UNIVERSITY OF BIRMINGHAM

Year Three Group Studies: Teaching the Aldebaran Nao Robot to Use a Swing

Joe Allen, Chantal Birkinshaw, Stuart Bradley, Tom Crossland,
William Edmondson, William Etheridge, Vincent Fisher,
Jack Ford, Fred Grover, Owan Haughey, Christopher Hounsom,
Michael Jevon, Ashe Morgan, Joe Preece, Chloé Smith,
Chris Smith, Peter Suttie, Daniel Tyrer and Michael Wright.

Date: Spring 2015

School of Physics and Astronomy
University of Birmingham
Birmingham, B15 2TT

Preface

——————— The following was contributed by: Chloé Smith ———————

Why are Robots so Interesting?

"Robot", was given to the modern world by Czech native Karel Čapek. His 1921 play "R.U.R" (Rossum's Universal Robots) conveys a society whose economy is entirely reliant on these "*roboti*". Tired of the forced labour and endless servitude, the robots rebel and kill every human on earth - apart from one man who "works with his hands; like us". Realising their race will die out soon too as they have no means to create new robots, they task the one surviving human to search for a method; but he is unsuccessful. A bleak story, but the essence of a new part of popular culture was born.

Humans have evolved as a species to *learn*. Survival of the fittest - except it was not our fitness that kept us alive; but rather our intellect. So it is only natural that humans would want more intellect, more *knowledge*. But we are limited to how much knowledge we can hold in our organic brains. What if there was a machine that could find information, interpret it and then develop new technology or theories with said knowledge?

But on the other hand, our modern society is driven by money (or more specifically an economy which runs on money). People must be paid money for their labour to manufacture things which are then sold for... more money. But what if there was a machine that could manufacture these things for no payment? If there is a problem with the process the machine will not know what to do - it is not a human. What if it could be taught to take each situation separately and deal with it appropriately, and continue the manufacturing process? Cheap, guilt free labour.

Do you see the pattern? Humans will always strive to be better; and the path to that may be paved with robots.

Humans and Robots: The Inherent Difference

Look back at the example given of a robot "employed" in manufacturing. The robot can be taught to build whatever is necessary; but if a problem arises (say, its screwdriver breaks) it will not know *what to do*. So the human programs the robot to execute a number of tasks in order when a particular situation (or stimuli) arises. And here is the inherent difference. The robot *cannot learn*. A human will deduce independently what it could do in any situation given to them; but a robot can only do what the human told it to do in said situation. It is a subject that is being constantly deliberated, but one can consider this as a test of **intelligence**.

This somewhat subtle intricacy is nicely conveyed by this very group study. NAO, our friendly humanoid robot can do many things; he can walk, talk, recognise faces and even practice Tai Chi. But, when you put him on a child's swing and tell him to swing, he does not know what to do. This is because swinging requires (and you may not immediately realise this) a certain amount of what could be referred to as *instinct*. When a human swings, they regularly change the frequency of kicking their legs in and out (the driving frequency) to match the **natural frequency**. They swing with a greater amplitude each time as they reach resonance. The vestibular system (situated in your inner ear) communicates with your cerebral cortex; and you "sense" you are accelerating. So at the point you feel acceleration (i.e. the peak amplitude of the swing) you kick your legs out, or bring them in. But how can NAO do this? Even this over simplified description of how the human anatomy responds to such stimuli is impossible for our humanoid. But he does have a an accelerometer in his chest, and

an impressive 1.6GHz CPU inside his head. So perhaps some code could be written for NAO to execute the perfect motion at particular values from the accelerometer?

Our Place in Research

Back in 1998, Jette Randløv and Preben Alstrøm published an article where an "agent" (think of this as a virtual robot) tried to learn to ride a bicycle. Reinforcement learning (utilising the Sarsa(λ) algorithm) and a technique called "shaping" (where smaller, simpler tasks are learnt to build up to the more difficult one) were used [26]. And, even in 1998, with the power of mathematics and good code they were successful.

In 2009, Brenna Argall et al. published an article in "Robotics and Autonomous Systems" surveying the technique *Learning from Demonstration* (LfD) [25]. Put simply, LfD is a method where a "teacher" demonstrates a skill (or action) to the robot agent, and the robot repeats. This is contrasting to the example given above, where the robot learnt to ride the bicycle via trial and error; or through *experience* [25]. Much more recently, Tesca Fitzgerald and Andrea Thomaz describe in their article "Skill Demonstration Transfer for Learning from Demonstration" their working in finding a system where a robot can remember and repeat tasks learnt, even in unfamiliar environments [21]. This is a limitation of LfD, the agent will not necessarily realise the skill it learnt is applicable in more than the exact situation in which it learnt said skill [21].

There are enough examples of work of this variety to fill a whole new report - but these particular examples are all linked by **Machine Learning** (ML). ML is the study striving to create algorithms and code that can, to put it simply, *learn*. This group study did not successfully use ML itself to get NAO to swing, but a great foundation has been set for subsequent groups to do just that.

Of course there are ethical arguments to be had regarding this whole line of research. **Artificial Intelligence** may very well eventually be reached using ML. In which case we must re evaluate the whole example of using robots as a means of cheap, "guilt free" labour. Would it be, for the lack of a better word, *right* to use sentient beings as slaves? Of course it would not be. But for now let us just consider getting NAO to swing like a human.

Contents

Chapter 1

Introduction

——— The following was contributed by: Joe Preece ———

1.1 An Insight into Robotics

Since the ancient world, the concept of autonomous machines has fascinated engineers, artists, and authors alike. The idea of a machine that possesses the capability to replicate the intricate behaviours of humans, improve efficiency in the workplace, and to even think for itself has made for pioneering research, intriguing tales, and something that many children look forward to opening on Christmas morning.

Since the dawn of computational processing, these machines have become a reality. Robots now seem commonplace, especially in places such as the factory floor where they have replaced humans in order to reduce the risk of injury. Many variations of robots have been engineered over the years, in order to achieve a desired task most efficiently. Over the past few decades, the advances in the field of humanoid robots have been astronomical, and robots of this design have now become available commercially.

One of these humanoid robots is *Nao*, produced by the French robotics company *Aldebaran*. Nao, which stands at 58 centimetres and weighs 4.3 kilograms, is programmable in a number of languages, and is compatible with all major operating systems. This makes it incredibly popular with enthusiasts and beginners alike.

1.2 Motivations for this Project

The production of humanoid robots - and the subsequent advances in the technologies behind them - have led to projects that attempt to simulate human movements and behaviours. These range from the seemingly simple (walking, turning) to more advanced motions such as standing up from sitting, performing push-ups, and dancing. Swinging most definitely falls into the latter category, and is one of the key motivations as to why this motion was chosen as the central theme for this project; the challenges that would ensue were suitable for a group of twenty persons.

Mathematically, swinging may be understood by producing Lagrangians and Hamiltonians for simplified models, such as single, double, and even triple pendula. Physically, the models develop numerous complications. Another motivation for this project is for the group to overcome these challenges, and to understand how a robot might be able to swing. This would contribute to the plethora of projects that have already been undertaken on humanoid robots and Nao.

1.3 Desired Outcomes

Prior to properly starting the project, the desired outcomes were decided upon. First and foremost (after recommendation from our supervisor) it was decided that a variety of coding languages were to be used, in order to advance the knowledge of individuals within the group. Languages hypothesised were variations on C, including C++ and Python.

Ultimately, the outcome of the project was to understand the rudiments of both the mathematics and the physics behind swinging. To develop an understanding of the principles, three main project goals were decided upon: to make the robot swing whilst standing up; to make the robot swing whilst sitting down; to make the robot initiate either type of swinging from rest. This would require reviews of current literature, the creation of computational solvers, and specifically engineered hardware to assure that the robot would swing as naturally as possible.

