system.

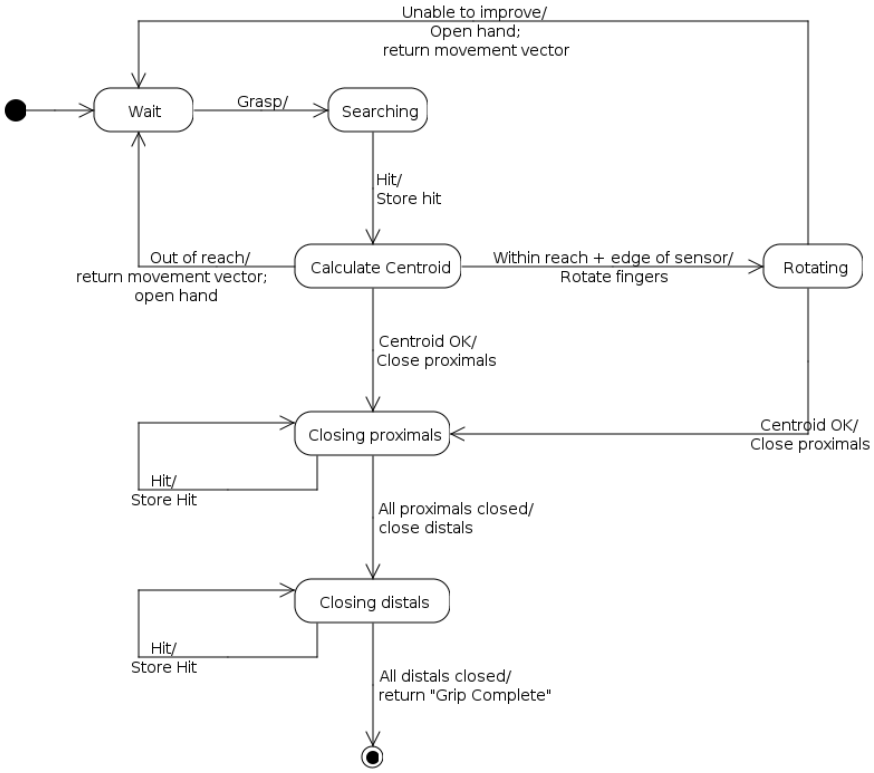


Figure 18: State diagram for the gripping algorithm

The *close grasp* part of the algorithm focuses on bringing together the fingers, but only the ones that have not collided with the object yet. This is done by analysing the SensorHit objects stored globally. When the command to close the fingers is called, it loops through all the recorded hits (these are obviously reset when a "new" grasp is made), and checks if there have been any hits that correspond to the axis it is closing. For example, in the SDH library, axis 1 corresponds to the finger with sensors 0 and 1 on. If any hits on these sensors exists in the collection, a flag is raised, inhibiting this axis from approaching its target. For each hit, the closing motion is halted, and it returns how many axes have been inhibited. By analysing this return value, the system can determine if it has closed all the fingers as much as it can, or if there is still an axis it can close in

on the target to get a better grip. In the implementation the closing motion is divided into the proximal links and the distal links. First it will attempt to close in the proximal links as much as possible. A proximal link is inhibited both by a hit on the proximal or the distal sensor on the same finger. When the closeProximals method reports that it is unable to close the proximal links further, it is time to close the distal links. Depending on if the hits inhibiting the proximal links were on the proximal or distal sensors, the distal links will now also close in on a target angle. The algorithm is very similar to the one for the proximals: if a hit is detected, it is recorded in the global collection, and the closing motion is halted. When the motion is recommenced, it checks the collection for hits on **the distal sensors** this time, inhibiting the corresponding links. When there are no links that are uninhibited, the grasping motion is complete, and a message is sent to the overall controller telling it the grasp is successful. Figure 18 shows the steps in the algorithm before the grip is complete.

The application is based on starting separate threads for several different tasks. For example, having a CPU-intensive loop in the same thread as the user interface is undesirable since it makes for a sluggish and unresponsive experience for the operator. Instead the user interface initialises a thread via a button. As an example, when the operator presses the "Connect to HW" button, it starts, among other things, a loop that constantly polls the sensors in the fingers. One important feature with the SDH library is that most of the "move"-calls are *blocking*. For example after targets have been set for an axis or a finger, the method *MoveHand()* or *MoveFinger()* has to be called. This initialises the actual movement of the fingers. The problem is that the application waits at this line in the code until the movement is complete. In these cases it is especially relevant to use multi threading. When a call is made to move one or more axis, it is not made directly to the hardware. Instead, it creates a new thread that runs a specified method. While this thread is blocked by its hardware call, another thread is checking the sensors constantly, and if a pressure surpassing the threshold is detected, it makes a call to stop the hardware and abort the *move*-thread immediately.

5.2 Improving the off-centre handling

During the implementation of the part of the algorithm that calculates and evaluates the centroid, a better solution was found. Instead of assuming the position and size of an object, based on one contact point and pre-defined values, why not let the gripper just try gripping it anyway? Figure 18 (p. 25) shows the state diagram for the initially planned algorithm from chapter 4.2. Point 6 indicates that the results of the estimation of the centroid should decide if the gripping continues or is aborted. However in the new algorithm it will now instead jump from point 5 to 9 (Figure 18, p. 25): When a contact is made, it will halt the finger the contact is detected on, and keep the other two still going, approaching their pre set target. Again, if one of those two fingers hit something, it will stop, and the last un-affected finger will continue in towards the central palm-axis. When all fingers have stopped, meaning each finger has either hit something or reached the central axis, the algorithm now checks how many fingers have had hits on them. If this test returns only one finger, the grip is not complete since it will not be able to grasp with only one finger. This new algorithm is shown in figure 19.

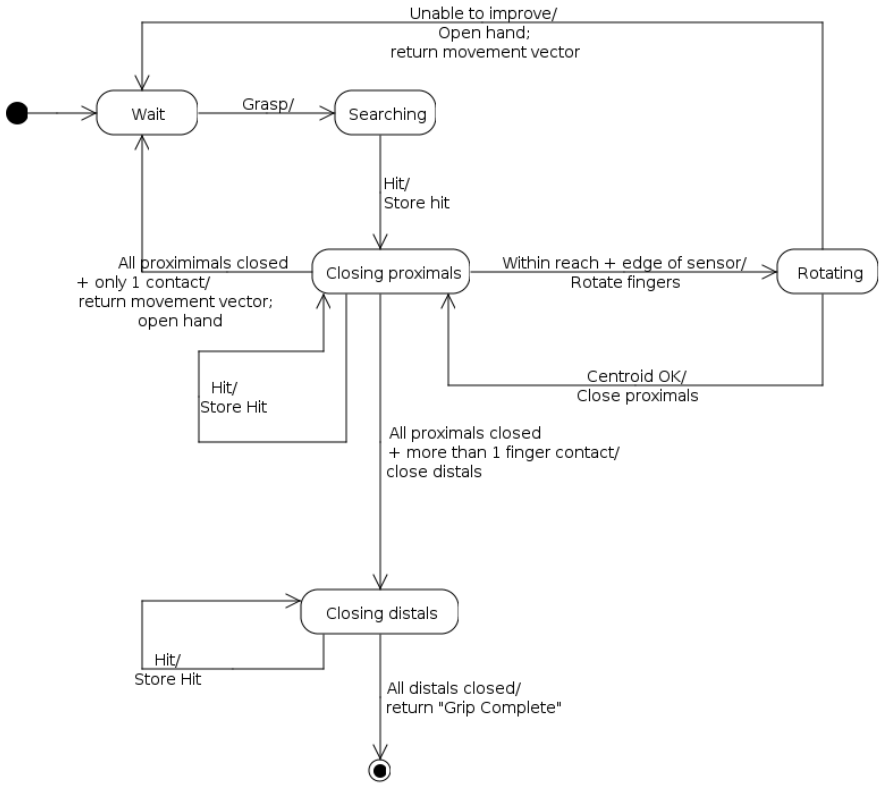


Figure 19: New state diagram with improved handling of off-centred objects

Notice how the state "Calculate centroid" from the state machine in figure 18 has been removed, and instead another output from the "Closing proximals" state has been added. In practice what this algorithm does, is test if any object intersects either the paths of one or more fingers, or the palm-axis of the gripper. Figure 20 illustrates the idea of gripping an object that might still be off centre relative to the palm axis (red beam). Since the blue ball intersects both the path of finger 1 and the palm axis, the fingers will close in on it and be able to hold on to it nicely.

When the algorithm returns an "out of reach" message due to only one finger making contact, it will use the position of the contact as a new target for the gripper. Notice how this again differs from the original algorithm: Instead of assuming a radius on a spherical object, it only uses the contact point of the object. The x and y position of the hit are calculated by using a spherical to Cartesian coordinate transformation. The radius of the sphere in the spherical coordinate system is calculated by evaluating which sensor was hit, proximal or distal. Furthermore it uses the y-position on this sensor to see how far in a distal direction this hit occurred. The distance along the finger will then be a value calculated according to the sensor hit position with some constant offsets added. Since this is all occurring in the *close proximals* process, there is no need to consider the angle of the distal links. The x and y positions relative to the palm axis are calculated. The variable *dist* is the position of the hit along the finger. Figure 21 shows the spherical

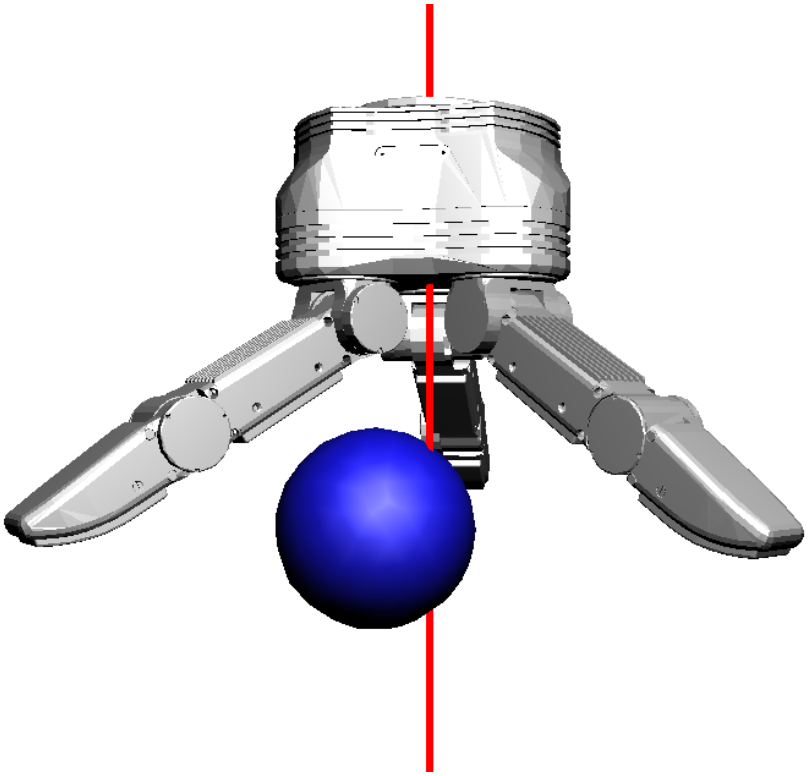


Figure 20: An off-centre object can still be gripped successfully.

coordinates on the gripper. Note how this will find the offset from the base of the finger, not the centre of the palm. Therefore an offset is applied to the x- and y values to compensate for the distance from the centre of the palm to the base of the finger. Figure 22 shows the gripper module from a bottom view. All three fingers are positioned the same distance from the centre of the palm, and this distance is the *offset*.

The following equations give the transformation from spherical to Cartesian coordinates. Recall that α is the angle of the proximal links, and $\alpha = 0$ when the finger is pointing straight down towards the table, and $-\pi/2$ when fully open. β is the rotation of the fingers, where $\beta = 0$ is when finger 0 is facing in the opposite direction of finger 1 and 2. The *offset* value is distance from the centre of the palm along the x or y axis to the position of the finger.

For finger 0:

$$x = -dist \cdot sin(\alpha) - offset \tag{1}$$
$$y = 0 \tag{2}$$

For finger 1:

$$x = dist \cdot cos(90 + \alpha) \cdot cos(\beta) + offset \tag{3}$$
$$y = dist \cdot cos(90 + alpha) \cdot sin(\beta) + offset \tag{4}$$

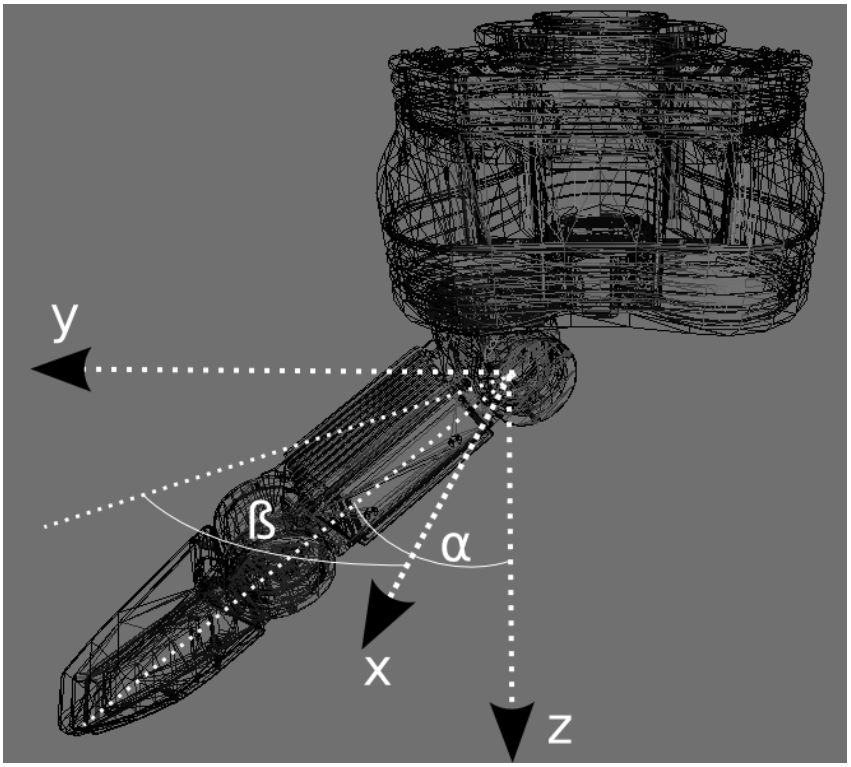


Figure 21: Description of the spherical coordinates.

For finger 2:

$$x = dist \cdot cos(90 + \alpha) \cdot cos(\beta) + offset \tag{5}$$

$$y = -dist \cdot cos(90 + alpha) \cdot sin(\beta) - offset \tag{6}$$

Since this calculation will position the gripper on top of the contact point and not necessarily the centroid of the object, this might result in the new position of the gripper being slightly off-positioned to the object. It should still be enough to get a grip on it, since now at least some part of the body of the object will intersect the grippers palm-axis.

5.3 Rotating fingers for better position

It became apparent that using the initial idea of rotating the fingers while in contact with the object would not work. This idea was based on a rotation around an axis that would lie along the length of the finger. In this case, the surface of the sensor pads could be rotated while not moved at all. Since this rotating axis will only lie along the length of the finger when the proximal link is close to 0 degrees, it is not very useful. If the proximal link is at 0 degrees, the finger is very close to the centre of the hand, and an egg would not be able to fit in there anyway. Another solution was therefore devised. Instead of relying on a constant contact with an object, it could just rotate an amount that would be dependent on the distance of the hit, measured from the palm. The closer

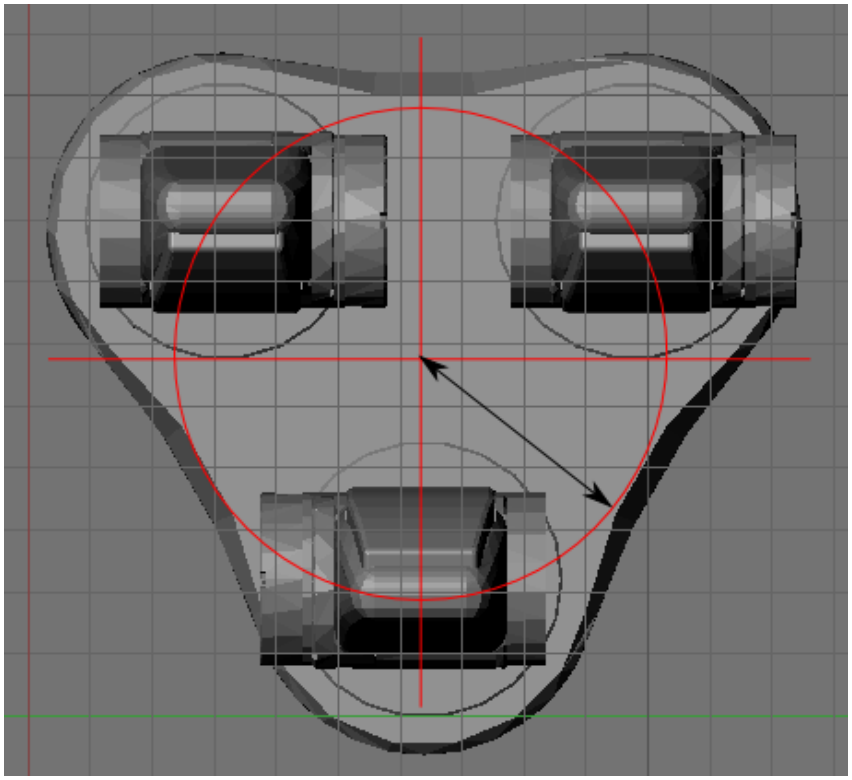


Figure 22: All three fingers are positioned symmetrically around the centre of the palm.

to the palm, the more rotation is needed, and as such a simple function was created to apply finger rotation:

$$rot = 10.0 - hitDistance \cdot 10.0/160.0 \tag{7}$$

Hit distance is the measured distance from the palm to the point of contact along a finger. This was tested, but it brought forth another problem: By first contacting and then rotating the finger, the finger would pull the object with it. Especially when it was a lighter object, such as the chocolate egg. In some cases, when the egg was balanced on a foot, it even tumbled the egg down with the rotation. An extra step was applied to solve this: before rotating the finger, move out just a little bit to lose contact. After this it was clear to perform rotation without disturbing the object, and when the rotation was complete it could move in again.

6 Testing

Figure 23: The testing setup for the gripping. The gripper is suspended in the air by supporting beams across two containers.

6.1 The test setup

The control for the robot manipulator or the overall system is not present, and as such the testing scenario will attempt to isolate the parts of the gripping algorithm that can be tested. The SDH gripper module is mounted on a square wooden stand with fingers pointing upwards, but the gripper in this position will not be able to test lifting capabilities or picking up object from a flat surface. Therefore the SDH platform is held upside down between two platforms, suspended on crossing beams. The setup is shown in figure 23. Objects can be placed in different heights and positions relative to the hand, and the systems reaction can be measured. In the cases where the hand reports the need to move in order to get a better grip on the target, the corresponding movement will be done by moving the target object instead in the opposite direction. By using chequered paper or a ruler the object can be moved exactly the distance and direction requested by the SDH gripper. The primary target object is a chocolate egg, and the main objective is to see if the SDH is able to grip this without destroying or damaging it. However it could be interesting to test other objects as well, both spherical (an apple) and more irregularly shaped (cell phone, soda can). If the gripping algorithm damages any of these objects, it is likely the chocolate egg will not stand much chance.

6.2 Collecting data

The application is programmed to print data to file, as it is read from the sensors. Later analysis of the pressure data and the pressure distribution can help improve the gripping algorithm, and possibly identify problem areas where the grip is too tight and damages

the egg, or so weak it is unable to hold on. The main verification will be done visually, as it will be easy to see if the grip has damaged the egg.

6.3 Testing scenarios

To test the versatility of the gripper, some different scenarios will be set up and the results recorded. First, different objects will be tested, positioned at different heights, and relatively centred under the gripper. This will test if the gripper is able to maintain a grasp both by holding the object with the proximal and the distal links. The objects to be tested are an empty 33 cl. soda can, an apple and a chocolate egg. The soda can will be attempted gripped both standing up, and lying on its side. This will test the control algorithms ability to grip non-symmetric objects. These setups will conclude the tests where objects are positioned relatively centred in the gripper. The next set of tests will show if the gripper is able to determine if an object is graspable, and how far it requests the overall controller to move it. This can be verified by following the instructions given by the gripper controller, but since the gripper cannot move in this setup, the target object will instead be moved by hand in the opposite direction. This scenario will be tested with the upright soda can, the apple and the chocolate egg.

7 Experimental Results

This section will show some of the relevant results from the experiments. Included are plots of the max-pressure readout per sensor. It should be noted that the sensors are not temperature calibrated, and as such there is no linear conversion between the number read out and a physical force. Around 3450 seemed to be the maximum possible output for the sensors. The data from the sensors was automatically written to a file during the experiment. This was later read into MATLAB and made into plots. The plots show the max readout per sensor (y-axis) over iterations in the sensor-polling thread. After the grip was complete, the supporting platform beneath the object was removed to see if the gripper could support the object without dropping it.

In addition to the photographs and the collected data from the sensors, some of the experiments were filmed. These films, together with more photographs, can be found on the attached CD. The source code and compiled win32 binaries are also on the CD.

7.1 Soda can, standing up

The first test was the standing up soda can (figure 24 and 25). It was positioned at such a height that the proximal links would contact the soda can first, and when the proximals were completely closed in on the can, the distal links started closing in. The reason for using a soda can in this experiment is to test the control systems ability to grip non-symmetric objects. On top of that, it is easy to observe if this object is being gripped too hard, as the metal will buckle (an empty can is used).

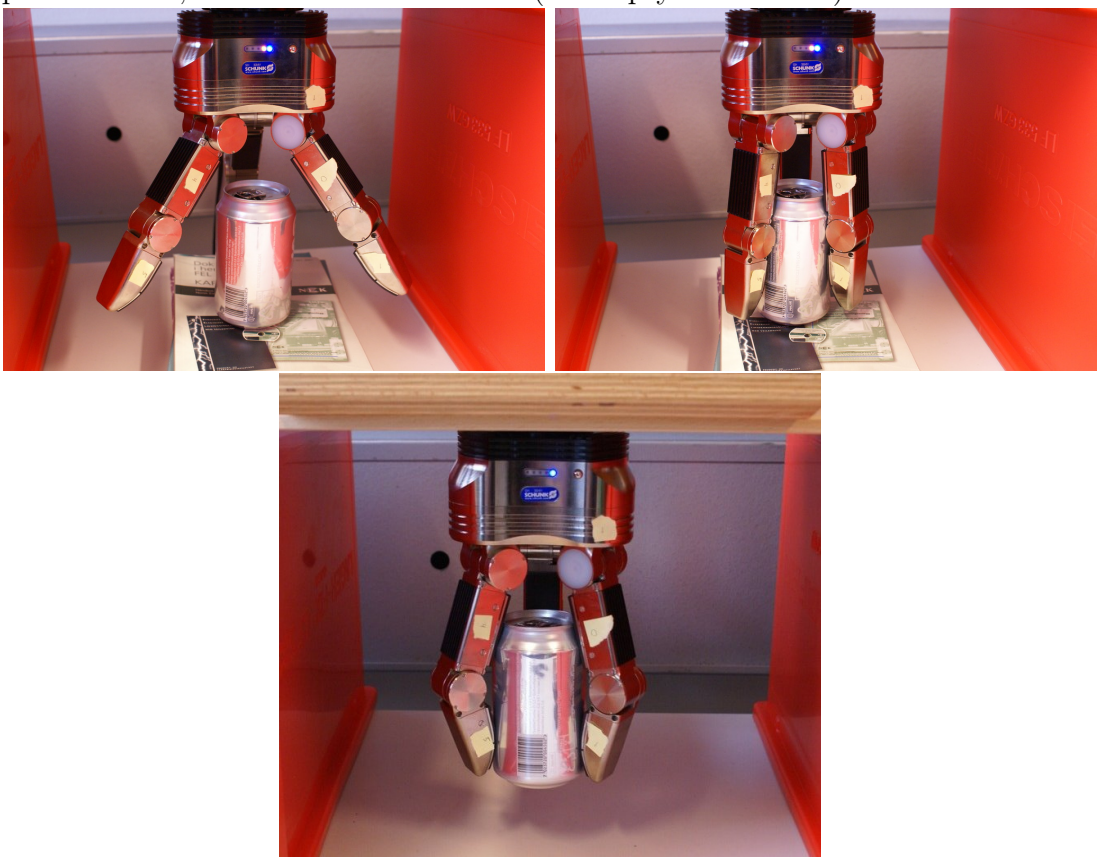


Figure 24: Picking up soda can positioned for the proximal links to impact first. Threshold was set to 6.

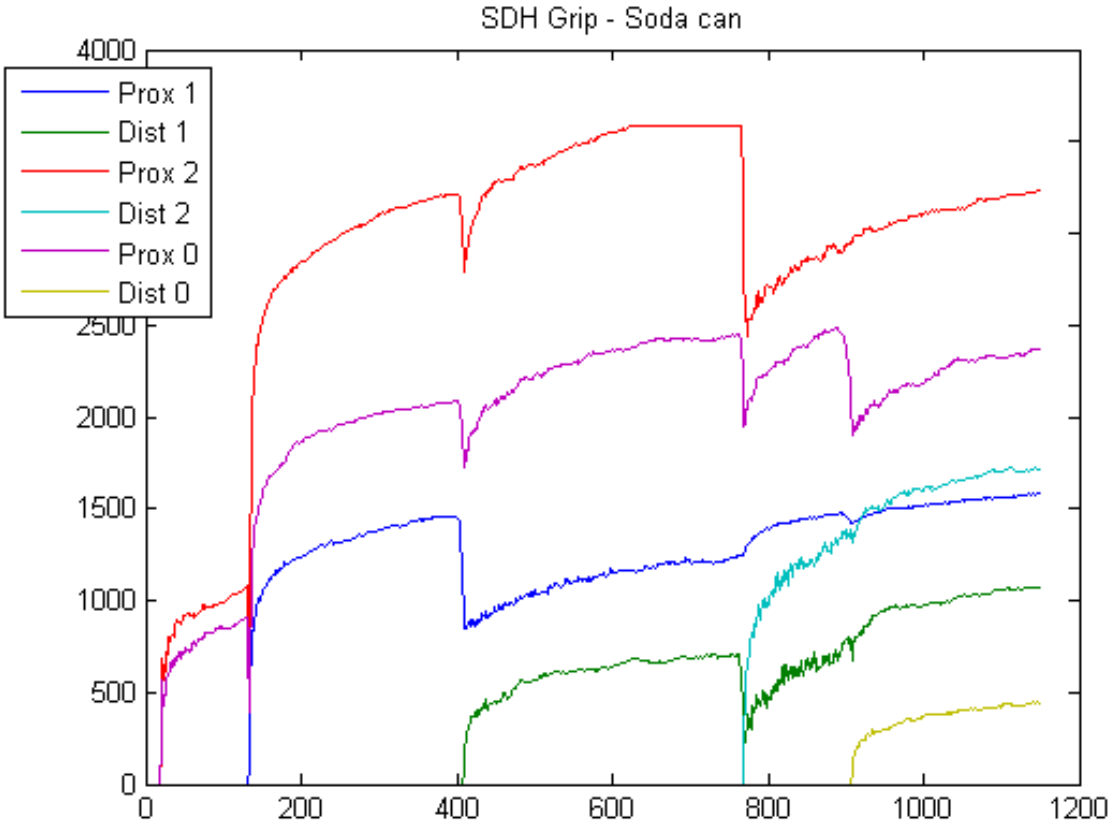


Figure 25: This plot shows the sensor max pressure during the process of gripping an upright soda can. The soda can was positioned for the proximal links to impact first. Note how the pressure for Prox 2 saturates at about sample 600.

The fingers were able to hold the can firmly, but not so hard that the can got bent or damaged. In spite of this, the sensors reached max pressure output, as can be seen from the plot in figure 25: the measurement in sensor Prox2 flattens out on top, meaning the value is saturated. Luckily the proximal links gripped the can by the upper rim of the can which provides additional structural support.

7.2 Soda can, lying down

This test also uses the soda can, but in this case, lying down. The soda can was centred under the palm, and lying on its side. This will test the grippers ability to pick up irregularly shaped objects. The soda can was placed relatively far down on the table, resulting in only the distal links making contact with it. Figure 26 and 27 shows the result of this operation.

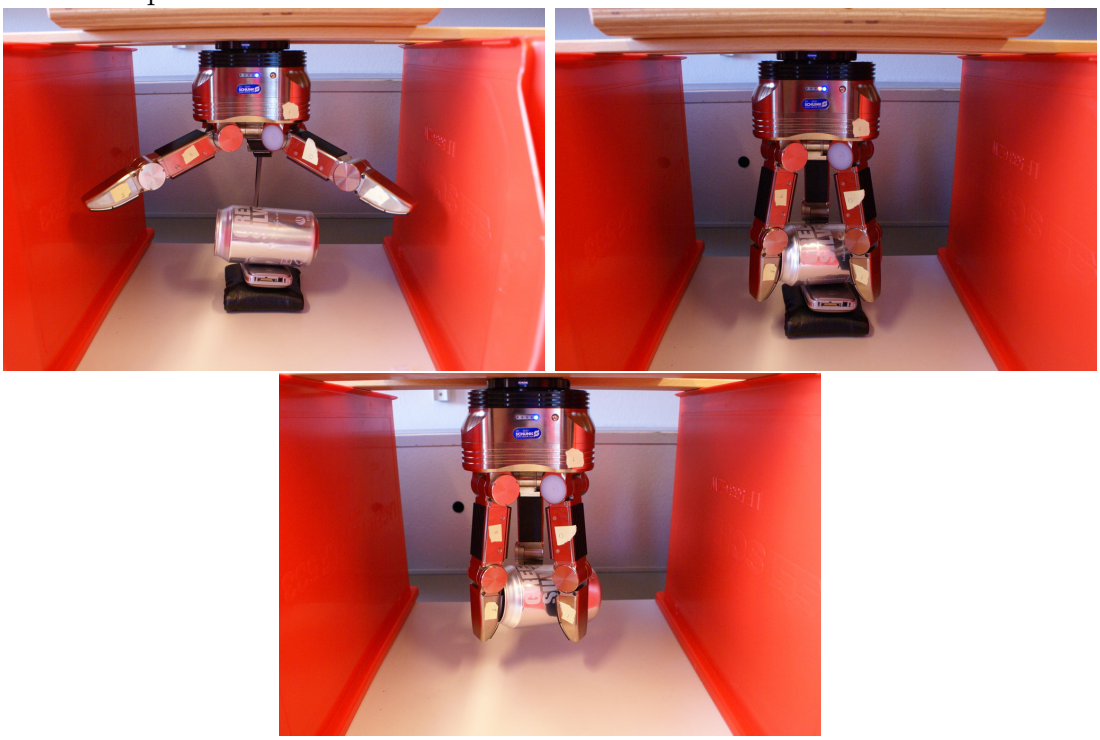


Figure 26: Picking up soda can lying down. The gripper is able to support the object after the platform beneath is removed.

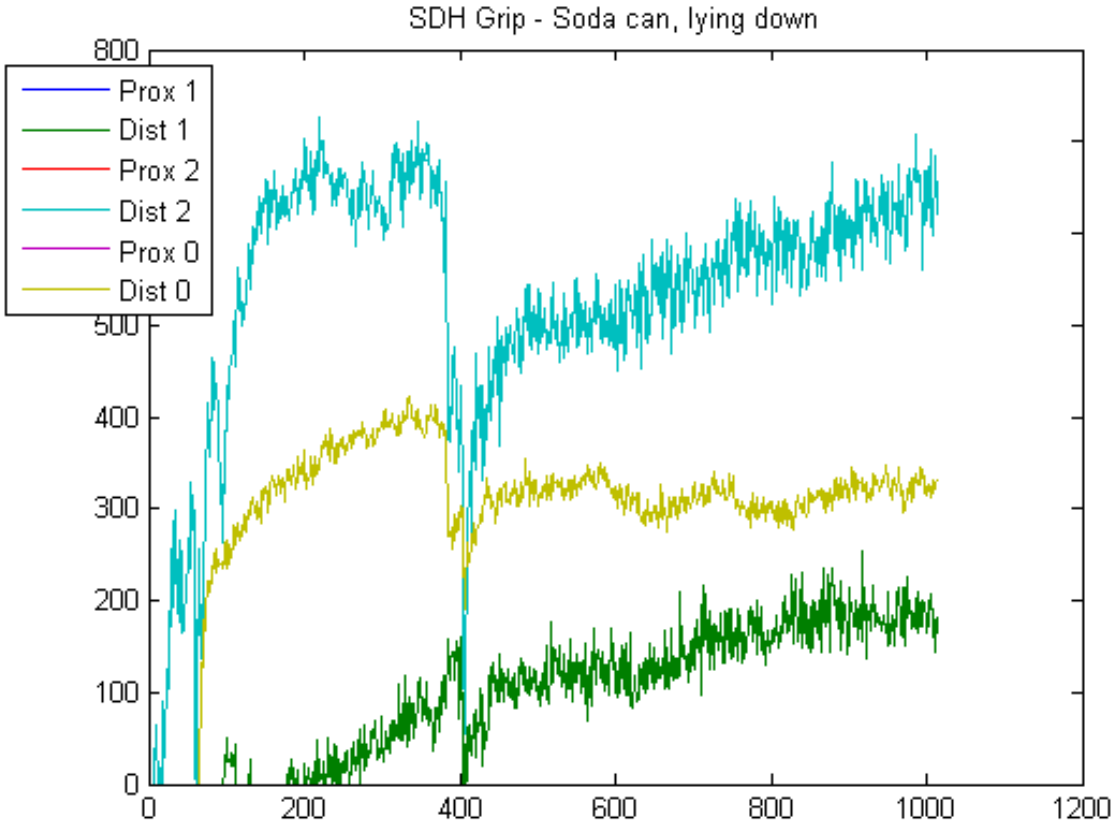


Figure 27: This figure shows the max sensor pressure detected while picking up soda can lying down. Compared to the previous test (figure 25), it can be seen that the pressure registered is quite a bit lower. This might be because less force is applied the further out the contact is made.

The gripper was able to pick up the lying down can nicely and the pressure levels stayed satisfactorily low. Note in the plot (Figure 27) how the pressure drops on all three distal sensors at about sample 400. This was caused by a slight jerk on the can as the platform beneath it was removed by hand. The gradual increase in pressure after this (from 400 onwards) is a bit strange, since the fingers do not move, but might be caused by the sensors calibration properties changing over time by having constant pressure on them. This experiment also tested the new finger-rotation algorithm. The initial hit by Dist 2 was on the edge of the sensor. The finger opened up slightly, rotated and closed in again, getting a nice and centred contact this time.