N.B. The disk submitted alongside this report contains all the code files and videos referenced to throughout this report, in addition to user guides. The same material can be downloaded from the following url:

`https://mega.nz/#!QFAy2LwB!sU5TOHkah_4zPXxDXPgSlzKeKHmN83-4ajd2RCTq4v4`

Chapter 2

Theory

2.1 Mathematics and Physics of Swinging

——————— The following was contributed by: Stuart Bradley ———————

2.1.1 Lagrangian and Hamiltonian Mechanics

Due to the nature of the problems that we have been looking at through the course of the project, Newtonian mechanics would not have been most suitable for use in answering most of the problems we have faced. While it wouldn't give incorrect results, it would have made the mathematics overly complicated and most impractical to solve. Instead for the duration we have been mostly using Lagrangian and Hamiltonian Mechanics, these are alternative formulations of classical mechanics, which allows use to reach our answers by alternative routes that have advantages and disadvantages, depending on the problem in question.

2.1.1.1 Lagrangian Mechanics

Lagrangian mechanics is based on the principle of least action, which states that given a start and end point, the system will follow a path from start to finish that minimises the action, $\delta \mathrm{S} = 0$. The action, normally given the symbol S, is a function of the trajectory, here denoted by the Lagrangian $\mathcal{L}$.

$$S = \int_{\mathrm{t=a}}^{\mathrm{t=b}} \mathcal{L}\,\mathrm{d}t \tag{2.1.1}$$

Where,

$$\mathcal{L} = T - V \tag{2.1.2}$$

The Lagrangian is a function which is equal to the total kinetic energy T minus the total potential energy V, it is formed by a set of generalised coordinates, each coordinate corresponding to a degree of freedom in the system. Lagrangian mechanics allows us to pick an independent generalised coordinate set that completely defines the system, this means that symmetries and constraints can by used in the calculation that otherwise wouldn't have been applicable. This means that terms in Newtonian mechanics that would be difficult, like variable force terms, are not present in the equations.

Since the Lagrangian totally describes the system, it is possible to use it to calculate the equations of motion for the system by use of the Euler-Lagrange equation.

The Euler-Lagrange Equation:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}_i}\right) = \frac{\partial \mathcal{L}}{\partial q_i} \tag{2.1.3}$$

Where q_i is a generalised coordinate and $\dot{q}_i$ the generalised velocity. This is true for systems that involve conservative forces and no dissipative forces like friction. The output of using the Euler-Lagrange equation would be a single second order differential equation for each coordinate that can then be solved to give an equation of motion.

2.1.1.2 Hamiltonian Mechanics

Hamiltonian mechanics is another reformulation of classical mechanics, instead of defining our system by $\mathcal{L}(q_i, \dot{q}_i, t)$, we are now using canonical coordinates to describe the system $\mathcal{H}(q_i, p_i, t)$, where p_i is the canonical momenta. Canonical coordinates are a set of coordinates that can completely describe a physical system at any point in time. The Hamiltonian can be calculated from the Lagrangian by;

$$\mathcal{H}(q_i, p_i, t) = \sum_i \dot{q}_i p_i - \mathcal{L} = \sum_i \dot{q}_i \frac{\partial \mathcal{L}}{\partial \dot{q}_i} - \mathcal{L} \tag{2.1.4}$$

For a closed system, a system that doesn't exchange matter with surroundings or subject to an external force, the Hamiltonian is equal to the sum of the total kinetic and total potential energy. Also since changing to the canonical coordinates T is now a function only of p and V is a function only of q.

$$\mathcal{H}(q_i, p_i, t) = T + V \tag{2.1.5}$$

$$T = P^2/2m \tag{2.1.6}$$

Just as with Euler-Lagrangian equation for calculating the equations of motion for a problem using the Lagrangian formulation, there are a series of Hamilton's equations that produce two first order differential equations, as opposed to the single second order differential equation of the Euler-Lagrange equation.

$$\dot{p}_i = -\frac{\partial \mathcal{H}}{\partial q_i}, \dot{q}_i = \frac{\partial \mathcal{H}}{\partial p_i} \tag{2.1.7}$$

——— The following was contributed by: Michael Jevon ———

2.1.1.3 Lagrange Multipliers

When producing mathematical models of driven swinging motions, it was important to be able to investigate the effort required to drive the system: to do this, knowledge of the forces providing the motion is necessary. This was performed with the introduction of Lagrange multipliers, λ. This extension to the use of Lagrangian mechanics as described in section 2.1.1.1 allows the inspection of some forces as well as an alternative way to apply constraints. In the standard formulation of Lagrangian mechanics, constraints are applied to the system at the earliest point, reducing the degrees of freedom of the system to their minimum number and using them to create a set of generalised coordinates. When using multipliers, constraints are applied as separate equations, of the form

$$G_i(q_1, q_2, \ldots, q_n, t) = 0$$

which implies the constraint must be holonomic[1]. These can be formulated in traditional or generalised coordinates, provided that the constraint is not made redundant by reducing the degrees of freedom too far. The Euler-Lagrange equations for a system with N constraints are then modified to include the constraint equation and Lagrange multipliers:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{q}_j}\right) = \frac{\partial L}{\partial q_j} + \sum_{i=1}^{N} \lambda_i \frac{\partial G_i}{\partial q_j} \tag{2.1.8}$$

from which expressions for λ_i can be found along with equations of motion. $\lambda_i \frac{\partial G_i}{\partial q_j}$ terms represent generalised forces, and as such λ_i equations can be used to calculate forces, torques or other generalised analogues required to apply a given constraint at any point in time.

[1] A holonomic constraint is only defined by positional coordinates and time, without inequalities

Plane Pendulum This method is easily demonstrated through the example of a plane pendulum: normally, the Lagrangian for a pendulum of mass m and length l would be represented by

$$L = \frac{1}{2}ml^2\dot{\theta}^2 - mgl\cos\theta,$$

where θ is the angle between the vertical and the rod. In order to find the tension in the rod of the pendulum, the length of the pendulum is no longer constrained at this step, and so is replaced with a generalised coordinate r, subject to the constraint $r - l = 0$. The modified Lagrangian is then

$$L = \frac{1}{2}m(r^2\dot{\theta}^2 + \dot{r}^2) - mgr\cos\theta$$

and the subsequent Euler-Lagrange equations, formulated in the manner of equation (2.1.8) are

$$\begin{aligned} mr^2\ddot{\theta} &= mgr\sin\theta \\ m\ddot{r} &= mr\dot{\theta}^2 - mg\cos\theta + \lambda. \end{aligned}$$

Solving these equations is very straightforward with the inclusion of the constraint and its time derivatives, which show $r = l$ and $\ddot{r} = 0$. These lead to

$$\begin{aligned} \ddot{\theta} &= \frac{g}{l}\sin\theta \\ \lambda &= mg\cos\theta - mr\dot{\theta}^2. \end{aligned}$$

λ is equivalent to the tension in the rod, which is easily seen to be correct: the sum of the tension and the radial component of the gravity acting on the bob equals a centripetal force.

Application Within the context of mathematically modelling swing systems, this method opens up multiple useful avenues for investigating and interpreting simulations. As previously mentioned, the primary goal is to calculate the forces required to produce certain kinds of motion: the models produced involved a plane pendulum (either mathematical, compound, double or variations thereof) with an additional element that performed some set action, such as another pendulum connected to the bob that swings by the motion of an actuator at the hinge or a mass that moves up and down the rod, in a pumping motion modelling a human squatting on a swing. In the case of an actuator, the inclusion of the Lagrange multiplier allows direct calculation of the torque needed to maintain a motion, which can then be used to model the force required for a human standing on a swing to lean backward and forward.

While this is useful knowledge in itself, these forces can also then be used to evaluate effort functionals such as equation (2.1.14). When combined with a suitable reward function, this allows the implementation of machine learning methods and quantification of the performance of chosen motions. In comparison to machine learning, inspection of the multiplier allows a more direct and mathematical approach to selection of preferred action. For example, a set of driving functions could be chosen that ensures that the multiplier only ever imparts energy into the system rather than alternating between removing and adding work; this should produce a swinging motion that only ever increases in amplitude as the motion continues.

A supplementary usage of Lagrange multipliers involves the calculation of the work done by the corresponding generalised force. At a point in the system, the instantaneous infinitesimal work done by a constraining force F_i is given by

$$\delta W_i = F_i\delta q = \lambda_i\frac{\partial G_i}{\partial q}\delta q \tag{2.1.9}$$

Again, this may be used within an effort functional, but can also be used to analyse the error within a simulation. When numerically solving ordinary differential equations, discrepancies in conserved quantities of the system can be used to observe error propagation of the solver: in the case of equations of motion for constrained bodies moving in conservative potentials, the sum of potential and kinetic energies is often a good choice. However, with driving forces such as those provided by an actuator, the energy taken in and out of the system by the motive element must be considered to calculate the total energy of the system, and thus the work must be integrated to a given point. This will introduce further apparent error dependent on the method of integration; this is described in more detail in Section 2.1.3.3.

The following was contributed by: Chantal Birkinshaw

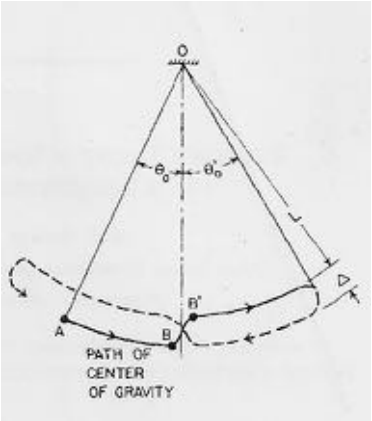


Figure 2.1.1: Illustration showing the path of the centre of mass over a swing period for a squatting motion.

2.1.2 Review of Literature

There has been much research conducted in the last half-century into mathematically modelling how a human drives, or 'pumps', a swing. Analysis of the resulting motion has been done in a variety of different ways, including deriving the equations of motion of the swinger from its Lagrangian, a somewhat more Newtonian and less mathematical analysis by examining the forces involved, and by examining input and conservation of angular momentum or energy during a swing cycle. The literature covers several possible methods used to pump a swing; the three most common are: squatting whilst standing on the swing, leaning whilst standing on the swing, and leaning and extending the legs whilst standing. Different papers have sometimes used different systems to model the same swing mechanism. Methods of swinging have varying levels of effectiveness; the increase in swing amplitude per cycle of the swing, and the motion of the swinger can often be characterised in different ways depending on the height of the swinger. Ultimately, swings can be said to be driven by an undulating centre of mass, or by producing a torque about the pivot (and in some models, both).

The 'standing-squatting' method is primarily modelled in literature with a mathematical pendulum, where the length of the rod is periodically varying by a small amount; a model of raising and lowering the centre of mass. See Figure I (i). This model is a parametric amplifier; when pumped at the swing's natural frequency, the amplitude of the swing increases, driven by a periodic change in a parameter (the distance from the pivot to the centre of mass). The three main papers, modelling this system, Tea & Falk(i), Curry(h), and Burns(c), all use different mathematical analyses, but return very similar conclusions from them. For maximum effectiveness, Tea & Falk, Curry, and Burns agree that the swinger should stand upright when the swing is at its lowest point, thus injecting the maximum amount of energy into the system as work is done against both gravitational and centrifugal forces at this point. Similarly, the swinger should squat at the peak of the cycle to remove least energy from the system. All of these models assume that the swing is being pumped at its natural frequency (the operative stands and squats twice per period), this of course being the point of resonance where the maximum possible amplitude can be achieved. Case(d), proves via deriving the equations of motion from the Lagrangian that the growth of this system is exponential with time; the rate of growth of amplitude is proportional to the current amplitude. Curry, using conservation of energy, also proves this result. In other words, mathematical proof that it is impossible to start this system from true rest, as a rate of growth proportional to 0 would also be 0. In reality, as Curry notes, a system is never truly at rest as it has thermal motion of order kT, but even so starting from this point is very difficult. Gore(g), points out that this because a point-mass system is used; this method is an oversimplification and swings can actually be started from rest using this method in reality. This, as Case points out, is visible on a playground; children are most likely to use the most effective method, and not one uses this squatting motion, which shows it has a visible ineffectiveness.

The other method of driving a swing whilst standing is to lean oneself back and forth. This is modelled as a double pendulum; McMullan(b) analyses the system as a mathematical one, Figure IIa(b), while Gore(g)(a) and Case(d) model the second pendulum as being physical (Figure IIb(b)). Indeed, as noted in Case & Swanson's paper(e), it is possible to model a standing swinger using their dumbbell model with the second mass set to 0

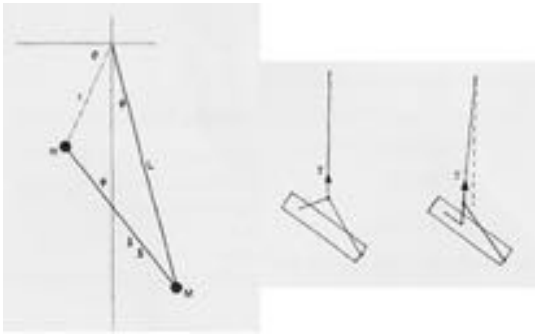


Figure 2.1.2: Schematic diagrams of models to represent swinging whilst standing. a. McMullan's model, using mathematical pendula [**?**]. b. Gore's model, using physical pendulum with flexible rope [**?**].

(see Figure III), but this model is described in the paper as being incomplete as it does not model the vertical motion of the swinger. Gore analyses this system by qualitatively looking at the forces involved. Gore postulates that a swinger at rest who pulls with their arms upsets the equilibrium and causes the tension in the rope to have a horizontal component which accelerates the centre of mass forwards, thus starting swing motion. There is an inherent assumption in this that the ropes are not rigid. This method does show that this model will start a swing from rest and sustain motion, but there are no mathematical conclusions involved, and Gore does not look at the effectiveness of the system. Case, via the method of Lagrangian analysis, argues that the swing is not a parametric system during standard operation; the equation of motion contains both driving and parametric terms, and for the small-to-medium angles at which a swing generally operates, the driving terms dominate and the amplitude of the swing grows linearly. However, at large enough angles it does indeed behave like the parametric systems described by other authors and exhibits exponential growth. As well as the mathematical argument, Case's other evidence for this case is the phase difference between swinger and swing motion for a parametric oscillator is different to that seen in practise, which is closer to the predicted phase relation for the driven oscillator. These conclusions are proved by the experiment conducted by Post et al.(j); at small angles the system is largely a driven oscillator, with parametric pumping playing a minor role, and as the angle increased so too did the relative contribution of parametric pumping. McMullan's analysis of the system is a similar mathematical form to Case, although he does not study methods of energy injection in the way that Case did. He does show that the motion of the swinger takes the form of exponential growth – yet another form of analysis showing this to be the case – and that there is always net angular momentum about the point of support; i.e. there is always a torque sustaining the motion. He comes to the same conclusion as Gore in that is far easier to start a swing with a flexible rope; however these are much harder to numerically model so do not come into this report.