7.3 Chocolate egg, centred

The next test was picking up a chocolate egg. This was one of the main objectives of this paper, and this experiment will reveal if the gripper is able to halt its movement fast enough to not destroy the egg. The chocolate egg is still protected by its original aluminium wrapping. This is intentionally left on to not pollute the hardware with chocolate. The egg was placed relatively close to the palm-axis, and at such a height that the gripping would be done by the distal links. Figures 28 and 29 show the results of this test.

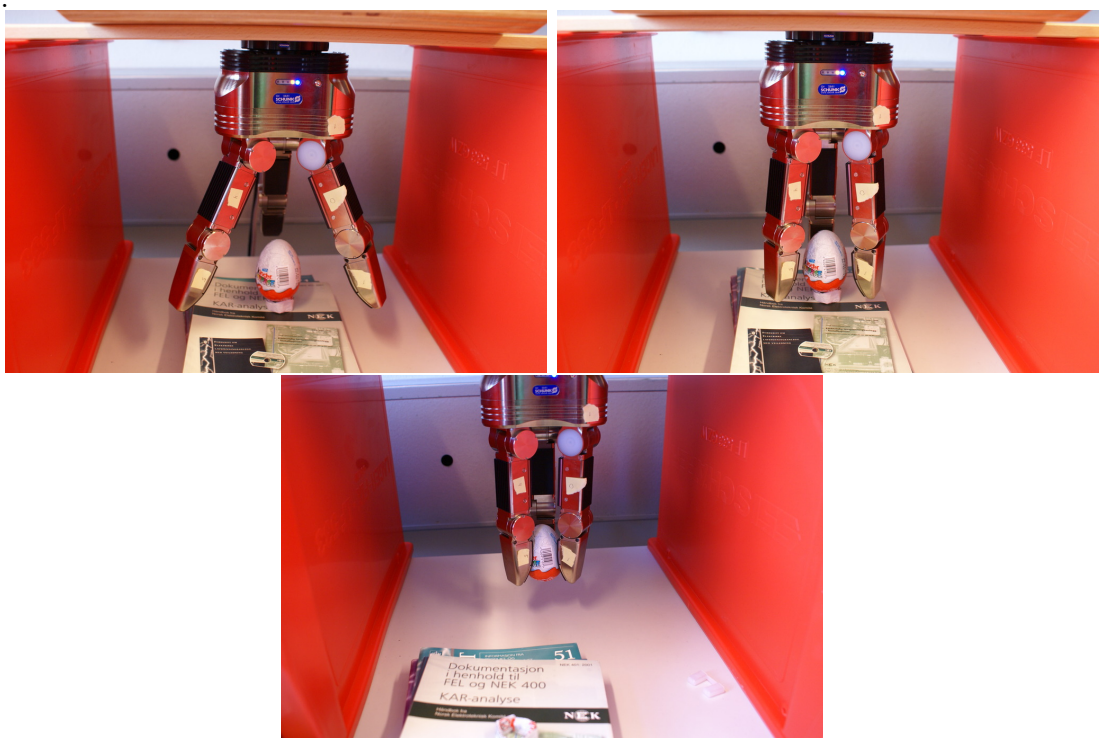


Figure 28: Attempt at gripping a chocolate egg. The egg is positioned relatively close to the palm-axis, and close to the table for a distal grip.

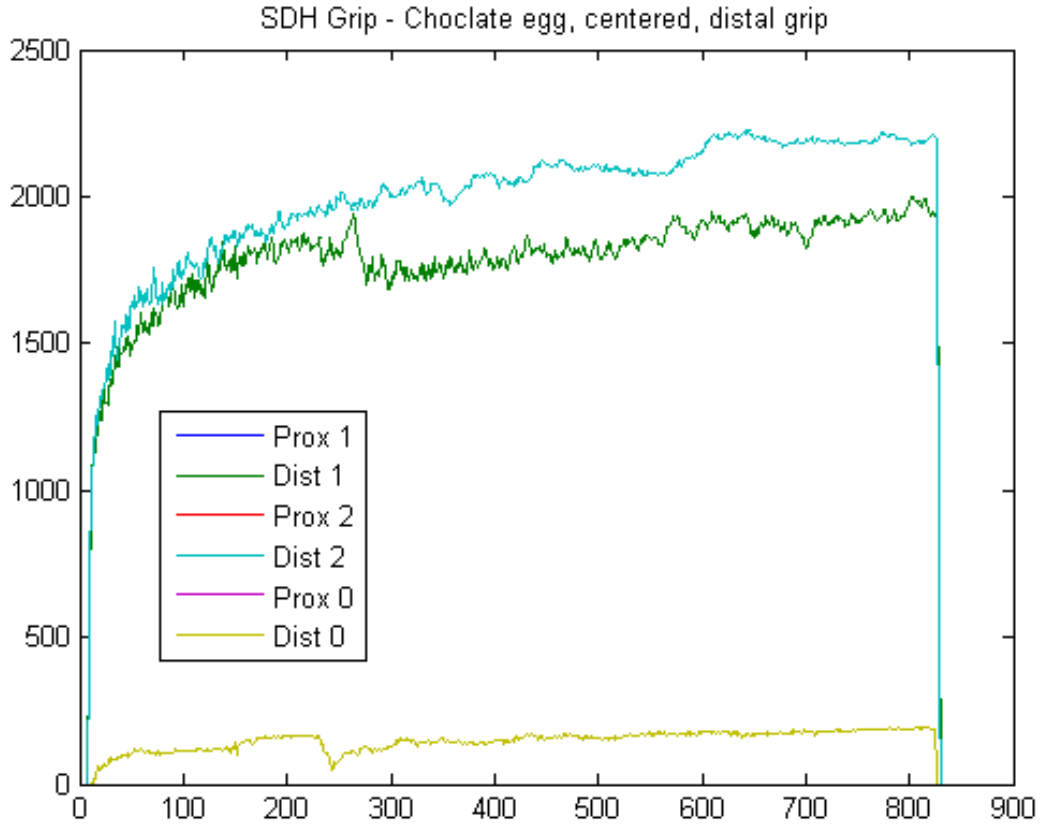


Figure 29: Plot showing the max pressure per sensors while gripping a chocolate egg. The three distal links are able to hold on to the chocolate egg without damaging it.

Note in the plot (Figure 29) how there is a very sudden increase in pressure in Dist 1 and 2, while Dist 0 is able to stop relatively quick without much pressure being applied. This can be explained by the velocity profile of the movements. When a movement is issued to the gripper, it attempts to perform this movement in an economical manner, by using a velocity profile. This velocity profile makes the start and end of a movement slow, while the middle part is faster. Since the gripper has been programmed to halt and recommence movement after a contact event, the gripper will have stopped when it hit the egg with the first two fingers, and after that continued the movement for finger 0. Since finger 0 was already quite close to the egg, it did not get to build up much speed before the contact, and was thus able to halt much sooner than the other two. The sudden drop at sample 830 is the release of the grip.

7.4 Apple, centred

This test is similar to the chocolate egg in the respect that it attempts to grip a spherical object. The apple is less fragile than the chocolate egg, however it weighs quite a bit more, and has a relatively slippery surface. This test will help determine if this gripping algorithm is able to hold on to an object of some weight without dropping it and without damaging it. In figure 30 it can be seen that the apple is placed on a small "foot" made from aluminium wrapping, to lift the apple off the surface below a little. This can be useful if the fingers are expected to "curl" around the object, making a more cupping grip.

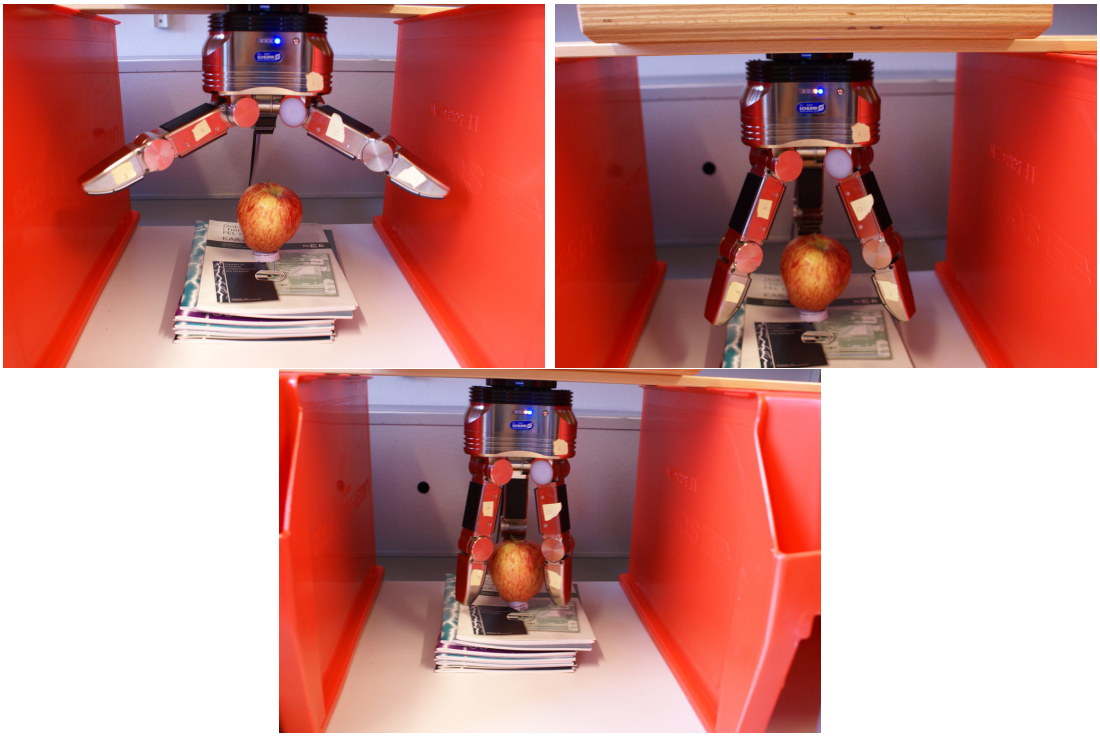


Figure 30: This test will attempt to grip and hold an apple. The platform beneath it is removed after the SDH GUI reports the grip to be completed.

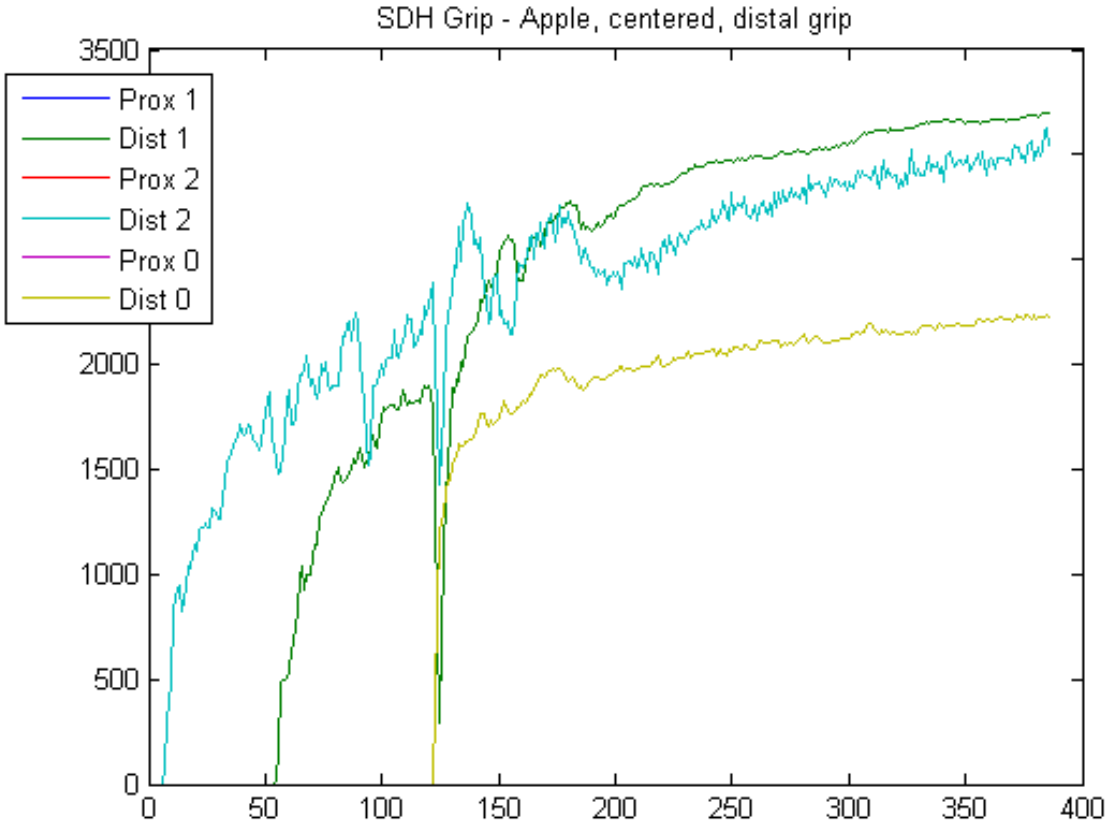


Figure 31: The plots show that the pressure keeps increasing slowly, even after the grip is reported to be complete and the controller has stopped the hand. As discussed earlier, this might be a property of the sensors, where the continued pressure in a sensor makes it gradually output higher and higher values.

This test gave a satisfactory result: The apple was held up nicely by the three distal links, and the gripper did not drop it even after the platform beneath the apple was removed. The plot in figure 31 shows that the pressure did not reach the maximum for the sensor (about 3400), however another unforeseen problem appeared: From the last of the three images in figure 30 it can be seen that the apple almost touches the joint. If the contact happens between the proximal and the distal links the object will only hit the joint in between the links. This does not have any sensors and as such the fingers will continue to close. This was not the case in this test, but had the apple been about 50mm higher up, it would likely have been damaged by the joints.

7.5 Apple, out of reach

Placing an apple out of gripping range, but still in contact range will test the controllers ability to send the correct messages to the parent controller. The apple is placed with all of its body outside of the palm-axis and underneath finger 2. The grasp search first detected the apple on finger 2, halted this finger and kept the other two going. When they reached their target angles without any contacts, the control system set the state "out of reach", calculated the offset needed to centre on the object, and returned this to the parent controller.

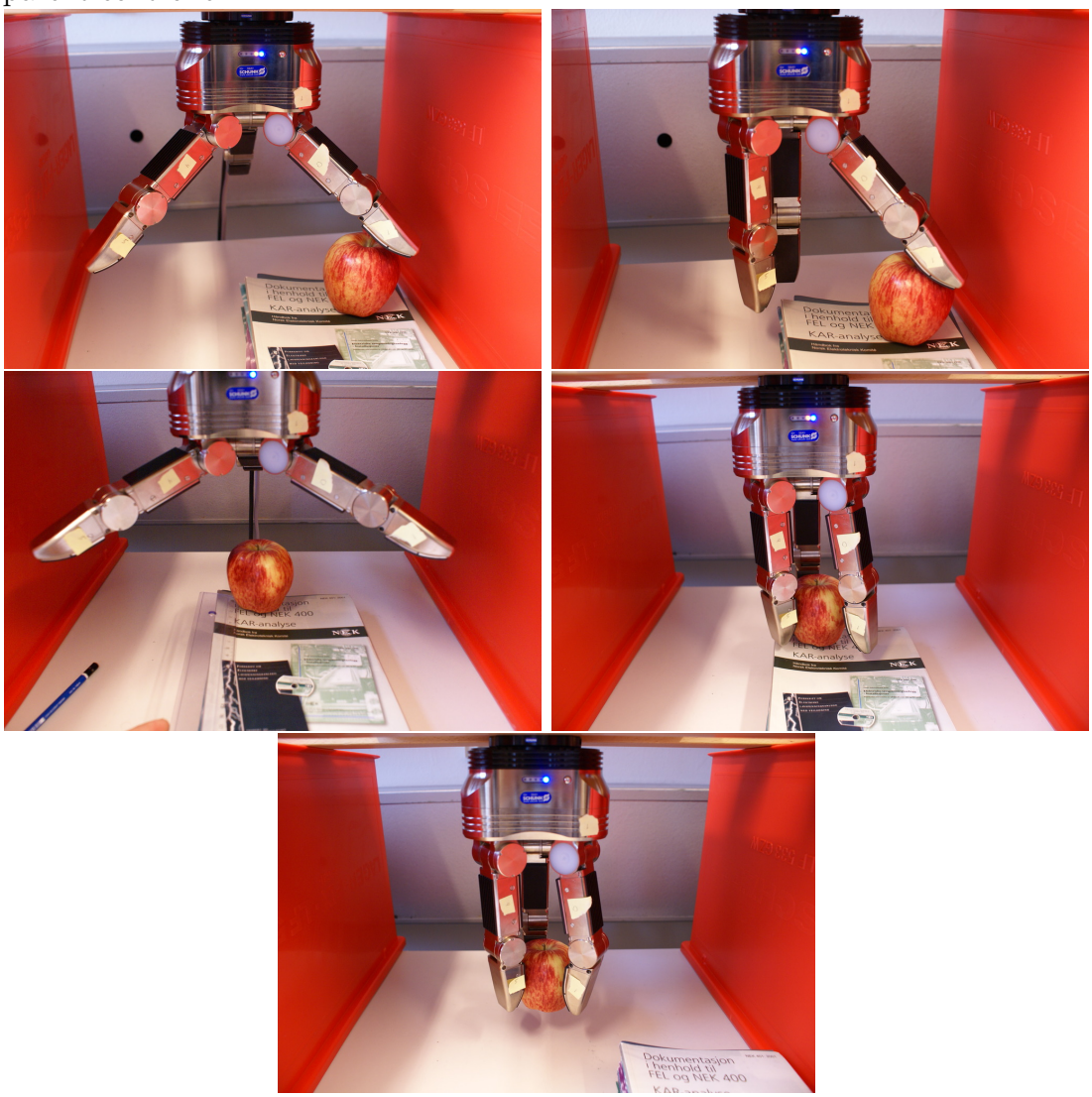


Figure 32: Detecting, relocating and gripping the off-centred apple. After the first finger detects the apple, the other fingers converge on the palm axis. Without any more contacts than from the one finger, it is deemed un-graspable from this position. The application subsequently returns the status to the parent controller and requests a translation to be positioned better.