Models of seated swingers vary depending on whether or not the legs are included. Gore and McMullan use models which could be for either a seated or a standing swinger, (effectively a physical double pendulum with actuator, a model which depends on whether the second pendulum is viewed as the motion of the torso of a seated person with massless legs, or as the motion of a massive person standing and leaning. This model has therefore already been analysed above. An alternative method is seen in Case & Swanson's(e) paper, which is to model the swinger as a single pendulum at the bottom of which is fixed a dumbbell, shown in Figure III(e). The dumbbell is set to move at a defined, periodically varying angle to the pendulum rod. Upon leaning back, the swinger acquires angular momentum, but as there are no external torques, total angular momentum must be conserved and so the centre of mass acquires an equal and opposite angular momentum. This causes the CoM to move forward slightly from its equilibrium position. Thus, the motion is driven by a combination of repetitively applied torque (driven oscillation), and parametric amplification(h). This is proved somewhat more mathematically by Case and Swanson who, after some series truncation and approximation, prove that the equation of motion for the dumbbell system contains a driving term and three parametric terms. For angles below 1.8rad, the driving term dominates and growth is linear; above this, the parametric terms dominate and growth is exponential. At this critical angle, the system transitions from one regime into another. The paper also concludes that there are two mathematically possible phases – leaning either back or forward as the swing

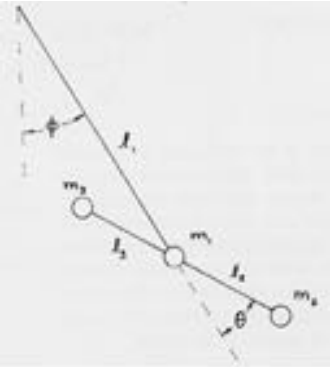


Figure 2.1.3: Schematic diagram of model used to represent pumping a swing from the seated position [**?**].

moves forward through its lowest point, and vice-versa for moving backward – but only the backward-leaning phase is ever used, or even experimentally successful, which implies there is more complexity to how a human swings. Their conclusion is only valid for angles less than 0.7rad, less than the critical angle, but it has been proved experimentally by Post et al.. This experiment also verified the use of the dumbbell model, as the motion of the torso and the legs of a swinging person was proven to be closely coupled, though perhaps not perfectly in phase. Analysis of a person swinging is also conducted in this report.

The literature definitely proves that a swing driven by squatting has its amplitude increase exponentially with time by doing work against gravity and centrifugal forces. This method is therefore most effective once the swing has got going, but it cannot easily be started from rest. Leaning and sitting swingers, while originally assumed to work by parametric motion, were discovered to operate as a driven oscillator for small angles, with corresponding linear growth in amplitude, and as amplitude increases the system becomes more like a parametric amplifier and exhibits more exponential growth. Ideally, a system would be made for the robot such that it would start off seated so its amplitude increases by a set number of radians per cycle, then once it reaches a critical angle, it would change its motion to the squatting motion so that its amplitude continues to increase exponentially with time until the resistive forces increase such that it becomes a parametric oscillator; a parametric amplifier operating at maximum amplitude. The seated swing does eventually exhibit more exponential growth, but it only begins to do so after 1.8rad(e); up until this point it only has minor parametric character as opposed to the fully parametric character of the squatting swing.

Developments of models made in this report have been made to build on some of the previously conducted research. Oftentimes, small angle approximations have been made or series truncated so the mathematical analysis becomes easier; while these approximations do allow conclusions to be drawn, the accuracy of the models produced can be limited, especially once the swing reaches larger amplitudes. For the models detailed in this report, numerical models have been used so the motion can be examined without having to make approximations. Direct algebraic analysis is not necessary for these models as the driving systems for swings are already known; it is the character of the motion, in particular the chaotic and resonant features, that are interesting. Case and Swanson did put in a numerical model in their paper, but as they have made several approximations their model is only truly valid for angles below 0.7rad. The models included later in this report have not made such assumptions, and the results are therefore valid for all angular amplitudes. Also, double pendulums are an inherently chaotic system, so exhibit many points of resonance, and ideally this behaviour should be looked at. As well as looking at on-resonance behaviour, the models included in this report also look at motion around and off-resonance. The models all make the major assumption that the swing is being driven at its natural frequency. This is reasonable for a human swinger; however the robot does not immediately know the natural frequency of the swing and so has to proceed more algorithmically. Therefore, it is also useful to look into the motion at non-resonant and near-resonant frequencies. This algorithmic approach has been looked at over the course of the project, to varying degrees of success, and is a useful direction for future projects to take. Another limitation of the models in literature is that none of the models incorporate friction or air resistance in any way. Models which included these would find that their amplifier system would eventually reach a point where it reaches a maximum amplitude, as the resistive forces would become sufficiently large such that they would

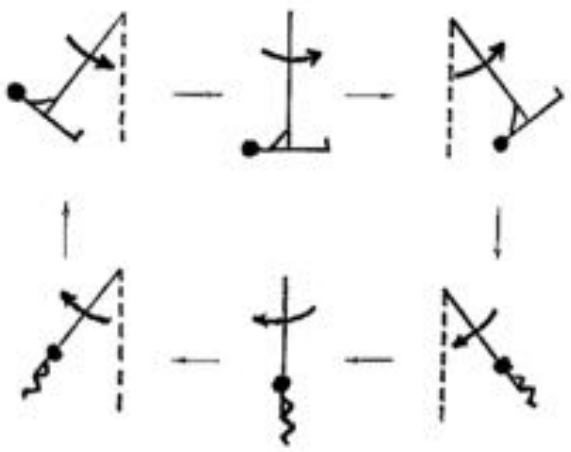


Figure 2.1.4: Illustration of motions performed by a person sitting on a swing [31]

dissipate all the energy the swinger would inject. The robot's swing is more complex than this, however. As well as improving on simpler models for squatting, leaning, and seated motion, another, more complex, model has been constructed to try analyse the motion for the robot's swing, which includes a pivot partway down. The models included later also examine 'effort' put into a system; this is a different way, other than the growth achieved per cycle, of measuring the effectiveness of a system. This analysis will be done for different swing types as well as different functions used to model how the swing is pumped.

——————— The following was contributed by: Owan Haughey ———————

2.1.2.1 Swinging whilst Sitting

The most common way a human swings sitting down by alternating between two positions.
An extended position, where the legs are straight and the body is held as close to horizontal as possible, and a closed position, where the body is pulled in to poles and head pushed forwards with legs tucked up under the seat.

From a sitting position, quickly alternate between extended and closed positions to start swinging. When swinging motion is achieved switches happen at two positions in a swing. The front where the swing is as far forwards as possible and the back where the swing is as far back as possible. At the back of the swing the body changes from the closed to the extended position and then holds this position until the front of the swing is reached. At the front change the body from extended to closed position and again hold until the swing reaches the back. These changes in body position can be seen in Figure: 2.1.4.

2.1.2.2 Swinging whilst Standing

The position a person holds onto the swing varies by swing type. For solid swing poles the hands are held just above waist height on pole. For a rope swing the hands are often placed higher up to get a higher second pivot which is not possible with solid bars.

During the swing a person will switch between two positions. Standing - the back and legs are straight, with elbows bent holding on to pole. Squatting - the arms are extended and knees bent making the body as low as possible in line with the swing, in this position the arms are often fully extended.

To get the swing moving a person alternates between crouching and standing. When swinging motion is happening switch between 3 positions in swing. Front, bottom and back. During the changes the body should move in a fluid motion to avoid jerks. As a person moves through the bottom of a swing they should be squatting. As they approach the top and the bottom they should stand up. At both top and bottom they should hold the standing position for a short period of time then quickly move to the squatting position before they reach the bottom of the swing. A greater increase in energy can be gained by standing up quickly as the body passes the bottom of the swing but this becomes less efficient at higher amplitudes.

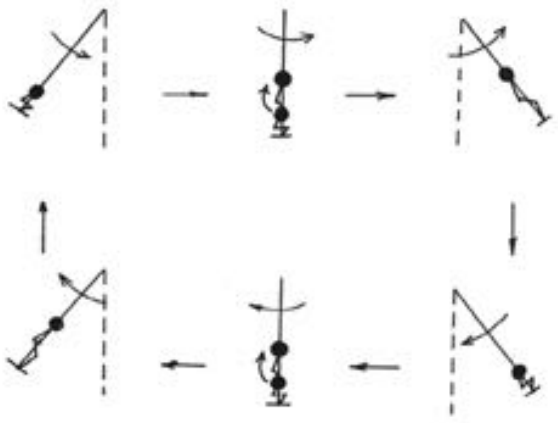


Figure 2.1.5: Person standing on a swing [31]

This set of motions requires four movements of the body however a person can also swing with two changes of the body by leaning forwards and backwards with minor squatting or none at all. This only requires two movement points so may be more efficient. These changes in body position can be seen in Figure 2.1.5.