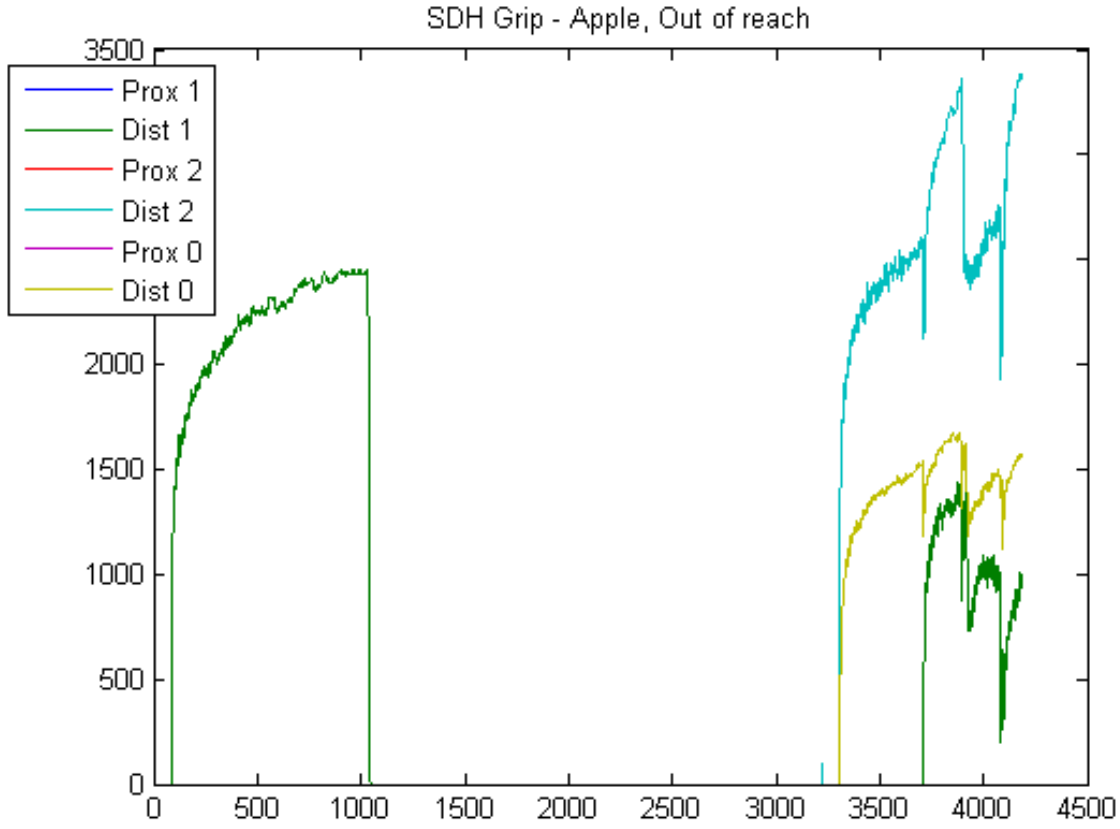


Figure 33: The first contact is made by the distal link in finger 1. At about sample number 1000 the grip is released, and it sends the requested offset to the parent controller. When this reports it to be in place, it starts with a new grip, this time able to contact with all three distal links, and hold on to it. Some empty data in between the two contacts has been removed to emphasise the important events.

Notice the plot in figure 33 how first one sensor hits, and waits for the other two fingers to close. When they find nothing, the coordinates of the hit are used to calculate a desired translation. The gripper control requested the following (x,y) translation: *(72, -80)*. These coordinates are relative to the coordinate system described in chapter 3, figure 8 and given in millimetres. The hand is opened by the gripper control (done manually from the GUI) and instead of moving the gripper, the apple is moved in the opposite direction, *-72,80)*, again relative to the gripper-fixed coordinate system. Measuring the distance with a ruler, moving the apple and then activating the grasp-search again simulates the parent-controller moving the gripper and giving the message to resume search. The apple ended up fairly centred under the palm, and the gripper was able to close in on the apple and hold it, even with the platform beneath removed. In the last image in figure 32 it can be seen how even with the apple ending up slightly off-centred the gripper can close its fingers around it and hold it up, even with the platform beneath removed.

7.6 Chocolate egg, out of reach, distal grip

This test will be similar to the apple out of reach, however the chocolate egg is a smaller and more fragile target. If the translated distance is very inaccurate it could result in an off-centred grip that might damage the egg. To hold the egg in an upright position, a crude "foot" is made from aluminium foil. This does not offer enough support to keep the egg in position if the approaching finger has a high threshold on its pressure sensor.

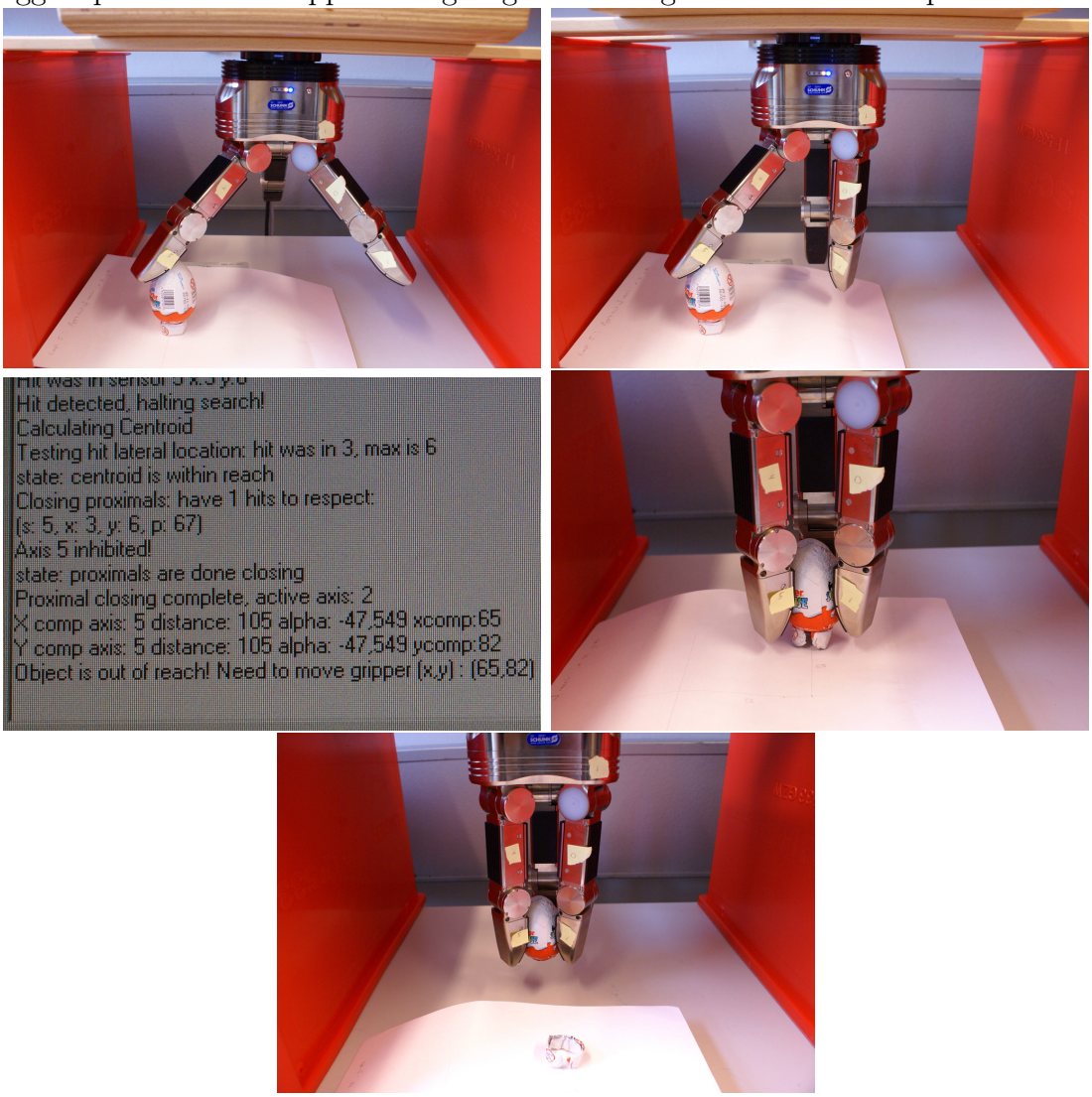


Figure 34: These images show the process of contact with first one finger, the request to be translated, and the second attempt at gripping.

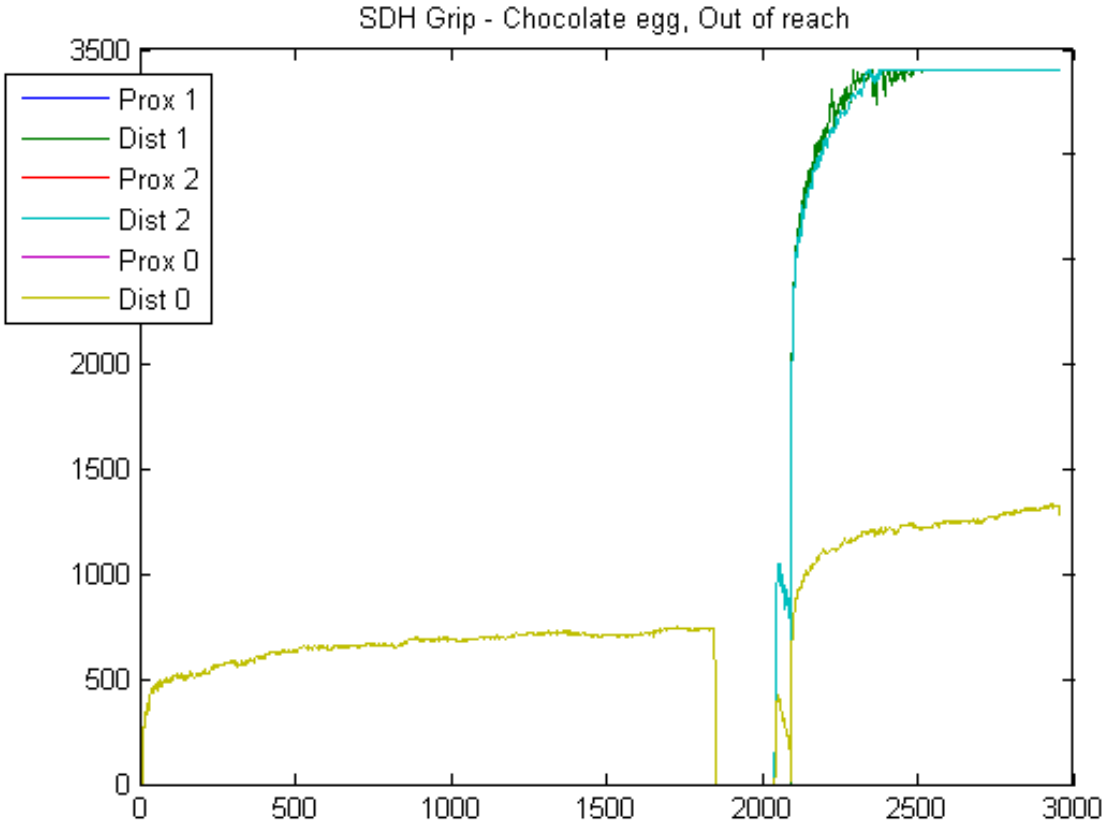


Figure 35: The plot shows how first one finger impacts, the grip is opened and by the time it attempts a new grip, all three fingers are touching the egg. The second grip shows that the pressure read on two of the fingers reaches saturation. This did still not damage the egg fortunately.

The egg was placed under finger 1. After the contact the two others closed in without any hits. The calculated needed offset was *(65,82) mm*. This was measured up on the paper, and the egg moved (in the opposite direction). The second grasp was able to get a nice firm grip on the egg. In figure 34 the gripping and offset-reporting can be seen: In the first image, finger 1 contacts the egg and halts while the other two fingers close in on the centre. Since no other contact is found, the position of the first contact made by finger 1 is used to request a translation (seen in the screen shot from the application). After the egg is moved by hand, the SDH was able to grip and hold it.

8 Discussion

This section will discuss the results of the experiments, the problems encountered, their causes and possible solutions. The changes made to the gripping algorithm during the implementation will also be discussed and evaluated.

8.1 Sensor sensitivity

The first major issue encountered in the experiments was the high threshold in the sensors before they registered contact. This can be seen from the plots in the previous chapter: there is a very sudden "jump" from 0 up to a high level of pressure. As such, even by setting the pressure threshold in the gripping algorithm to 0 meaning any sensor activity above zero would trigger a hit, it did not register anything before the pressure was too high for the intended purpose. Testing showed that if the chocolate egg was not held in place and a finger approached it from the side, the weight in the egg was not enough to trigger a hit in the sensor. If the finger approached from an angle, in a downward motion, it would be able to trigger the hit without damaging the egg. This is due to the downward pressure caused by the finger, in a sense pressing it down towards the table. The problem became apparent when the object was not totally centred, but not far enough out to get hit by the downward motion of the finger. In these cases the finger had a very lateral approach, and in the case of the soda can and chocolate egg, ended up pushing the object against the other two fingers. When the movement of the object got restricted like this, the sensors in the fingers triggered hits and could stop. The apple was heavy enough to trigger a hit without being moved. If the objects were to be fixed in place, for example by some sort of attaching foot underneath, this would not be a big problem, as the foot would restrict it from moving. However, more sensitive pressure sensors would be preferable since it is undesirable that the gripper should "pull" objects in towards its centre.

8.2 Improved off-centred object handling

To handle off-centred objects and determine if they were within reach the initial plan was to estimate a centroid from the contact point. Using this centroid and a pre-determined radius for the object, it would be possible to make an educated guess to see if the object was within range. This is far from optimal, and not even very robust or adapting. It relies on a pre-determined shape and a pre-determined object radius. A simpler and more adaptive solution was implemented: When one finger has a contact, the other two will continue converging on the palm-axis (z-axis). When they reach the target angle (α_{max}) the application logic sees that only one of the fingers have had contact, meaning it is not graspable. However, from the one finger that made contact, the sensor (proximal or distal), and the position on the sensor is retrieved. The angle of the proximal link relative to the palm is also read. Using this data the point of contact can be estimated. The x and y-components of this point is used to create a vector from the palm of the hand to the contact point, and this is the vector that is returned to the parent controller. In practice what this method does is check if there is a part of the object that intersects

the palm axis, or the trajectories of the fingers. This has the advantage of being much more robust than the initial idea. The shape of the object is not that important, only that it is positioned in such a way that part of it is within reach of at least two fingers. Consider the following example: The gripper control uses the initial idea of calculating a centroid using a pre-defined radius for it. However the gripper is used to pick up something larger, or non-spherical, for example a banana. The first finger contacts the banana in point relatively far from the palm-axis. This would result in the centroid being estimated outside the reach of the gripper. With the new algorithm, however, the two other fingers will continue closing in on the palm-axis, where they will find the banana. All three fingers (or at least two) will get a good grip on the banana, and have saved the manipulator arm a translation. The main disadvantage with this new method is that it takes slightly longer than the first idea. The two other fingers have to continue converging on the palm axis, and since the movements should preferably run slowly to be able to stop in time, this takes 10-15 seconds to complete. This might still save time, given the manipulator arm will have to do less movement to close in on targets.

There was also a change in the plan on how to handle contacts that occurred on the

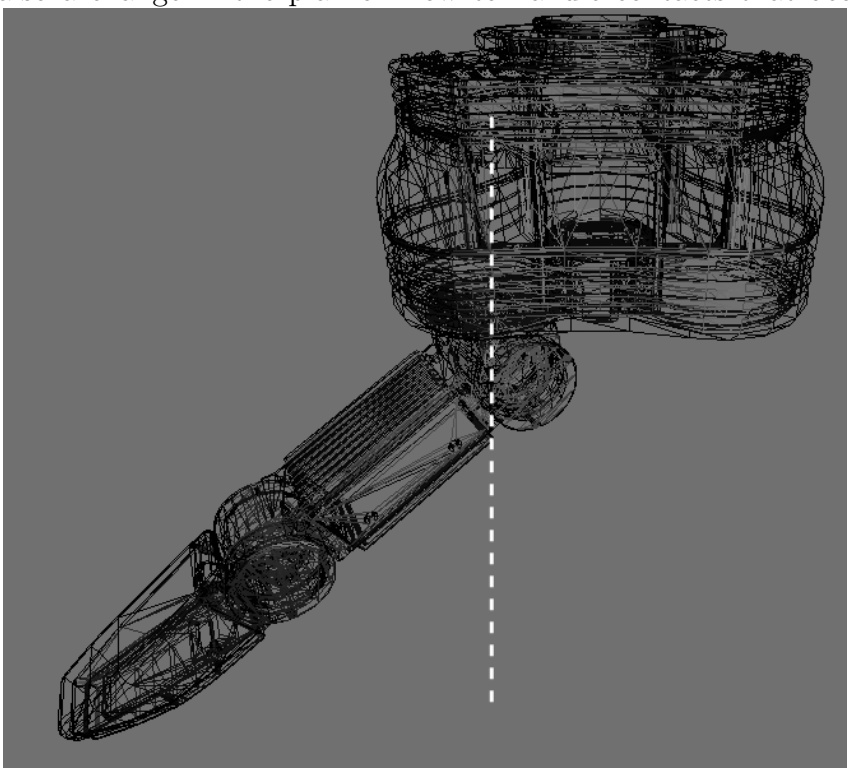


Figure 36: The white dashed line shows the axis of rotation for the finger rotation β .

edge of the sensors. The initial idea was to keep the finger in place, and rotate it in order to see if the contact point could be improved. Some further insight into this solution revealed that this method would only work if the fingers could rotate around their own axis, something that would be true only if the proximal link angle was 0 degrees. This is illustrated in figure 36. A proximal angle of 0 degrees would mean the fingers would be so closed together that an egg would not fit in there anyway, and as such this method was discarded. Instead, a simpler and more intuitive approach was taken: if contact was detected on the edge of a sensor, rotate the fingers an amount that would be linearly

dependent on the distance of the contact point, measured from the root of the finger. This meant a contact very close to the root needed more rotation of the finger, than if the contact was far out on the tip of a finger. This method was not tested in itself, but the experiment with picking up a lying down soda can (chapter 7.1) involved one of the fingers getting a hit on the edge of a sensor. The finger opened up a tiny bit, performed a rotation, and then closed in again, with the contact this time hitting almost in the middle of the sensor.

8.3 Application stability and threading

The stability of the application was perhaps the biggest issue in this project and the major cause for damaged objects during the testing. Since this application is built around the idea of having separate threads for testing pressure, performing movements, running the state machine and other logic, it follows that if the threads start crashing it leads to many problems. One example where it resulted in a destroyed egg was when the thread performing the logics crashed during a grasp search. This thread could then no longer use the data from the sensor-polling thread and was unable to stop the fingers from closing in on the egg, even if the pressure was well over the threshold. The result was a completely destroyed chocolate egg (seen in figure 37) It is not completely clear yet

Figure 37: The result of a crashed thread that caused the fingers to continue closing in despite pressure threshold being surpassed.

what is the cause of this, but some testing showed that larger delays made it more stable and less likely to crash. This might mean that some of the threads need some time to close down and clean up resources before they can be re-initialised. As an example the part of the algorithm that closes the distal fingers runs a while-loop to keep on closing fingers that have not had an impact, or have not reached the target angle. This initially ran without any delays in it, and caused a lot of crashes. By putting in a delay of 100 milliseconds, the crashes became much less frequent, and with half a second, the crashes from this thread disappeared entirely. This is not really a viable solution for the rest of the threads however, since for example the sensor-polling thread is required to run as fast

as possible, in order to halt the movement before crushing the target object. A likely improvement would be to rebuild the application, but this does not fix the fundamental issue, which is the loss of control when there are many threads involved. Multi-threading is an intuitive concept to understand, but with several threads running simultaneously it is easy to control over the logic. The problem is in the SDH library: When a movement is issued, it blocks all calls, and there is nothing one can do until this movement is completed. One approach used during the implementation was to add try-catch blocks around functions that were failing often. This did neutralise almost all the crashes, and instead displayed an error message in the application output window. Nonetheless this is not a solution to the problem, rather just smoothing out the consequences of it. Even by catching most of the exceptions, there were times where it still crashed, or the logic failed, because an important thread had crashed.

8.4 Future work and improvements

This project has shown the potential in adaptive robotic gripping using the SDH, but there are still plenty of improvements and work ahead. The major complications arose around the SDH library's handling of hardware calls. With their blocking calls, the application had to be separated up into several parallel threads. This caused a lot of problems, the threads got aborted when a "halt"-message was given, causing them to crash at times. The biggest priority for future work would therefore be to re-design the application from scratch in the hope of making it more stable and the thread-logic easier to handle. An even bigger improvement would be if future versions of the SDH library came with alternative functions that were non-blocking.

The second big issue encountered was the high threshold on the contact sensors. This made the gripper unable to sense when it came in contact with light objects, like the chocolate egg. It was still able to sense the egg, but not without pushing it against the other fingers. Although it was able to stop before it caused damage, it is undesirable that the gripper has to move objects and squeeze them between its fingers to sense them. There is a lack of calibration options in the SDH's sensor system, and this would have helped immensely. Again, possible future releases of the SDH library might offer better control over the sensors.

This project has designed an entire system consisting of a robotic gripper, a manipulator arm, and their control systems. Since the work done has only been on the gripper control system, the logical next step would be to design a corresponding control system for a manipulator arm, and an overall control system to manage them both.

9 Conclusion

Robotic gripping offers a fantastic possibility of improving performance, precision and safety in almost all industries. As technology advances, the possible applications of robotics and robotic gripping also increase. However, their increased complexity and the growing complexity of their tasks call for more and more advanced control systems.

This project has looked at the task of safely gripping objects that do not have a pre-programmed position, size, shape or durability. By using adaptive gripping methods, the gripper attempts to close the fingers around the object, and hold on to it in a manner that mimics the human hand. One of the central aspects of this operation is the feedback, both in the human hand and in the robotic analogy. This project has focused on working with the SCHUNK Dextrous Hand, and with its tactile feedback sensors it has proven itself as a worthy candidate for an adaptive gripper. The robotic gripper, as with the human hand, is not very useful without an arm to move it. This project is thus designed with the idea of the gripper being attached to a robotic manipulator. The control systems for both those pieces of hardware should communicate via an overall control system. This way the feedback from the gripper can make small changes in the robotic arm in order to improve the position.

This paper describes the work done on the control system for the gripper. The communication to and from the overall control system has been abstracted out, and replaced with a simple interface created in C++ .NET. This interface allows inputs and outputs of the same types that the gripper would use to communicate with the overall control system, were it present. An algorithm to grip unknown objects, both in position, size and shape was implemented and tested. The tests were done with an apple, a soda can (both upright and lying down) and a chocolate egg. The latter would illuminate the grippers ability to handle fragile objects, while still holding on firmly enough to not let them slip out of the grip.

Tests show that the gripping algorithm was very successful in all cases, and did not damage any of the objects, while still holding on to them firmly enough to not drop them, even when the platform beneath them was removed. The main problem encountered was in the application itself. Since the application was programmed in multiple threads, the logic was hard to keep track of, and caused stability problems with the application. Crashing threads caused the logic to fail at times, and this **did** cause some objects to get damaged. It should be emphasised that this was a problem with the implementation of the application, not the gripping algorithm itself. A re-structuring of the application might help control the logic and the different threads with more ease.

Another problem encountered was the lack of sensitivity in the sensors on the gripper. This can be seen from the plots in the results chapter. There is a very sudden jump from a 0-pressure level up to a very high level. The result of this was that the gripper was unable to "touch" lighter objects, like the chocolate egg, without pushing them. When the egg was pushed against the other two fingers, it would trigger the sensors, and the contact would be detected without having damaged the egg. Still, this

is not optimal as it is not desirable to have to "pull" objects into the palm to detect them.

One problem that surfaced during the experiments was the space between the sensors. Although this did not affect any of the tests done in this project, this is a potential problem. There is a gap of approximately 1.5 cm, between the proximal and the distal finger sensors. If this section is the only one in contact with the object, the finger would not detect the object, and continue closing in, most likely damaging the object if it is fragile. There is not much that can be done with this, other than perhaps supplementing the entire system with optical or other detection methods that could help detect the objects.

This project has shown some of the possibilities with using an adaptive gripping control system, and discussed some of the possible applications for it. Although there are still some problems with the application, and the overall control system has not been created, this project shows promise in the area of an adaptive gripping and multi-fingered gripper control and feedback.

References

[1] Bullet Physics. http://bulletphysics.org.

[2] Open Dynamics Engine. http://ode.org.

[3] SCHUNK Dextrous Hand, 2010. http://www.schunk-modular-robotics.com/left-navigation/service-robotics/components/actuators/robotic-hands/sdh.html.

[4] P. Allen. Integrating vision and touch for object recognition tasks. *Multisensor integration and fusion for intelligent machines and systems*, pages 407–440, 1995.

[5] P. Allen and R. Bajcsy. Object recognition using vision and touch. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence, Los Angeles, CA*, pages 1131–1137, 1984.

[6] C.H. An, C.G. Atkeson, and J.M. Hollerbach. *Model-based control of a robot manipulator*.

[7] W. Hardin. Designers make robotic grippers with productivity, energy savings in mind. Oct 2005. *http://www.industrialcontroldesignline.com/howto/roboticsprototyping/172301304*.

[8] R.D. Howe. Tactile sensing and control of robotic manipulation. *Advanced Robotics*, 8(3):245–261, 1993.

[9] J. Jørgensen and H. Petersen. Usage of simulations to plan stable grasping of unknown objects with a 3-fingered Schunk hand. In *Workshop on Robot Simulators: Available Software, Scientific Applications and Future Trends, ICRA*, 2008.

[10] S. Monteiro. Developing a user interface for schunk dextrous hand. 2009.

[11] A. Namiki, T. Komuro, and M. Ishikawa. High-speed sensory–motor fusion for robotic grasping. *Measurement Science and Technology*, 13:1767–1778, 2002.

A Source Code

A.1 SDHGrip2.cpp

```
1  // SDHGrip2.cpp : main project file.
2  /* This file is the main runnable file. Initialises the Windows ↩
       Form.
3   * SÃ¸lve Monteiro, (c) 2010
4   */
5
6  #include "stdafx.h"
7  #include "SDHGUI.h"
8
9  using namespace SDHGrip2;
10
11 [STAThreadAttribute]
12 int main(array<System::String ^> ^args)
13 {
14   // Enabling Windows XP visual effects before any controls are ↩
       created
15   Application::EnableVisualStyles();
16   Application::SetCompatibleTextRenderingDefault(false);
17
18   // Create the main window and run it
19   Application::Run(gcnew SDHGUI());
20   return 0;
21 }
```