2.1.3 Numerical Models

The following was contributed by: Michael Jevon

2.1.3.1 Numerical Simulations

To examine the behaviour of various swing systems, simplified mathematical models were designed and their appropriate equations of motion were found using Lagrangian mechanics and multipliers. However, in even the simplest case of a plane pendulum (with no restrictions upon validity), exact analytic solutions for motion cannot be easily found: therefore numerical methods must be utilised to solve these equations and investigate their evolution over time.

ODE Solving Equations of motion are second order ordinary differential equations (ODEs). Numerically solving ODEs is a large and important branch of numerical analysis and as such there are many extant methods that are potential candidates that are often very similar in use. In the scope of this project, the first step in the process is to reduce second order differential equations in terms of $q_i = q_i(t)$ into pairs of first order equations:

$$\ddot{q}_i = f(\dot{\mathbf{q}}, \mathbf{q}, t)$$

where, would become

$$\begin{aligned} \dot{q}_i = {}_1\dot{q}_i &= {}_2\mathbf{q} \\ \ddot{q}_i = {}_2\dot{q}_i &= f({}_2\mathbf{q}, {}_1\mathbf{q}, t). \end{aligned}$$

While this seems to be a trivial formality, it is important for the operation of the solvers used: they are extensible to any number of dimensions, but the equations must be first order. With this process applied to all of the generalised coordinates that need to be solved, the system can be represented by a vector function

$$\dot{\mathbf{q}} = \mathbf{f}(\mathbf{q}(t), t)$$

which can then be fed into various solvers along with initial values of $\mathbf{q}$ at time t. The starting point for investigation into the choice of solver was to implement the GNU Scientific Library (GSL), a large C library with many numerical analysis methods.

GNU Scientific Library GSL has many different solvers available for first-order ODEs of any number of dimensions: it offers an excellent advantage with regards to comparing and testing these solvers in its common environment for all solvers. After initialising appropriate workspaces, allocating memory for the solvers and setting up controllers[2] an ODE problem could be solved, and then the solver could be swapped with minimal modification to the code.

Solvers available in GSL fall into multiple categories: the most straightforward are the explicit Runge-Kutta family of methods, of varying orders. The Euler method can be considered a first-order Runge-Kutta method; and as such is useful for demonstrating the concept of this family of algorithms. It is conceptually simple: the solution $x(t)$ of a first order differential equation can be found by integrating the ODE $\dot{x} = f(x, t)$. By the fundamental theorem of calculus, $\dot{x}$ is the gradient of the solution function; starting from an initial value of x_0 and t_0, following the tangent to the curve (as given by $\dot{x}$) for a small step will provide an estimate for $x(t_1)$. This process is then repeated from the new value of x. At each step the discrepancy between the estimated value and the true value will accumulate, creating what is known as a *global truncation error*; as opposed to the *local* error produced from a single step. As the step size decreases, the local truncation error becomes smaller and thus the method becomes more accurate globally; as step size tends to zero, the result becomes exact. Higher-order methods evaluate $\dot{x}$ at midpoints and weight these to produce a more accurate estimate that converges more quickly to the exact solution as step size decreases.

Also included are implicit Runge-Kutta methods, which are of course similar to the explicit methods but have a larger bound of numerical stability at the cost of computation time. They require the Jacobian matrix for the system to be input and evaluated alongside the differential equations. Runge-Kutta methods discard information found at previous steps and only use the new estimated value in the calculation of the next. Linear multistep methods, the final major family of solvers, retain information about previous calculations to increase future computational efficiency. These solvers do not require the Jacobian but cannot start from a single set of initial values like Runge-Kutta methods. To overcome this limitation, another algorithm is used to generate sufficient steps for the chosen solver to operate - a feature built into GSL.

GSL also allows adaptive algorithms: all of the solvers can be used in a fixed form where order and step size are constant throughout the process, or alternatively the control system can change these as the simulation progresses. Runge-Kutta methods can feature variable step sizes, where the solver evaluates the local truncation errors within a region and adjusts the step size used to attain some chosen accuracy. If the function is flat and smooth over a region then the integrator will use a large step size in order to reduce computation time, and if there are rapid changes then small step sizes will be used to preserve accuracy. Linear multistep methods can also have their orders altered in a similar fashion- some portions of the simulation do not require as many terms in a calculation due to the system's predictability in that region, and thus the order of the solver is reduced temporarily.

Choice of Solver: To begin selecting an algorithm to use, the most obvious factors to consider are the stability of the system, the desired accuracy, and computational efficiency.

Some ODE systems exhibit *stiffness*. A stiff system is defined as one that, to achieve acceptable accuracies, requires a step size much smaller than would be expected when compared to the exact solution's smoothness at a point [23]. Some systems are only stiff within certain regions, which sometimes can be ignored, but sometimes a chosen solver is not stable when applied to the system. As such, if the system is seen to be stiff, certain solvers become ruled out: [23] explicitly, explicit Runge-Kutta methods are unstable when faced with a stiff equation; some linear multistep methods as well as the implicit Runge-Kutta algorithms deal with these well. However, none of the systems used[3] were stiff and therefore the simpler, more efficient family of explicit Runge-Kutta methods made a good candidate. This also removes the need for the Jacobian matrix which significantly reduces the amount of mathematics required to set up the system.

The classic fourth-order Runge-Kutta method, abbreviated as RK4, is a very widely used member of the family, providing an excellent balance between computational efficiency and quickly converging accuracy- with a step size of h, the global truncation error is expected to be of the order of $O(h^4)$. Using lower order methods would reduce computation time at the cost of accuracy, and higher orders or methods with more detailed

[2]Even though this process is fairly straightforward and does not require many changes when changing solvers or systems, it is fairly arcane in its documentation and procedure: this in part motivated the creation of a simpler, clearer RK4 implementation.

[3]Hamilton's equations for a non-driven double pendulum exhibited stiff behaviour, however this simulation was not used in the body of the project and thus will not be discussed in detail.

weightings would do the inverse. RK4 is widely seen as an optimum middle ground and a good general purpose integrator and as such was used for solving multiple simulations; also used was MATLAB's ODE45 routine which uses an adaptive step size algorithm that is a combination of fourth- and fifth- order Runge-Kutta methods.

Custom RK4: After the RK4 algorithm was selected and used within various simulations, it was decided that a shift from GSL would be a sensible recourse. This had several motivations: the first was portability, because the simulations were only using a small portion of GSL it was inefficient to integrate the library with other systems such as FLTK, whicg was used to create animated representations of motion. The robot was also capable of running C code, and as such a lighter weight implementation could potentially be used on board, although this was not implemented. Another reason for the switch was for transparency: while GSL was using an adaptive RK4, there was limited knowledge about how the interaction between the time dependent actuator functions and the evolutionary algorithms took place. This would not be an issue if the actuator functions were unchanged throughout the simulation, however with the introduction of responsive functions that were defined in a piecewise manner would make calling them ahead of time give potentially unreliable results. As such, a new RK4 method was implemented which gave a better understanding of this interaction. This used a fixed step size with the classic RK4 method described by

$$\begin{aligned} \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \\ t_{n+1} &= t_n + h \end{aligned}$$

where

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(\mathbf{y}_n, t_n) \\ \mathbf{k}_2 &= \mathbf{f}(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_1, t_n + \frac{h}{2}) \\ \mathbf{k}_3 &= \mathbf{f}(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_2, t_n + \frac{h}{2}) \\ \mathbf{k}_4 &= \mathbf{f}(\mathbf{y}_n + h\mathbf{k}_3, t_n + h). \end{aligned}$$

The operation of the new implementation was also made clearer in comparison to GSL, making for a more user friendly solver. For instance, the memory allocation steps are automatically handled with a single line initialisation, and the current state of the variables at any step are more readily accessible via the solver class which allowed for easier integration with FLTK graphics.