A.2 SDHGUI.h

```
1  /* SDHGUI.h
2   * Contains most of the Windows-generated Forms data, as well as ↩
       some
3   * GUI Methods to connect to backend
4   * SÃ¸lve Monteiro, (c) 2010
5   * */
6
7  #pragma once
8  #include "SDHBackend.h"
9
10
11 namespace SDHGrip2 {
12
13   using namespace System;
14   using namespace System::ComponentModel;
15   using namespace System::Collections;
16   using namespace System::Windows::Forms;
17   using namespace System::Data;
18   using namespace System::Drawing;
19   using namespace System::Threading;
20   using namespace System::Collections;
21
22   /// <summary>
23   /// Summary for SDGUI
24   /// </summary>
25   public ref class SDHGUI : public System::Windows::Forms::Form
26   {
27   public:
28
29     Thread ^sensorThread, ^graspThread;
30     SDHBackend ^sdhbackend;
31     int maxPressureValue;
32     bool running;
33   private: System::Windows::Forms::Timer^  printTimer;
34   private: System::Windows::Forms::Label^  label1;
35   private: System::Windows::Forms::Label^  label_maxPressure;
36   private: System::Windows::Forms::Timer^  timer_maxPressurePoll;
37   private: System::Windows::Forms::Button^  button3;
38   private: System::Windows::Forms::Button^  button4;
39   private: System::Windows::Forms::Button^  button5;
40   private: System::Windows::Forms::TrackBar^  ↩
       trackBar_changeThreshhold;
41
42   private: System::Windows::Forms::Label^  label2;
43   private: System::Windows::Forms::Label^  label_threshhold;
44   private: System::Windows::Forms::Button^  button6;
```

```
45    private: System::Windows::Forms::Button^  button7;
46    private: System::Windows::Forms::Button^  button8;
47    private: System::Windows::Forms::ProgressBar^  pb_0;
48    private: System::Windows::Forms::ProgressBar^  pb_1;
49    private: System::Windows::Forms::ProgressBar^  pb_3;
50    private: System::Windows::Forms::ProgressBar^  pb_2;
51    private: System::Windows::Forms::ProgressBar^  pb_5;
52    private: System::Windows::Forms::ProgressBar^  pb_4;
53    private: System::Windows::Forms::Label^  label3;
54    private: System::Windows::Forms::Label^  label4;
55    private: System::Windows::Forms::Label^  label5;
56    private: System::Windows::Forms::Label^  label_state;
57
58
59    public:
60      ArrayList ^printBuffer;
61      SDHGUI(void)
62      {
63        maxPressureValue=0;
64        running=true;
65        InitializeComponent();
66
67        printBuffer = gcnew ArrayList();
68        printout("Form initialised");
69        sdhbackend = gcnew SDHBackend();
70        sdhbackend->init(printBuffer);
71
72      }
73
74      void printout(Object ^o){
75        printBuffer->Add(o);
76      }
77
78
79
80    protected:
81      /// <summary>
82      /// Clean up any resources being used.
83      /// </summary>
84      ~SDHGUI()
85      {
86        if (components)
87        {
88          delete components;
89        }
90      }
91    private: System::Windows::Forms::Button^  button1;
92    protected:
93    private: System::Windows::Forms::RichTextBox^  richTextBox1;
```

```
94    private: System::Windows::Forms::Button^  button2;
95    private: System::ComponentModel::IContainer^  components;
96
97    private:
98      /// <summary>
99      /// Required designer variable.
100     /// </summary>
101
102
103 #pragma region Windows Form Designer generated code
104     /// <summary>
105     /// Required method for Designer support - do not modify
106     /// the contents of this method with the code editor.
107     /// </summary>
108     void InitializeComponent(void)
109     {
110       this->components = (gcnew System::ComponentModel::Container()↩
          );
111       this->button1 = (gcnew System::Windows::Forms::Button());
112       this->richTextBox1 = (gcnew System::Windows::Forms::↩
          RichTextBox());
113       this->button2 = (gcnew System::Windows::Forms::Button());
114       this->printTimer = (gcnew System::Windows::Forms::Timer(this↩
          ->components));
115       this->label1 = (gcnew System::Windows::Forms::Label());
116       this->label_maxPressure = (gcnew System::Windows::Forms::↩
          Label());
117       this->timer_maxPressurePoll = (gcnew System::Windows::Forms::↩
          Timer(this->components));
118       this->button3 = (gcnew System::Windows::Forms::Button());
119       this->button4 = (gcnew System::Windows::Forms::Button());
120       this->button5 = (gcnew System::Windows::Forms::Button());
121       this->trackBar_changeThreshhold = (gcnew System::Windows::↩
          Forms::TrackBar());
122       this->label2 = (gcnew System::Windows::Forms::Label());
123       this->label_threshhold = (gcnew System::Windows::Forms::Label↩
          ());
124       this->button6 = (gcnew System::Windows::Forms::Button());
125       this->button7 = (gcnew System::Windows::Forms::Button());
126       this->button8 = (gcnew System::Windows::Forms::Button());
127       this->pb_0 = (gcnew System::Windows::Forms::ProgressBar());
128       this->pb_1 = (gcnew System::Windows::Forms::ProgressBar());
129       this->pb_3 = (gcnew System::Windows::Forms::ProgressBar());
130       this->pb_2 = (gcnew System::Windows::Forms::ProgressBar());
131       this->pb_5 = (gcnew System::Windows::Forms::ProgressBar());
132       this->pb_4 = (gcnew System::Windows::Forms::ProgressBar());
133       this->label3 = (gcnew System::Windows::Forms::Label());
134       this->label4 = (gcnew System::Windows::Forms::Label());
135       this->label5 = (gcnew System::Windows::Forms::Label());
```

```
136         this->label_state = (gcnew System::Windows::Forms::Label());
137         (cli::safe_cast<System::ComponentModel::ISupportInitialize^ ↩
               >(this->trackBar_changeThreshhold))->BeginInit();
138         this->SuspendLayout();
139         //
140         // button1
141         //
142         this->button1->Location = System::Drawing::Point(13, 26);
143         this->button1->Name = L"button1";
144         this->button1->Size = System::Drawing::Size(98, 23);
145         this->button1->TabIndex = 0;
146         this->button1->Text = L"Connect to HW";
147         this->button1->UseVisualStyleBackColor = true;
148         this->button1->Click += gcnew System::EventHandler(this, &↩
               SDHGUI::GUIConnect);
149         //
150         // richTextBox1
151         //
152         this->richTextBox1->Location = System::Drawing::Point(12, ↩
               238);
153         this->richTextBox1->Name = L"richTextBox1";
154         this->richTextBox1->Size = System::Drawing::Size(660, 219);
155         this->richTextBox1->TabIndex = 1;
156         this->richTextBox1->Text = L"";
157         //
158         // button2
159         //
160         this->button2->Location = System::Drawing::Point(13, 84);
161         this->button2->Name = L"button2";
162         this->button2->Size = System::Drawing::Size(84, 23);
163         this->button2->TabIndex = 2;
164         this->button2->Text = L"Open Hand";
165         this->button2->UseVisualStyleBackColor = true;
166         this->button2->Click += gcnew System::EventHandler(this, &↩
               SDHGUI::GUIOpenHand);
167         //
168         // printTimer
169         //
170         this->printTimer->Enabled = true;
171         this->printTimer->Interval = 50;
172         this->printTimer->Tick += gcnew System::EventHandler(this, &↩
               SDHGUI::printOutBuffer);
173         //
174         // label1
175         //
176         this->label1->AutoSize = true;
177         this->label1->Location = System::Drawing::Point(407, 25);
178         this->label1->Name = L"label1";
179         this->label1->Size = System::Drawing::Size(74, 13);
```

```
180         this->label1->TabIndex = 3;
181         this->label1->Text = L"Max Pressure:";
182         //
183         // label_maxPressure
184         //
185         this->label_maxPressure->AutoSize = true;
186         this->label_maxPressure->Location = System::Drawing::Point↩
                (487, 25);
187         this->label_maxPressure->Name = L"label_maxPressure";
188         this->label_maxPressure->Size = System::Drawing::Size(13, 13)↩
                ;
189         this->label_maxPressure->TabIndex = 4;
190         this->label_maxPressure->Text = L"0";
191         //
192         // timer_maxPressurePoll
193         //
194         this->timer_maxPressurePoll->Enabled = true;
195         this->timer_maxPressurePoll->Interval = 10;
196         this->timer_maxPressurePoll->Tick += gcnew System::↩
                EventHandler(this, &SDHGUI::getMaxPressure);
197         //
198         // button3
199         //
200         this->button3->Location = System::Drawing::Point(103, 84);
201         this->button3->Name = L"button3";
202         this->button3->Size = System::Drawing::Size(97, 23);
203         this->button3->TabIndex = 5;
204         this->button3->Text = L"Grasp (Search)";
205         this->button3->UseVisualStyleBackColor = true;
206         this->button3->Click += gcnew System::EventHandler(this, &↩
                SDHGUI::StartSearchMode);
207         //
208         // button4
209         //
210         this->button4->Location = System::Drawing::Point(206, 84);
211         this->button4->Name = L"button4";
212         this->button4->Size = System::Drawing::Size(75, 23);
213         this->button4->TabIndex = 6;
214         this->button4->Text = L"Trigger hit";
215         this->button4->UseVisualStyleBackColor = true;
216         this->button4->Click += gcnew System::EventHandler(this, &↩
                SDHGUI::GUIManualHit);
217         //
218         // button5
219         //
220         this->button5->Location = System::Drawing::Point(596, 209);
221         this->button5->Name = L"button5";
222         this->button5->Size = System::Drawing::Size(75, 23);
223         this->button5->TabIndex = 7;
```

```
224         this->button5->Text = L"Clear Log";
225         this->button5->UseVisualStyleBackColor = true;
226         this->button5->Click += gcnew System::EventHandler(this, &↩
               SDHGUI::GUIClearLog);
227         //
228         // trackBar_changeThreshhold
229         //
230         this->trackBar_changeThreshhold->Location = System::Drawing::↩
               Point(410, 54);
231         this->trackBar_changeThreshhold->Maximum = 500;
232         this->trackBar_changeThreshhold->Name = L"↩
               trackBar_changeThreshhold";
233         this->trackBar_changeThreshhold->Size = System::Drawing::Size↩
               (186, 45);
234         this->trackBar_changeThreshhold->TabIndex = 8;
235         this->trackBar_changeThreshhold->TickFrequency = 10;
236         this->trackBar_changeThreshhold->Value = 20;
237         this->trackBar_changeThreshhold->ValueChanged += gcnew System↩
               ::EventHandler(this, &SDHGUI::GUIChangeThresshold);
238         //
239         // label2
240         //
241         this->label2->AutoSize = true;
242         this->label2->Location = System::Drawing::Point(407, 38);
243         this->label2->Name = L"label2";
244         this->label2->Size = System::Drawing::Size(66, 13);
245         this->label2->TabIndex = 9;
246         this->label2->Text = L"Threshhold: ";
247         //
248         // label_threshhold
249         //
250         this->label_threshhold->AutoSize = true;
251         this->label_threshhold->Location = System::Drawing::Point↩
               (487, 38);
252         this->label_threshhold->Name = L"label_threshhold";
253         this->label_threshhold->Size = System::Drawing::Size(19, 13);
254         this->label_threshhold->TabIndex = 10;
255         this->label_threshhold->Text = L"20";
256         //
257         // button6
258         //
259         this->button6->Location = System::Drawing::Point(13, 113);
260         this->button6->Name = L"button6";
261         this->button6->Size = System::Drawing::Size(98, 23);
262         this->button6->TabIndex = 11;
263         this->button6->Text = L"Close Proximals";
264         this->button6->UseVisualStyleBackColor = true;
265         this->button6->Click += gcnew System::EventHandler(this, &↩
               SDHGUI::GUICloseProximals);
```

```
266         //
267         // button7
268         //
269         this->button7->Location = System::Drawing::Point(13, 142);
270         this->button7->Name = L"button7";
271         this->button7->Size = System::Drawing::Size(98, 23);
272         this->button7->TabIndex = 12;
273         this->button7->Text = L"Close Distals";
274         this->button7->UseVisualStyleBackColor = true;
275         this->button7->Click += gcnew System::EventHandler(this, &↩
               SDHGUI::GUICloseDistals);
276         //
277         // button8
278         //
279         this->button8->Location = System::Drawing::Point(181, 26);
280         this->button8->Name = L"button8";
281         this->button8->Size = System::Drawing::Size(100, 23);
282         this->button8->TabIndex = 13;
283         this->button8->Text = L"Full Auto Grasp";
284         this->button8->UseVisualStyleBackColor = true;
285         this->button8->Click += gcnew System::EventHandler(this, &↩
               SDHGUI::GUIInitAutoGrasp);
286         //
287         // pb_0
288         //
289         this->pb_0->Location = System::Drawing::Point(352, 142);
290         this->pb_0->Maximum = 3600;
291         this->pb_0->Name = L"pb_0";
292         this->pb_0->Size = System::Drawing::Size(100, 14);
293         this->pb_0->Style = System::Windows::Forms::ProgressBarStyle↩
               ::Continuous;
294         this->pb_0->TabIndex = 14;
295         //
296         // pb_1
297         //
298         this->pb_1->Location = System::Drawing::Point(458, 142);
299         this->pb_1->Maximum = 3600;
300         this->pb_1->Name = L"pb_1";
301         this->pb_1->Size = System::Drawing::Size(100, 14);
302         this->pb_1->Style = System::Windows::Forms::ProgressBarStyle↩
               ::Continuous;
303         this->pb_1->TabIndex = 15;
304         //
305         // pb_3
306         //
307         this->pb_3->Location = System::Drawing::Point(458, 122);
308         this->pb_3->Maximum = 3600;
309         this->pb_3->Name = L"pb_3";
310         this->pb_3->Size = System::Drawing::Size(100, 14);
```

```
311         this->pb_3->Style = System::Windows::Forms::ProgressBarStyle↩
               ::Continuous;
312         this->pb_3->TabIndex = 17;
313         //
314         // pb_2
315         //
316         this->pb_2->Location = System::Drawing::Point(352, 122);
317         this->pb_2->Maximum = 3600;
318         this->pb_2->Name = L"pb_2";
319         this->pb_2->Size = System::Drawing::Size(100, 14);
320         this->pb_2->Style = System::Windows::Forms::ProgressBarStyle↩
               ::Continuous;
321         this->pb_2->TabIndex = 16;
322         //
323         // pb_5
324         //
325         this->pb_5->Location = System::Drawing::Point(458, 162);
326         this->pb_5->Maximum = 3600;
327         this->pb_5->Name = L"pb_5";
328         this->pb_5->Size = System::Drawing::Size(100, 14);
329         this->pb_5->Style = System::Windows::Forms::ProgressBarStyle↩
               ::Continuous;
330         this->pb_5->TabIndex = 19;
331         //
332         // pb_4
333         //
334         this->pb_4->Location = System::Drawing::Point(352, 162);
335         this->pb_4->Maximum = 3600;
336         this->pb_4->Name = L"pb_4";
337         this->pb_4->Size = System::Drawing::Size(100, 14);
338         this->pb_4->Style = System::Windows::Forms::ProgressBarStyle↩
               ::Continuous;
339         this->pb_4->TabIndex = 18;
340         //
341         // label3
342         //
343         this->label3->AutoSize = true;
344         this->label3->Location = System::Drawing::Point(349, 190);
345         this->label3->Name = L"label3";
346         this->label3->Size = System::Drawing::Size(51, 13);
347         this->label3->TabIndex = 20;
348         this->label3->Text = L"Proximals";
349         //
350         // label4
351         //
352         this->label4->AutoSize = true;
353         this->label4->Location = System::Drawing::Point(455, 190);
354         this->label4->Name = L"label4";
355         this->label4->Size = System::Drawing::Size(38, 13);
```

```
356         this->label4->TabIndex = 21;
357         this->label4->Text = L"Distals";
358         //
359         // label5
360         //
361         this->label5->AutoSize = true;
362         this->label5->Location = System::Drawing::Point(12, 209);
363         this->label5->Name = L"label5";
364         this->label5->Size = System::Drawing::Size(38, 13);
365         this->label5->TabIndex = 22;
366         this->label5->Text = L"State: ";
367         //
368         // label_state
369         //
370         this->label_state->AutoSize = true;
371         this->label_state->Location = System::Drawing::Point(56, 209)↩
                ;
372         this->label_state->Name = L"label_state";
373         this->label_state->Size = System::Drawing::Size(0, 13);
374         this->label_state->TabIndex = 23;
375         //
376         // SDHGUI
377         //
378         this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
379         this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::↩
                Font;
380         this->ClientSize = System::Drawing::Size(684, 469);
381         this->Controls->Add(this->label_state);
382         this->Controls->Add(this->label5);
383         this->Controls->Add(this->label4);
384         this->Controls->Add(this->label3);
385         this->Controls->Add(this->pb_5);
386         this->Controls->Add(this->pb_4);
387         this->Controls->Add(this->pb_3);
388         this->Controls->Add(this->pb_2);
389         this->Controls->Add(this->pb_1);
390         this->Controls->Add(this->pb_0);
391         this->Controls->Add(this->button8);
392         this->Controls->Add(this->button7);
393         this->Controls->Add(this->button6);
394         this->Controls->Add(this->label_threshhold);
395         this->Controls->Add(this->label2);
396         this->Controls->Add(this->trackBar_changeThreshhold);
397         this->Controls->Add(this->button5);
398         this->Controls->Add(this->button4);
399         this->Controls->Add(this->button3);
400         this->Controls->Add(this->label_maxPressure);
401         this->Controls->Add(this->label1);
402         this->Controls->Add(this->button2);
```

```
403         this->Controls->Add(this->richTextBox1);
404         this->Controls->Add(this->button1);
405         this->Name = L"SDHGUI";
406         this->Text = L"SDH Adaptive Gripping";
407         this->FormClosing += gcnew System::Windows::Forms::↩
               FormClosingEventHandler(this, &SDHGUI::shutDown);
408         (cli::safe_cast<System::ComponentModel::ISupportInitialize^  ↩
               >(this->trackBar_changeThreshhold))->EndInit();
409         this->ResumeLayout(false);
410         this->PerformLayout();
411 
412     }
413 #pragma endregion
414 
415 
416 //Runs in a separate thread!
417 public: void GUIPollSensors(){
418 
419     while (running){
420     //  try{
421         maxPressureValue = sdhbackend->getMaxPressure();
422         Sleep(20);
423     //  } catch (Exception ^e){
424     //  }
425 
426     }
427   }
428   private: System::Void GUIConnect(System::Object^  sender, System↩
       ::EventArgs^  e) {
429         sdhbackend->connect();
430         sensorThread = gcnew Thread(gcnew ThreadStart(this, &↩
               SDHGUI::GUIPollSensors ));
431         sensorThread->Start();
432       }
433   private: System::Void GUIOpenHand(System::Object^  sender, System↩
       ::EventArgs^  e) {
434         sdhbackend->openHandStart();
435       }
436   private: System::Void printOutBuffer(System::Object^  sender, ↩
       System::EventArgs^  e) {
437         while(printBuffer->Count>0){
438           richTextBox1->AppendText(printBuffer[0]->ToString()+"\n")↩
                 ;
439           richTextBox1->ScrollToCaret();
440           printBuffer->RemoveAt(0);
441         }
442       }
443 
444 
```

```
445
446 private: System::Void getMaxPressure(System::Object^  sender, ↩
       System::EventArgs^  e) {
447
448        label_maxPressure->Text = maxPressureValue.ToString();
449        pb_0->Value = (int)(sdhbackend->sensorMax[0]);
450        pb_1->Value = (int)(sdhbackend->sensorMax[1]);
451        pb_2->Value = (int)(sdhbackend->sensorMax[2]);
452        pb_3->Value = (int)(sdhbackend->sensorMax[3]);
453        pb_4->Value = (int)(sdhbackend->sensorMax[4]);
454        pb_5->Value = (int)(sdhbackend->sensorMax[5]);
455        label_state->Text = sdhbackend->state;
456
457
458      }
459
460
461 private: System::Void shutDown(System::Object^  sender, System::↩
       Windows::Forms::FormClosingEventArgs^  e) {
462        running = false;
463        sdhbackend->shutDown();
464
465      }
466 private: System::Void StartSearchMode(System::Object^  sender, ↩
       System::EventArgs^  e) {
467        printout("GUI starting search!");
468         sdhbackend->graspSearchStart();
469      }
470 private: System::Void GUIManualHit(System::Object^  sender, System↩
       ::EventArgs^  e) {
471        sdhbackend->manualTriggerHit();
472      }
473 private: System::Void GUIClearLog(System::Object^  sender, System::↩
       EventArgs^  e) {
474         richTextBox1->Clear();
475      }
476 private: System::Void GUIChangeThresshold(System::Object^  sender, ↩
       System::EventArgs^  e) {
477        int iVal = trackBar_changeThreshhold->Value;
478        label_threshhold->Text = iVal.ToString();
479        sdhbackend->iPressureThreshhold = iVal;
480      }
481 private: System::Void GUICloseProximals(System::Object^  sender, ↩
       System::EventArgs^  e) {
482         Thread ^proximalsThread = gcnew Thread(gcnew ThreadStart(↩
            sdhbackend,&SDHBackend::closeProximalsStart));
483        proximalsThread->Start();
484        //sdhbackend->closeProximalsStart();
485      }
```

```
486 private: System::Void GUICloseDistals(System::Object^  sender, ↩
       System::EventArgs^  e) {
487         Thread ^distalsThread = gcnew Thread(gcnew ThreadStart(↩
               sdhbackend, &SDHBackend::closeDistalsStart));
488         distalsThread->Start();
489       }
490 private: System::Void GUIInitAutoGrasp(System::Object^  sender, ↩
       System::EventArgs^  e) {
491         printout("Starting state machine");
492         Thread ^stateMachineThread = gcnew Thread(gcnew ThreadStart(↩
               sdhbackend, &SDHBackend::startStateMachine));
493         stateMachineThread->Start();
494       }
495 };
496
497
498 }
```

A.3 SDHBackend.h

```
1  /*
2   * SDHBackend.h
3   * Header file with prototypes for the SDHBackend class. Has ↩
        methods for communication with
4   * SDH hardware, as well as most of the application logic.
5   * SÃ¸lve Monteiro, (c) 2010
6   * */
7
8  #pragma once
9  #include "sdh.h"
10 #include "dsa.h"
11 #include "util.h"
12 #include "sdhlibrary_settings.h"
13 #include "basisdef.h"
14 #include "sdhoptions.h"
15 #include "SensorHit.h"
16 using namespace System;
17 using namespace std;
18 using namespace SDH;
19 using namespace System::Windows::Forms;
20 using namespace System::Collections;
21 using namespace System::Threading;
22 using namespace System::IO;
23
24 ref class SDHBackend
25 {
26 public:
27
28   //Max angles in closing for proximal links and finger rotation
29   double alpha_max, beta_max;
30   int iGrabVelocity;
31   double fingerDelta;
32
33   //SDH Hand and sensor objects
34   cSDH *hand;
35   cDSA * dsa;
36
37   //The cSDHOptions object is a simplified way of setting params on↩
        the hardware
38   cSDHOptions * options;
39
40   //Printbuffer for outputting to GUI
41   ArrayList ^printBuffer;
42
43   //Some different threads needed in logic
44   Thread ^graspingThread, ^openHandThread, ^closeGraspThread, ^↩
```

```
        proximalsThread, ^distalsThread;
45
46      //Logic flags used in the state machine
47      bool noHits, running, moving;
48      String ^state;
49
50      //Stores the last registered hit on a new sensor
51      SensorHit ^lastHit;
52      int iMaxPressure;
53      int iPressureThreshhold;
54      ArrayList ^sensorMax;
55
56      //Global collection of sensor hits
57      ArrayList ^ sensorHits;
58
59      //Flags for halting fingers from closing
60      bool inhibitAxis1, inhibitAxis3, inhibitAxis5, proximalsDetected;
61      bool isFullAuto;
62      int activeAxis;
63
64      //File-writing output
65      TextWriter ^tw;
66
67      //Constructor
68      SDHBackend(void);
69
70      //Initialise method. This is called immediately after constructor
71      void init(ArrayList ^printBuffer);
72
73      //Attempts to connect to hardware. Prints error message if it ↩
            fails.
74      void connect();
75
76      //Print method. Prints to GUI output window
77      void printout(Object ^o);
78
79      //Start the open-hand thread.
80      void openHandStart();
81
82      //The open-hand thread runs this method
83      void openHand();
84
85      //Print sensor info to output window
86      void printSensors();
87
88      //Get the global max registered pressure
89      int getMaxPressure();
90
91      //Shut down method, closes connections
```

```
92    void shutDown();
93
94    //Start the grasp-search thread
95    void graspSearchStart();
96
97    //Method for the grasp-search thread
98    void graspSearch();
99
100   //Trigger a manual hit, for debugging
101   void manualTriggerHit();
102
103   //Semi-automatic method, closes remaining fingers
104   void closeGrasp();
105
106   //Clears target angles from the SDH hardware and sets them to the↩
          current value
107   //This helps against jerks when stopping/starting
108   void clearTargets();
109
110   //Start the close-proximals thread
111   void closeProximalsStart();
112
113   //Method for the close-proximals thread
114   void closeProximals();
115
116   //Start the close-distals thread
117   void closeDistalsStart();
118
119   //Method for the close-distals thread
120   void closeDistals();
121
122   //Test if centroid is within reach, and/or if we need finger ↩
          rotation
123   void calculateCentroid();
124
125   //Start the finger rotation
126   void rotateFingersStart();
127
128   //Start the fully automated search and grasp
129   void startStateMachine();
130
131   //Get how many fingers have had contacts on them
132   int getContactingFingers();
133
134   //Get the distance from the palm of the hand the contact occured
135   int getHitDistalPosition(SensorHit ^sensorHit);
136
137   //Get which proximal axis corresponds to the sensor that was hit
138   int getHandAxisFromSensor(int iSensor);
```

```
139
140     //Get which proximal axis corresponds to the sensor that was hit
141     int getHandAxisFromSensor(SensorHit ^sensorHit);
142
143     //Convert degrees to radians
144     double deg2rad(double deg);
145
146     //Get the x-position in millimeters from the center of the palm ↩
            of the contact point
147     int getGlobalXFromSensorHit(SensorHit ^sensorHit);
148
149     //Get the y-position in millimeters from the center of the palm ↩
            of the contact point
150     int getGlobalYFromSensorHit(SensorHit ^sensorHit);
151
152 };
```

A.4 SDHBackend.cpp

```
1  /* SDHBackend.cpp
2   * Source file for the SDHBackend class.Has methods for ↩
       communication with
3   * SDH hardware, as well as most of the application logic.
4   * SÃ¸lve Monteiro, (c) 2010
5   * */
6
7  #include "StdAfx.h"
8  #include "SDHBackend.h"
9
10
11
12 SDHBackend::SDHBackend(void)
13 {
14 }
15
16 void SDHBackend::printout(Object ^o){
17   printBuffer->Add(o);
18 }
19
20 void SDHBackend::init(ArrayList ^printBuffer){
21   SDHBackend::printBuffer = printBuffer;
22   printout("Backend initialised");
23   options = new cSDHOptions();
24   running = true;
25   isFullAuto=false;
26   iPressureThreshhold = 20;
27   alpha_max = 5.0;
28   beta_max = 90.0;
29   fingerDelta = 60; //Default value for rotation
30   iGrabVelocity = 5;
31   sensorHits= gcnew ArrayList();
32   String ^time = (DateTime::Now).ToString();
33   time = time->Replace( : , . );
34   time = time->Replace(    , . );
35   printout("Time: "+time);
36   tw = gcnew StreamWriter("data/"+time+".txt");
37   printout("Created file data/"+time+".txt");
38
39   //tw->WriteLine("Initialising textwriter");
40   //tw->Close();
41   sensorMax = gcnew ArrayList();
42   for(int i=0;i<6;i++)
43     sensorMax->Add(0);
44 }
45
```

```
46 double SDHBackend::deg2rad(double rad){
47   return rad*180.0/Math::PI;
48 }
49
50 void SDHBackend::connect(){
51   printout("Connecting to hand");
52   if(hand!=NULL){
53     if(hand->IsOpen()){
54       printout("Error: Connection to hand already open!");
55       return;
56     }
57   }
58   try{
59     hand = new cSDH();
60     hand->OpenRS232( options->sdhport, options->rs232_baudrate, ↩
          options->timeout, options->sdh_rs_device );
61     if(hand->IsOpen()) printout("Connected to SDH hardware!");
62
63     dsa = new cDSA(0,1,options->dsa_rs_device);
64     dsa->SetFramerate( 1);
65     /*
66     printout(dsa->GetControllerInfo().ToString());
67     printout(dsa->GetSensorInfo());
68     printout(dsa->GetMatrixInfo(options->matrixinfo));
69     */
70
71   } catch (Exception ^ex){
72     printout("Error connecting to SDH hardware: "+ex);
73   }
74 }
75
76 void SDHBackend::clearTargets(){
77
78   try{
79     std::vector<double> angles = hand->GetAxisActualAngle(hand->↩
          all_axes);
80     hand->SetAxisTargetAngle(hand->all_axes,angles);
81   }catch (Exception ^e){
82     printout("Error: "+e->ToString());
83   }
84 }
85
86 void SDHBackend::openHandStart(){
87   openHandThread = gcnew Thread(gcnew ThreadStart(this, &SDHBackend↩
       ::openHand));
88   openHandThread->Start();
89 }
90
91 void SDHBackend::openHand(){
```

```
92    fingerDelta=60.0;
93    if(hand==NULL){
94      printout("Error: connection to SDH not estabelished");
95      return;
96    }
97    if(!hand->IsOpen()){
98      printout("Error: connection to SDH not estabelished");
99      return;
100   }
101
102   //Clear all hits, starting over
103   sensorHits->Clear();
104
105   printout("Opening hand!");
106   //This can be a bit faster
107   hand->SetAxisTargetVelocity(hand->All,40.0);
108
109   //Rotation
110   hand->SetAxisTargetAngle(0,fingerDelta);
111
112   //Proximal angles : -70 deg, distal: 0
113   hand->SetAxisTargetAngle(1,-70.0);
114   hand->SetAxisTargetAngle(2,0.0);
115   hand->SetAxisTargetAngle(3,-70.0);
116   hand->SetAxisTargetAngle(4,0.0);
117   hand->SetAxisTargetAngle(5,-70.0);
118   hand->SetAxisTargetAngle(6,0.0);
119
120   //Do the movement
121
122   try{
123     moving = true;
124     hand->MoveHand();
125     moving = false;
126   } catch (Exception ^e){
127     hand->Stop();
128     moving = false;
129     printout("Error moving hand: "+e->ToString());
130     return;
131   }
132
133   //Disable the motors
134   hand->SetFingerEnable(hand->All,false);
135
136   //Kill all threads
137 /*  try{
138     graspingThread->Abort();
139   }catch (Exception ^){
140   }*/
```

```
141   try{
142     openHandThread->Abort();
143   }catch (Exception ^){
144   }
145   try{
146     closeGraspThread->Abort();
147   }catch (Exception ^){
148   }
149   try{
150     proximalsThread->Abort();
151   }catch (Exception ^){
152   }
153   try{
154     distalsThread->Abort();
155   }catch (Exception ^){
156   }
157   //printout("Hand opening complete! Motors disabled");
158 }
159
160 void SDHBackend::printSensors(){
161   dsa->UpdateFrame();
162   cDSA::sMatrixInfo matrixInfo[6];
163   cDSA::tTexel texel;
164   printout("Sensorpolling initialised");
165   for(int m=0;m<6;m++){
166     matrixInfo[m] = dsa->GetMatrixInfo(m);
167     for(int y=0; y<matrixInfo[m].cells_y;y++){
168       for(int x=0; x<matrixInfo[m].cells_x;x++){
169         texel = dsa->GetTexel(m,x,y);
170         printout("("+m+","+x+","+y+"): "+(int)texel);
171       }
172     }
173   }
174 }
175
176 int SDHBackend::getMaxPressure(){
177   String ^dataLogStr="";
178   int iSensorMax; //Max per sensor
179   if(!running) return 0;
180   try{
181     dsa->UpdateFrame();
182   } catch(Exception ^e){
183     printout("Error reading pressure: "+e->ToString());
184     return 0;
185   }
186   cDSA::sContactInfo contactInfo;
187   cDSA::sMatrixInfo matrixInfo[6];
188   int iMax = dsa->GetTexel(0,0,0);
189   int iVal;
```

```
190   cDSA::tTexel texel;
191   for(int m=0;m<6;m++){
192     iSensorMax=0;
193     contactInfo = dsa->GetContactInfo(m);
194     /*
195     printout("Contactinfo for "+m+" (f, a, x, y) : " +contactInfo.force+" "
196       +contactInfo.area+" "
197       +contactInfo.cog_x+" "
198       +contactInfo.cog_y);
199     */
200     //printout("Pressure Point "+m+": " +dsa->GetContactArea(m));
201     matrixInfo[m] = dsa->GetMatrixInfo(m);
202     for(int y=0; y<matrixInfo[m].cells_y;y++){
203       for(int x=0; x<matrixInfo[m].cells_x;x++){
204         texel = dsa->GetTexel(m,x,y);
205         iVal = (int)texel;
206         if(iVal>iSensorMax) iSensorMax=iVal;
207         if(iVal>iMax) iMax = iVal;
208         if(iVal>iPressureThreshhold){
209
210           if(noHits){   //Dont bother printing if we re already stopped
211             //Lets only count this hit if we dont have any hits on the same sensor from before!
212             int iHitCount = sensorHits->Count;
213             bool hasHitAlready = false;
214             for(int i=0;i<iHitCount;i++){
215               SensorHit ^hit = (SensorHit ^)sensorHits[i];
216               if(hit->sensor == m){
217                 hasHitAlready= true;
218               }
219             }
220             if(!hasHitAlready){
221               printout("Pressure threshhold surpassed: " +iVal+" vs"+iPressureThreshhold);
222               printout("Hit was in sensor "+m+" x:"+x+" y:"+y);
223
224               noHits=false;
225               lastHit = gcnew SensorHit(m,x,y,iVal);
226               //printout("X comp:"+getGlobalXFromSensorHit(lastHit));
227
228               sensorHits->Add(gcnew SensorHit(m,x,y,iVal));
229             }
230           }
231         }
232
233       }
```

```
234       }
235       dataLogStr+=iSensorMax+" ";
236       sensorMax[m] = iSensorMax;
237     }
238     tw->WriteLine(dataLogStr);
239     //printout("Max pressure: "+iMax);
240     iMaxPressure = iMax;
241
242
243
244     return iMax;
245   }
246
247   void SDHBackend::shutDown(){
248     running=false;
249     if(dsa!=NULL) dsa->Close();
250     if(hand!=NULL){
251       hand->SetFingerEnable(hand->All,false);
252       hand->Close();
253     }
254     tw->Close();
255   }
256
257   //Initialise separate thread for grasp-searching - clears previous ↩
        hits!
258   void SDHBackend::graspSearchStart(){
259     if(hand==NULL){
260       printout("Error: connection to SDH not estabelished");
261       return;
262     }
263     if(!hand->IsOpen()){
264       printout("Error: connection to SDH not estabelished");
265       return;
266     }
267     sensorHits->Clear();
268     graspingThread = gcnew Thread(gcnew ThreadStart(this, &SDHBackend↩
          ::graspSearch ));
269     noHits=true;
270     graspingThread->Start();
271   }
272
273   void SDHBackend::graspSearch(){
274     printout("Grasp Search commenced - old hits cleared!");
275     //Open hand in seq mode (since we re already in a separate thread↩
          )
276     openHand();
277     sensorHits->Clear();
278     printout("Initializing inwards search");
279     closeGraspThread = gcnew Thread(gcnew ThreadStart(this, &↩
```

```
        SDHBackend::closeGrasp));
280   closeGraspThread->Start();
281   moving = true;
282   while(noHits && running){
283     if(!moving) {
284       printout("Movement complete");
285       return;
286     }
287   }
288   closeGraspThread->Abort();
289   printout("Hit detected, halting search!");
290
291   hand->Stop();
292   clearTargets();
293   //hand->SetFingerEnable(hand->All,false); //Lets leave the motors↩
        on since we re gripping
294   calculateCentroid();
295
296 }
297 //Closes grasp for ALL fingers!
298 void SDHBackend::closeGrasp(){
299   //Max angle for proximal joints. No egg will fit!
300   hand->SetAxisTargetVelocity(hand->All,iGrabVelocity); //We want ↩
        this to be quite slow
301   hand->SetAxisTargetAngle(0,fingerDelta); //Lets just keep 60 deg ↩
        (for a likesided triangle)
302
303   //Proximal joints target: alpha_max, distal: 0
304   hand->SetAxisTargetAngle(1,alpha_max);
305   hand->SetAxisTargetAngle(2,0.0);
306   hand->SetAxisTargetAngle(3,alpha_max);
307   hand->SetAxisTargetAngle(4,0.0);
308   hand->SetAxisTargetAngle(5,alpha_max);
309   hand->SetAxisTargetAngle(6,0.0);
310
311   hand->MoveHand();
312
313   // ^^ is blocking, so if we reach here, we ve reached alpha max ↩
        without hits
314   // so lets power down motors
315   moving = false;
316   hand->SetFingerEnable(hand->All,false);
317   printout("Search returned no results");
318   state = "STATE_NOHITS";
319 }
320
321 int SDHBackend::getContactingFingers(){
322   int finger0, finger1, finger2=0;
323   for(int i =0; i<sensorHits->Count;i++){
```

```
324       SensorHit ^sh = (SensorHit ^)(sensorHits[i]);
325       int sensor = sh->sensor;
326       if(sensor==0 || sensor==1) finger0=1;
327       if(sensor==2 || sensor==3) finger1=1;
328       if(sensor==4 || sensor==5) finger2=1;
329     }
330
331     return finger0 + finger1 + finger2;
332
333   }
334
335   void SDHBackend::manualTriggerHit(){
336     noHits = false;
337   }
338
339   void SDHBackend::closeProximalsStart(){
340     proximalsThread = gcnew Thread(gcnew ThreadStart(this, &↩
          SDHBackend::closeProximals));
341     proximalsThread->Start();
342     noHits=true;
343     moving = true;
344     proximalsDetected= false;
345     bool proximalsRunning = true;
346     while(proximalsRunning && running && noHits){
347       if(!moving && !isFullAuto) return;
348
349     }
350     proximalsThread->Abort();
351     printout("Hit detected, halting proximals!");
352     hand->Stop();
353     //hand->SetFingerEnable(hand->All,false); //Leave motors on
354     clearTargets();
355
356   }
357
358   void SDHBackend::closeProximals(){
359     clearTargets();
360     inhibitAxis1, inhibitAxis3, inhibitAxis5 = false;
361     SensorHit ^hit;
362     int sensor;
363     int iHitCount = sensorHits->Count;
364     printout("Closing proximals: have "+iHitCount+" hits to respect:"↩
          );
365     for(int i=0;i<iHitCount;i++){
366       hit = (SensorHit^)sensorHits[i];
367       printout(hit->getString());
368       sensor = hit->sensor;
369       if(sensor == 0 || sensor == 1){
370         inhibitAxis1 = true;
```

```
371         printout("Axis 1 inhibited!");
372       }
373       if(sensor == 2 || sensor == 3){
374         inhibitAxis3 = true;
375         printout("Axis 3 inhibited!");
376       }
377       if(sensor == 4 || sensor == 5){
378         inhibitAxis5 = true;
379         printout("Axis 5 inhibited!");
380       }
381     }
382     /*
383     We want to move the fingers that dont have any hits on them.
384     axis 1 has sensors 01
385     axis 3 has sensors 23
386     axis 5 has sensors 45
387     */
388     try{
389       hand->SetAxisTargetVelocity(hand->All,iGrabVelocity);
390     } catch (Exception ^e){
391       printout("Error setting velocity: "+e->ToString());
392       return;
393     }
394     activeAxis=0;
395     if(!inhibitAxis1){
396       activeAxis++;
397       hand->SetAxisTargetAngle(1,alpha_max);
398     }
399     if(!inhibitAxis3){
400       activeAxis++;
401       hand->SetAxisTargetAngle(3,alpha_max);
402     }
403     if(!inhibitAxis5){
404       activeAxis++;
405       hand->SetAxisTargetAngle(5,alpha_max);
406     }
407
408     proximalsDetected= true;
409     try{
410       hand->MoveHand();
411       state = "STATE_PROXIMALSCOMPLETE";
412     } catch (Exception ^e){
413       printout("Error closing proximals: "+e->ToString());
414     }
415     moving = false;
416     printout("Proximal closing complete, active axis: "+activeAxis);
417     //hand->SetFingerEnable(hand->All,false);
418     if(activeAxis==0){
419       state="STATE_PROXIMALSCOMPLETE";
```

```
420     }
421
422 }
423
424 void SDHBackend::closeDistalsStart(){
425   printout("Starting to close distals");
426   distalsThread = gcnew Thread(gcnew ThreadStart(this, &SDHBackend↩
        ::closeDistals));
427   distalsThread->Start();
428   noHits=true;
429   moving = true;
430   while(noHits && running){
431     if(!moving) return;
432   }
433   distalsThread->Abort();
434   printout("Hit detected, halting distals!");
435   hand->Stop();
436   //hand->SetFingerEnable(hand->All,false); //Leave motors on
437   clearTargets();
438
439 }
440
441 void SDHBackend::closeDistals(){
442   clearTargets();
443   bool inhibitAxis2, inhibitAxis4, inhibitAxis6 = false;
444   SensorHit ^hit;
445   int sensor;
446   int iHitCount = sensorHits->Count;
447   printout("Closing distals: have "+iHitCount+" hits to respect:");
448   for(int i=0;i<iHitCount;i++){
449     hit = (SensorHit^)sensorHits[i];
450     printout(hit->getString());
451     sensor = hit->sensor;
452     if(sensor == 1) inhibitAxis2 = true;
453     if(sensor == 3) inhibitAxis4 = true;
454     if(sensor == 5) inhibitAxis6 = true;
455   }
456   /*
457   We want to move the distals that dont have any hits on them.
458   axis 2 has sensors 1
459   axis 4 has sensors 3
460   axis 6 has sensors 5
461   */
462   hand->SetAxisTargetVelocity(hand->All,iGrabVelocity);
463   activeAxis=0;
464   if(!inhibitAxis2){
465     activeAxis++;
466     hand->SetAxisTargetAngle(2,beta_max);
467   }
```

```
468     if(!inhibitAxis4){
469       activeAxis++;
470       hand->SetAxisTargetAngle(4,beta_max);
471     }
472     if(!inhibitAxis6){
473       activeAxis++;
474       hand->SetAxisTargetAngle(6,beta_max);
475     }
476 
477     try{
478       hand->MoveHand();
479     } catch (Exception ^e){
480       printout("Error moving distals: "+e->ToString());
481       return;
482     }
483     //hand->SetFingerEnable(hand->All,false);
484     printout("Distal closing complete");
485     state = "STATE_DISTALSCOMPLETE";
486     moving = false;
487 
488   }
489 
490   int SDHBackend::getHandAxisFromSensor(int sensor){
491     int axis;
492     if(sensor==0 || sensor == 1)
493       axis = 1;
494     if(sensor==2 || sensor ==3)
495       axis = 3;
496     if(sensor==4 || sensor ==5)
497       axis = 5;
498     return axis;
499 
500   }
501 
502   int SDHBackend::getHandAxisFromSensor(SensorHit ^sensorHit){
503     return getHandAxisFromSensor(sensorHit->sensor);
504   }
505 
506   int SDHBackend::getHitDistalPosition(SensorHit ^sensorHit){
507     int sensor = sensorHit->sensor;
508     int y = sensorHit->y;
509 
510     //y=0 -> distal direction
511     //double angle = angles[axis];
512     int isDistal = sensor%2;  //proximal; 0, distal: 1;
513 
514     cDSA::sMatrixInfo matrixInfo = dsa->GetMatrixInfo(sensor);
515     int iSensorLength = matrixInfo.cells_y;
516     //TODO: find detailed dims
```

```
517   int proximalOffset=20;  //From palm to proximal
518   int distalOffset=100;   //From palm to beginning of distal
519   int sensorOffset= (iSensorLength-y)*0.75;
520   int iHitLength= (proximalOffset+sensorOffset)*(1-isDistal)+(↩
        distalOffset+sensorOffset)*isDistal;
521   return iHitLength;
522
523 }
524
525 int SDHBackend::getGlobalXFromSensorHit(SensorHit ^sensorHit){
526   int axis = getHandAxisFromSensor(sensorHit);
527   int dist = getHitDistalPosition(sensorHit);
528   double alpha = hand->GetAxisActualAngle(axis);
529   double beta = hand->GetAxisActualAngle(0);
530   int x=0;
531   int h=0;
532   if(axis==1){  //- y
533     x = dist*cos(deg2rad(90+alpha))*cos(deg2rad(beta));
534   }
535   else if(axis==3){ //Thumb
536     x = -dist*cos(deg2rad(90+alpha));
537
538   }
539   else if(axis==5){ // +y
540     x = dist*cos(deg2rad(90+alpha))*cos(deg2rad(beta));
541   }
542   printout("X comp axis: "+axis+" distance: "+dist+" alpha: "+alpha↩
        +" xcomp:"+x);
543   return x;
544 }
545
546 int SDHBackend::getGlobalYFromSensorHit(SensorHit ^sensorHit){
547   int axis = getHandAxisFromSensor(sensorHit);
548   int dist = getHitDistalPosition(sensorHit);
549   double alpha = hand->GetAxisActualAngle(axis);
550   double beta = hand->GetAxisActualAngle(0);
551   int y=0;
552   int h=0;
553   if(axis==1){  //- y
554     y = -dist*cos(deg2rad(90+alpha))*sin(deg2rad(beta));
555
556   }
557   else if(axis==3){ //Thumb
558     y = 0;
559   }
560   else if(axis==5){ // +y
561     y = dist*cos(deg2rad(90+alpha))*sin(deg2rad(beta));
562   }
563   printout("Y comp axis: "+axis+" distance: "+dist+" alpha: "+alpha↩
```

```
      +" ycomp:"+y);
564   return y;
565 }
566
567 void SDHBackend::calculateCentroid(){
568   printout("Calculating Centroid");
569   vector<double> angles = hand->GetAxisActualAngle(hand->↩
        all_real_axes);
570   int x = lastHit->x;
571   int y = lastHit->y;
572   int sensor = lastHit->sensor;
573   cDSA::sMatrixInfo matrixInfo = dsa->GetMatrixInfo(sensor);
574
575   int iHitLength = getHitDistalPosition(lastHit);
576
577
578   //Test if hit x is close to 0 or cells_x.
579   printout("Testing hit lateral location: hit was in "+x+", max is ↩
        "+matrixInfo.cells_x);
580   if(x==0 || x+1>=matrixInfo.cells_x){
581     state = "STATE_NEEDROTATE";
582     return;
583   }
584
585   //We re good , lets get on with the proximals!
586     //Tell state machine to start proximals STATUS_CENTROIDOK
587     state = "STATE_CENTROIDOK";
588
589
590 }
591
592 //Starts a thread that attempts to improve a detected contact. This↩
      one should not rely on collisions
593 //to be halted, but we need to make sure we don t go beyond max/min↩
      of fingers.
594 void SDHBackend::rotateFingersStart(){
595   printout("Commencing rotation");
596   double dAngleMin = 0;
597   double dAngleMax = 90.0;
598
599   //Assuming we re still usomg lastHit as the active hit.
600   //rotation = K*distance
601   int x = lastHit->x;
602   int iHitLength = getHitDistalPosition(lastHit);
603   double rot = -(10.0-(10.0*iHitLength)/160);
604   int axis=1;
605   double currRot = hand->GetAxisActualAngle(0);
606   int sensor = lastHit->sensor;
607   if(sensor == 0 || sensor == 1){ //axis 1
```

```
608       if(x != 0) rot*= -1;
609       axis = 1;
610     }else if(sensor == 4 || sensor == 5){ //axis 5
611       if(x==0) rot *= -1;
612       axis = 5;
613     }else{
614       printout("Hit was on the thumb (sensor "+sensor+")");
615       state = "STATE_CENTROIDOK";
616       return;
617     }
618     //Less rotation the larger hitLength is
619     double totalRot = currRot+rot;
620
621     if(totalRot>dAngleMax) totalRot = dAngleMax;
622     if(totalRot<dAngleMin) totalRot = dAngleMin;
623     hand->SetAxisTargetVelocity(0,10.0);
624     hand->SetAxisTargetVelocity(axis,10.0);
625     hand->SetAxisTargetAngle(axis,hand->GetAxisTargetAngle(axis)-3.0)↩
          ; // move out
626     hand->MoveHand();
627     sensorHits->Clear();
628     hand->SetAxisTargetAngle(0,totalRot);                    //rotate
629     hand->MoveHand();
630     hand->SetAxisTargetAngle(axis,hand->GetAxisTargetAngle(axis)+3.0)↩
          ; //move in
631     hand->MoveHand();
632     hand->SetAxisTargetVelocity(hand->All,5.0);
633     printout("Rotation complete");
634     state = "STATE_CENTROIDOK";
635     //Open up, 1 deg?, rotate, close 1 deg, state-> STATE_CENTROIDOK
636
637   }
638
639   void SDHBackend::startStateMachine(){
640     //lets make a blocking - single run, triggered by a call from ↩
          daddy
641
642     //Start with the grasp search?
643     state="STATE_SEARCHING";
644     sensorHits->Clear();
645     graspSearchStart();
646     while(state=="STATE_SEARCHING"){
647       //wait
648     }
649
650     //check the status set by calcCentroid
651     if(state=="STATE_OUTOFREACH"){
652       printout("State: out of reach");
653
```

```
654       //Abort grasp
655       //return message to daddy
656     }
657     if(state=="STATE_NEEDROTATE"){
658       printout("state: need some rotating");
659       rotateFingersStart();
660       //rotate Fingers according to lastHit - lets make this blocking↩
              !
661     }
662     if(state=="STATE_CENTROIDOK"){  //assume this is correct
663       printout("state: centroid is within reach");
664       closeProximalsStart();
665       //closeProximalsStart -  blocking
666       while(state != "STATE_PROXIMALSCOMPLETE"){
667         //-> closeProximalsStart - blocking
668         Sleep(300);
669         closeProximalsStart();
670       }
671     }
672     if(state=="STATE_PROXIMALSCOMPLETE"){
673       printout("state: proximals are done closing");
674       //Test how many fingers have reahced their targets
675
676       if(getContactingFingers()<2){
677         state = "STATE_OUTOFREACH";
678         int x = getGlobalXFromSensorHit(lastHit);
679         int y = getGlobalYFromSensorHit(lastHit);
680
681         printout("Object is out of reach! Need to move gripper (x,y) ↩
              : ("+x+","+y+")");
682         //calculate needed movement
683         return;
684       }
685       closeDistalsStart(); // - blocking
686       while(state!="STATE_DISTALSCOMPLETE"){
687         //closeDistals - blocking
688         Sleep(300);
689         closeDistalsStart();
690       }
691
692     }
693
694     if(state=="STATE_NOHITS"){
695       printout("state: Grasp found nothing");
696       //return message to daddy: no hits
697     }
698
699     if(state=="STATE_FIXROTATION"){
700       printout("state: Found target, but need gripper rotation!");
```

```
701     }
702
703     if(state=="STATE_DISTALSCOMPLETE"){
704       printout("state: Grip is complete!");
705       //hand->SetFingerEnable(hand->All,false);
706       //return message to daddy: grip complete
707     }
708
709
710   }
```