2.1.3.2 Piecewise Functions

A common problem encountered in driving these pendulum simulations arose with the use of periodic driving functions. For example, a double pendulum with an actuator at the hinge that constrained the angle between the rods as a sinusoidal function of time would easily begin swinging from rest if the frequency of oscillation was similar to the resonant frequency of the system. As expected, the amplitude of oscillations increases rapidly. However, large amplitudes alter the natural frequency and the driving force begins to move out of phase and proceed to damp the system dramatically.

A possible counter to this problem is to make the motive element respond to the state of the system rather than time; for example only performing an action when the apex of the swing has been reached. This requires piecewise definition of functions, where the driving constraint can sit at a steady state between actions for any length of time. Care must be taken in selecting these functions: the first and second derivative of the acting function can appear in equations of motion as well as the Lagrange multiplier, and as such the function must be sufficiently smooth to avoid singularities in its derivatives. A potential solution lies in the use of *splines*, which are smooth piecewise polynomials of at least quadratic order. Hyperbolic functions also offer a good alternative, particularly tanh and the reciprocal functions- they all feature horizontal asymptotes which tend to zero very quickly outside the central region. As an example, consider a function which produces a motion from one static state to another, with a roughly constant velocity transition. The spline could be defined as a horizontal segment transitioning to a linear segment via a quadratic, and then again back to horizontal; a similar effect can be achieved by using tanh.

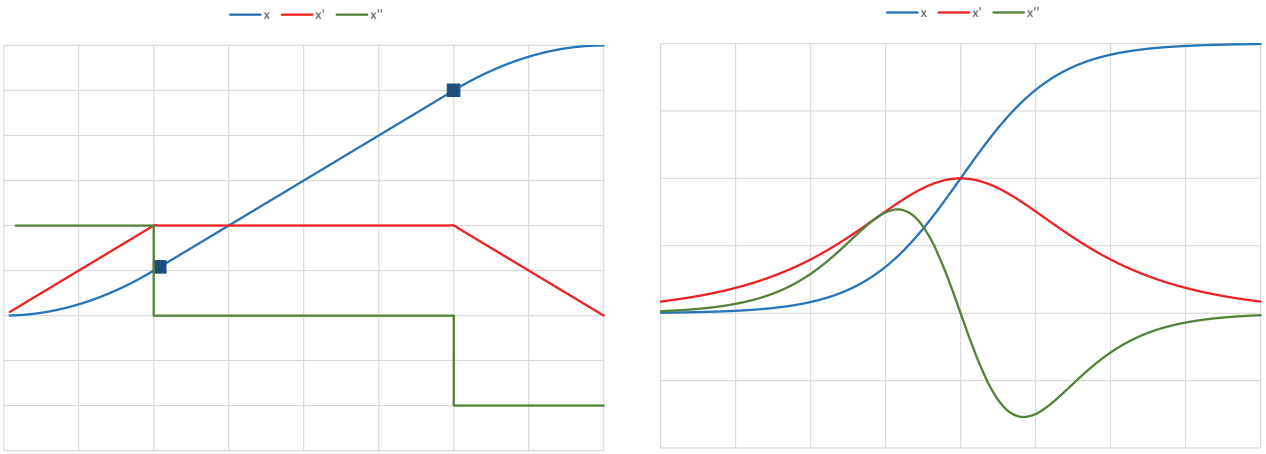


Figure 2.1.6: A comparison between a spline and a hyperbolic function to achieve a smooth state transition, along with their first and second derivatives

As shown in Figure 2.1.6, both functions transition smoothly from the lower state to the upper one. Although the hyperbolic function can be implemented in a single piece and has completely smooth derivatives, it has some drawbacks compared to the spline: these mostly lie in its lack of customisation. With a set maximum velocity, the transition always takes a set amount of time, whereas the length of the accelerating parts of the spline can be altered to change the time taken even with the same velocity of the middle segment. This can allow for more accurate representations of experimental actuators by tailoring the length of the segments to suit. The spline could also be more easily extended to have a continuous moving period. These benefits are only from convenience, however; the function is already split and thus easy to extend- the hyperbolic function can achieve these with careful separation and addition of new segments. However, using tanh is well suited for keeping models simple and avoiding accumulation of parameters- varying the amplitude and speed of a function like $f(t) = A\tanh(\omega t)$ that acts at swing extremes is a simple numerical task that can give much information about preferred frequencies and effort/reward calculations.

——— The following was contributed by: Tom Crossland ———

2.1.3.3 Error Analysis Method

Some of the following systems will be modelled numerically to aid in analysis, and hence the problem of accuracy arises. In models such as these, some conserved quantity is generally used to calculate the systems discrepancy, and hence produce some error analysis. However, as the systems in question involve the application of a generalised force to some coordinate (position of the moveable mass, or rotation of the dumbbell), we must consider the work done on the system as well as the kinetic and potential energies of its component parts.

If the generalised force, F, on a coordinate, q, is calculated, the work done by some fictional "motor" over a given time step, δW_n may be calculated using,

$$\delta W_n = (q_n - q_{n-1})\,\frac{F_n + F_{n-1}}{2}.$$

for step n. This approximation of the trapezium rule does not represent the most accurate method of calculation, but its simplicity and ease of application make it desirable in the models we shall create. The work done at each step is summed, and the percentage error at that step, n, is calculated by finding the difference between the initial energy of the system, and its current energy minus the overall work which has been done on the system, such that the percentage error,

$$\eta_n = \frac{E_n - W_n - E_0}{E_0} \times 100,$$

where E_n is the energy at step n, W_n is the total work done on the system by step n, and E_0 is the initial energy of the system. Using this, we may examine the errors inherent in the simulation, allowing the validity of the simulation to be determined.

A similar approach may be used to account for the presence of dissipative forces.

——————— The following was contributed by: Owan Haughey ———————

2.1.3.4 Simplified Robot Models

Mathematical models can be used to predict the behaviour of a system at simple and complex levels. These models should be tested to check the validity of the model. It is easier to test a simple prediction, so simple models were made which could be used to test behaviour and see if the results agree with predictions.

Sitting An initial aim was to make a simplified model that could be used to simulate the robot sitting and trying to achieve motion on the swing. For this, a single motion was devised to approximate the robot's legs. An initial simulation for this was a motor clamped to the seat of the swing with a piece of wire hooked through it. Masses could then be added to the wire to simulate a leg. The workshop drilled a hole through the shaft of the motor to attach the wire to the motor. The mass of the robot's legs below the knee was calculated to be 296g per leg a giving a combined centre of mass of 588g at 8.68cm below the knee. From this the legs were considered as a point mass at the centre of mass.Torque at a pivot is proportional to the mass and the length to the centre of mass for a physical pendulum. This gives a value for torque of $50.1sin(\theta)$N for the legs. To get a similar value of the model, a mass of 150g was hung at 34cm below the knee. This gives a torque value of $50sin(\theta)$N.

By alternating the current direction across the motor, motion of the swing was produced. The motor used here had an output torque of 2mNm, compared to a possible torque of 68mNm for each of the robot's knees. The fact that this motor was able to move the swing, 4.2kg, suggests that the robot should be able to move the swing using just its legs, despite its extra 4.6kg mass, as it possesses two motors with 68mNm torque each.

Standing When it was decided that the robot would be swinging standing up, a simplified model to represent this was required. The body would again be approximated as a point mass at a distance from the seat. For this model a pole the length of one of the swing poles was made by the workshop. This was then attached to an encoder at the top. A small stepper motor was screwed onto the bottom of the pole, model number *57BYGM201*, to be used to rotate a small arm in a controlled motion. When looking at the arm to be moved originally a simple model of the robot was considered. However the motor was unable to output enough torque to move the robot's mass, so a lower mass was used; the chosen mass was 200g attached to an aluminium arm 26cm long. These were chosen as the motor could easily move this without the arm slipping around the motor's shaft, which happened with larger masses.

The motion of the motor was controlled using a Phidget board, *bipolar stepper controller 1067*. The code for the Phidget board was written in C, while the code for the encoder was done in Python. These skeletons were provided by group members and modified for this model. The code for the encoder and the Phidget controller were changed to output Unix time - that is the elapsed time since 00:00:00 on Thursday, 1 January 1970. This was changed to make sure that the encoder and stepper outputted the same time to make referencing the outputs easier.

Initially the stepper was set to move between two points. The rate it does this is set initially and it will accelerate to that speed at a desired value. After looking at feedback from the motor angle, this acceleration is seen to be non-linear. The motor was set to move between $\frac{\pi}{4}$ either side of the vertical using this motion, with a velocity of 1500 sixteenths of a step per second and an acceleration of 8000 sixteenths of a step per second per second. By looking at the angles and visually, movement can be seen with increasing amplitude when the motor oscillates at close to the resonant frequency of the swing.

As the resonant frequency changes when the amplitude increases, there are flat points in the angle seen from the encoder where the pole briefly stops moving. This stop time increases as the number of cycles of motion increases. This effect may also be the cause of the anomalous encoder angle during the 14th motor cycle.

The resulting angle changing with time from the encoder and stepper can be seen in Figure2.1.9 below.

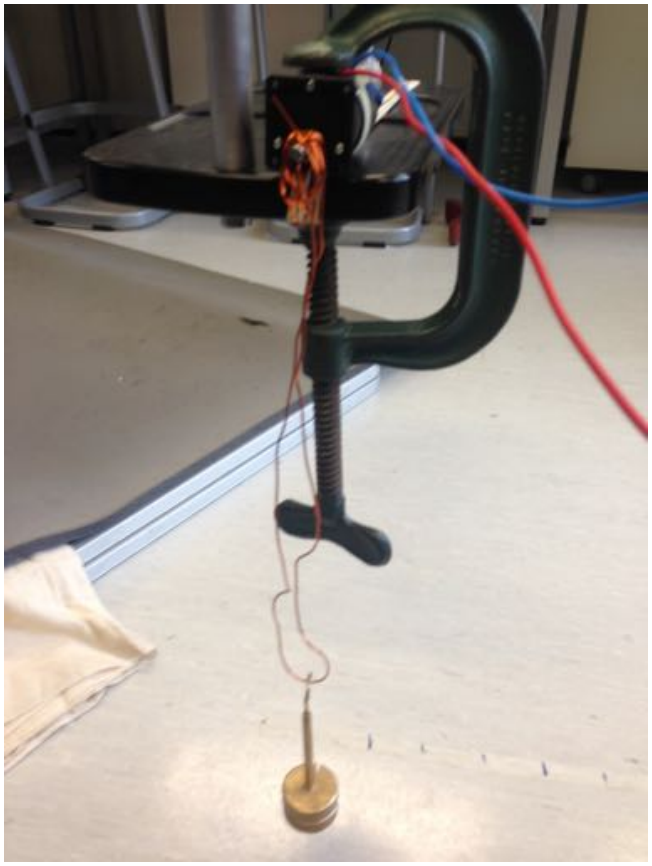
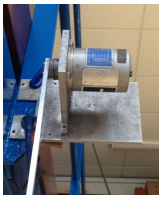


Figure 2.1.7: Swinging leg model

(a) Phidget controller

(b) Stepper motor



(c) Encoder

Figure 2.1.8: Standing model system: Phidget, Stepper and Encoder

(a) Encoder angle

(b) Stepper motor angle

Figure 2.1.9: Graphs of the encoder angle relation with the stepper motor angle

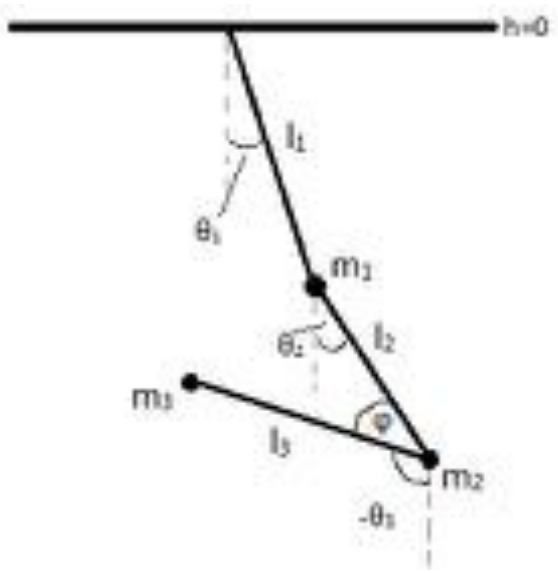


Figure 2.1.10: Triple Pendulum Model Setup

Future Models There were several attempts to make the motor move sinusoidally, and at the end of the project sinusoidal motion was achieved using angle feedback from the Phidget controller. A future target could be to have the motor moving sinusoidally with respect to time and using it to take measurements. Further goals could include using time-dependent functions to see what effect motion in different parts of the oscillation has on the amplitude, and using this to try to find the best way of swinging.

——— The following was contributed by: Chantal Birkinshaw ———

2.1.3.5 Triple Pendulum Model

One way of forming a model of the robot (or indeed a human) driving the swing by leaning is as a triple mathematical pendulum with an actuator at the joint between the second and third rods, as shown in Figure 2.1.10. θ_1, θ_2, and θ_3 are defined as the angles of each rod compared to its equilibrium position (straight down). The actuator is a model of the person driving the swing by leaning themselves back and forth. The angle ϕ, which the actuator controls, is defined slightly differently to the θ angles, for ease of telling how far from the positive vertical the third rod is (i.e. $\phi = 0$ is defined as the person standing upright).

$$\phi = \theta_3 - \theta_2 + \pi \tag{2.1.10}$$

In the analysis which follows, two functions for ϕ have been used;

$$\phi = \phi_0 sin(wt) \tag{2.1.11}$$

$$\phi = \phi_0(sgn(sin(wt)) + 1) \tag{2.1.12}$$

where sgn(x) is the sign function; this latter form of ϕ is a square wave. The former of the two functions is much easier to mathematically model and analyse, as it and its derivatives are continuous. The latter however is much closer to the natural motion that a human (and more specifically the robot) will perform; it involves only sudden motion, and at resonance these motions will be at the extremes of the swing. However, it is not a perfect model of the robot, as the robot moves more algorithmically; it is set to perform its sudden motion just before the swing is at its peak. ϕ_0 is set to be small ($\pi/20$) so the details of oscillatory motion could be analysed. For larger ϕ_0, at resonance the system can achieve very large angles, eventually ceasing its oscillation about the origin and resulting in circular motion with increasingly large acceleration, shown in Figure 2.1.11. For a model incorporating friction, the damping terms would eventually be sufficiently large as to cancel the growth caused by resonance, and the system would reach stable oscillations. As torque is produced around the pivot, the system is able to start from rest; all simulations begin from this stable equilibrium.