A.5 SensorHit.h

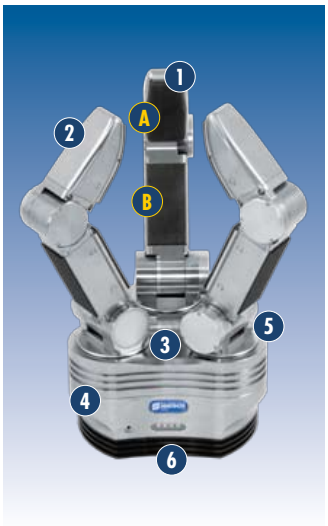
```
1  /* SensorHit.h
2   * Header file for SensorHit class
3   * SensorHit is an object representing a contact on a sensor.
4   * Sensor-ID, pressure, x and y position and finger configuration
5   * is stored in this object.
6   * SÃ¸lve Monteiro, (c) 2010
7   * */
8
9  #pragma once
10 using namespace System::Collections;
11 ref class SensorHit
12 {
13 public:
14   int sensor, x, y, pressure;
15   ArrayList ^proximalAngles;
16   SensorHit(int sensor, int x, int y, int pressure,ArrayList ^ ↩
       proximalAngles);
17   SensorHit(int sensor, int x, int y, int pressure);
18   System::String ^getString();
19
20 };
```

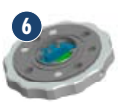
A.6 SensorHit.cpp

```
1  /* SensorHit.cpp
2   * Source file for SensorHit class
3   * SensorHit is an object representing a contact on a sensor.
4   * Sensor-ID, pressure, x and y position and finger configuration
5   * is stored in this object.
6   * SÃ¸lve Monteiro, (c) 2010
7   * */
8
9  #include "StdAfx.h"
10 #include "SensorHit.h"
11
12
13 SensorHit::SensorHit(int sensor, int x, int y, int pressure)
14 {
15   SensorHit::sensor = sensor;
16   SensorHit::x = x;
17   SensorHit::y = y;
18   SensorHit::pressure = pressure;
19
20 }
21
22 SensorHit::SensorHit(int sensor, int x, int y, int pressure, ↩
      ArrayList^ proximalAngles){
23   SensorHit::proximalAngles = proximalAngles;
24   SensorHit(sensor,x,y,pressure);
25 }
26
27 System::String ^SensorHit::getString(){
28   return "(s: "+SensorHit::sensor+", x: "+SensorHit::x+", y: "+↩
        SensorHit::y+", p: "+SensorHit::pressure+")";
29 }
```

SDH

Gripping Hands · 3-Finger Gripping Hand



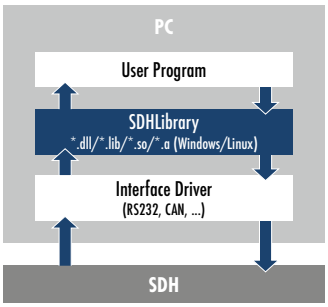


1. **Tactile Sensor Array** in each finger
2. **Modular Fingers** 2 active dof each with integrated drives
3. **Pivoting Joints** 2 fingers contrarywise 1 active dof
4. **Body** fully integrated control
5. **Modular Joints** distal joint (middle of finger) proximal joint (finger root)
6. **Manipulator Interface** EN ISO 9404-1-50 Industrial Standard

SCHUNK Dextrous Hand SDH

Servo-electric 3-Finger Gripping Hand SDH Universal gripping instrument

The revised, multiple-section 3-finger SDH gripping hand enables the form-fit and force-fit gripping of a wide range of objects and can be used both in areas of service robotics and in industrial applications. Since two fingers can change their positions, the SDH is suitable for many different gripping scenarios and features high flexibility in regard to form, size and position of the objects to be gripped.



Technical data

Description		SAH
Overall Length	[mm]	253
Finger Length	[mm]	155
Finger Base Distance	[mm]	66
Factor to Human Hand		1.4 : 1
Total Weight	[kg]	1.95
IP class	[IP]	65
Number of Fingers		3
DOF (active) total		7
DOF (active) per Finger		2
DOF (active) 2-Finger-Pivot.		1
Angular Joint Speed max.	[°/sec]	210
Moment (Proximal Joint)	[Nm]	2.1
Moment (Distal Joint)	[Nm]	1.4
Accuracy	[°]	0.011
Voltage	[V]	24
Currency (grasping object)	[A]	5
Communication:		CAN / Ethernet / RS232
Absolute Encoders		7
Tactile Sensor Arrays		6

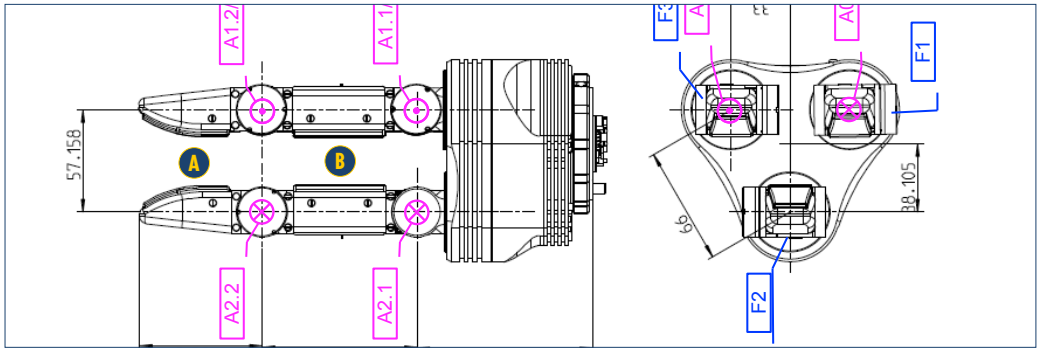

Parallel grip


3-finger central grip - comprehensive


3-finger cylinder grip


Large 3-finger centric grip parallel

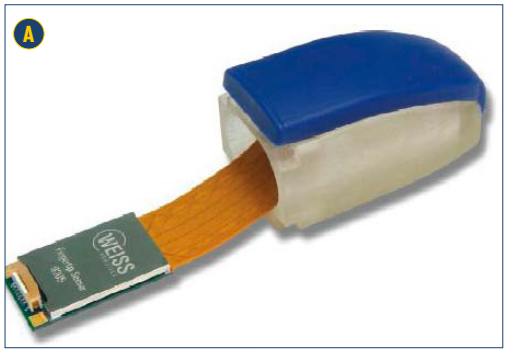
Main views



⊙ indicates „thumb pointing out of paper" (tip of arrow)

⊗ indicates „thumb pointing into paper" (back of arrow)

The tactile sensor system DSA 9200 is an optional enhancement of SCHUNK's multi-finger SDH gripping hand. As a force sensitive measurement system it enables the acquisition of a two-dimensional contact force profile allowing its usage in a variety of applications. Based on a resistive measurement principle it offers a high spatial resolution in a robust design. The modular sensor system can be integrated in to the hand completely. It consists of exchangeable transducer modules in the fingertips and proximal limbs, as well as of a sensor controller integrated in to the hand's base. The transducers are water-tight, so that they can function reliably under adverse conditions. The integrated signal processing electronics and the sampling method used allow the output of high-quality tactile contact information. The defined flexibility and adhesive characteristics of the surfaces allow the safe gripping also of difficult parts. The measurements can be processed by the hand control or an external processing unit, like a PC. Libraries and example programs are available. For visualizing the contact force profile the Software DSA-Explorer is recommended, which is available at request.

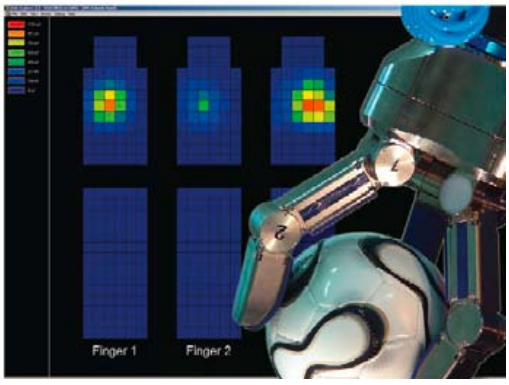


Fingertip transducer

Proximal limb transducer

Areas of application

Control of gripping force, Grasp analysis, Grasp Optimization, Tactile Object recognition



Technical data

Transducer Module		
Number of reading points	[Texels]	14 x 6
Spatial resolution	[mm]	3.4
Measurement Range	[kPa]	200
Sample Rate	[fps]	230
Gripping Material		Silicone rubber
Sensory System		
Resolution Preassure Profile	[Bit]	12
Sample rate	[fps]	40
Communication		serial
Power Consumption	[W]	0.8