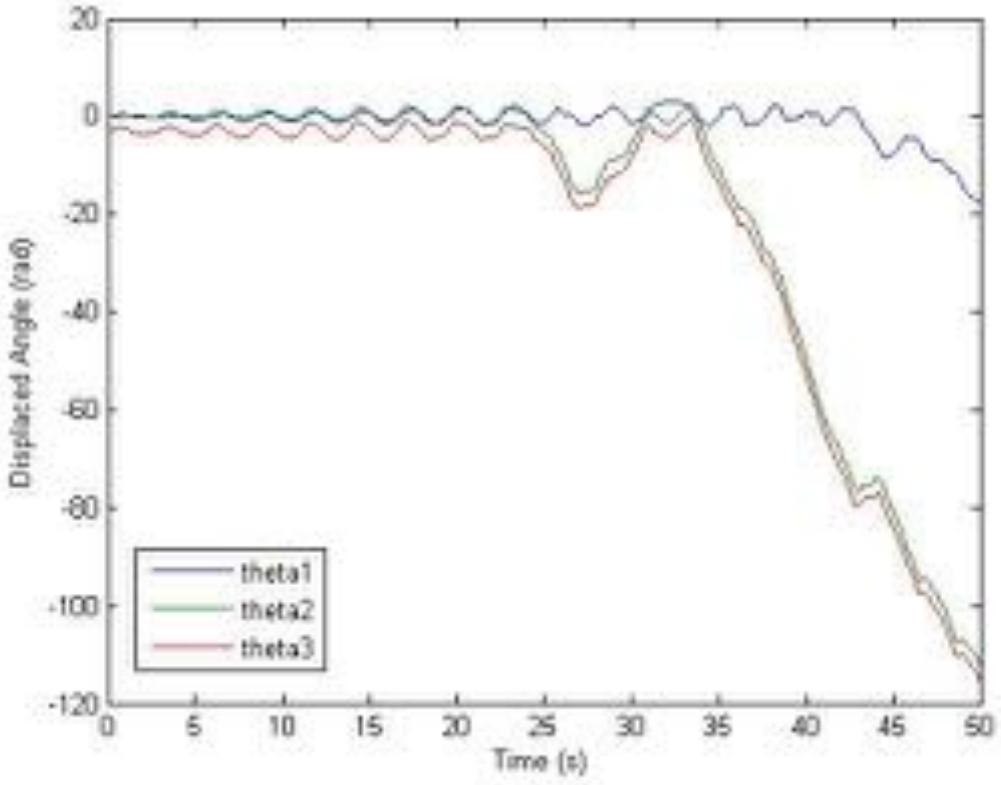


Figure 2.1.11: Resonance resulting in circular motion

The model was developed by working out the Lagrangian of the system with the three thetas as generalised coordinates, then using Euler-Lagrange equations to find the equations of motion for each mass, using the actuator angle ϕ as the constraint on θ_3. This was accomplished using the program Sage to handle the mathematics, and the resulting equations of motion were put into a Runge-Kutta numerical solver in MATLAB to find the motion of the three components of the system. The torque produced by the actuator was calculated using the method of Lagrange multipliers; by incorporating a 'generalised force' into the Euler-Lagrange equations. Having the torque allows the work done by the actuator to be calculated, which allows consideration of energy flow in and out of the system, and looking at energy conservation as a whole. A 0.53% error was calculated for the model using this method.

Figure 2.1.12a shows motion of the system far from resonance, $\omega = 1$, for the sine-wave actuator function. θ_3 is lower down on the graph as the third mass oscillates about the negative vertical, π radians. Its shape, while roughly sinusoidal, is characteristic of a chaotic system; it has a degree of unpredictability. This off-resonance shape happens because the actuator does not drive the third mass in sync with the natural frequency of the

system. As a result, it sometimes puts energy into the system as it does work, but it also sometimes results in removal of energy from the system, so over time, the maximum achieved angle does not change significantly over time from small values.

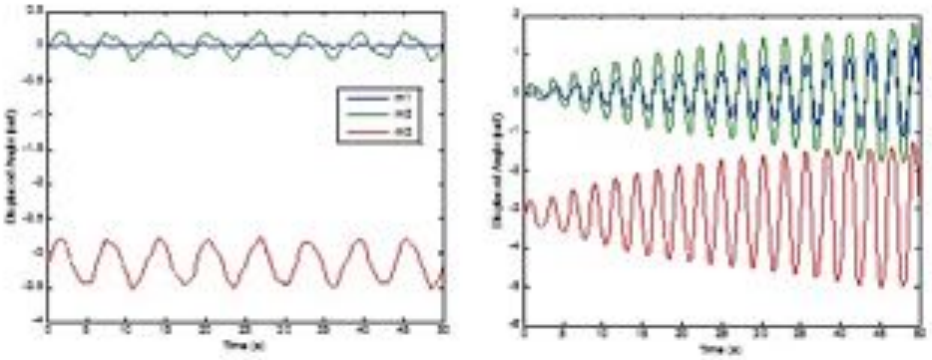


Figure 2.1.12: a: Off-resonance motion. b: On-resonance motion

Pumping at the natural frequency of the system results in resonance, and the achievable angle of the swing becomes very large, as shown in Figure 2.1.12b. The actuator pumps in sync with the movements of the system; it is continuously putting energy into the system as it does work driving ϕ, and hence can achieve increasingly large angles. The shape of the curve is also much smoother and more predictable than the off-resonance one.

The shape of this curve can also give insights into the type of motion the system is undergoing, and what is driving the motion. Figure 2.1.13 is the blue θ_1 curve taken from Figure 2.1.12b. At small-to-medium levels of θ_1, up to about 0.5rad, the system acts like a driven harmonic oscillator; where the system is driven by an external force (the actuator). This type of motion has a linear increase in angle per cycle, visible in the figure, which is related to the moments of inertia in the system. As θ_1 increases, the parametric terms in the equation of motion increase compared to the driving terms, and the system behaves increasingly like a parametric oscillator; where the motion is instead driven by the periodic variation of one parameter, in this case the distance from the pivot to the centre of mass. A parametric regime exhibits exponential growth per cycle. In effect, this model is an extended and, off resonance, somewhat more chaotic, version of the one developed in 1996 by Case[d], which, using a similar method to get the equations of motion from the Lagrangian, proved that a standing swinger acts like a driven oscillator at small angles, and like a parametric oscillator at large angles, but for the realm of operation of a standard swing, the driven oscillator mechanism is the far closer approximation.

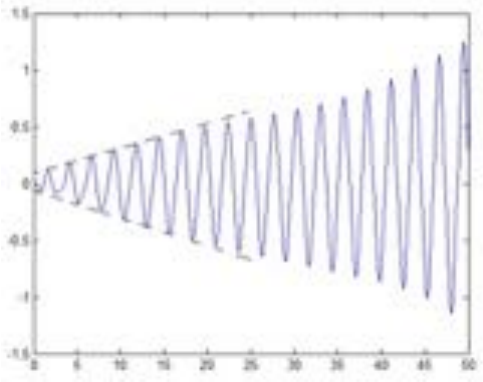


Figure 2.1.13: Driven and Parametric Regimes

Figure 2.1.14 shows graphs of maximum achieved angle as the driving frequency of the actuator changes. As this is a chaotic system, there are many points of resonance that the system can achieve, the greatest one in this model being at a driving frequency of around 2.35 radians per second. Figure 2.1.14b shows the behaviour around this point. The main resonant peak visible in Figure 2.1.14b is actually formed with many smaller

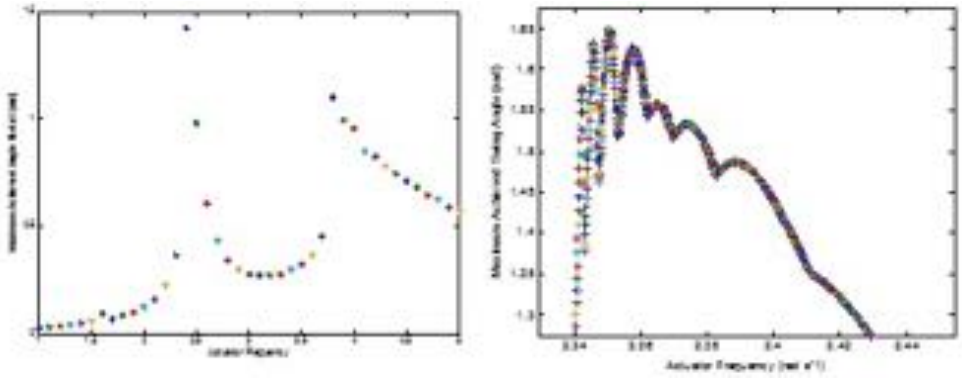


Figure 2.1.14: Achieved Theta-1 Angles a: Large Range. b: Close to Resonance

sub-resonances around it. This is a result of having parts of the system interfering with each other. When two very similar frequencies interact, in this case if the actuator frequency is very close but not exactly equal to the natural frequency of the system, they interfere with each other, causing what is acoustically known as a beat. Beats are formed of large, slow envelope wave enclosing a much faster wave. This is visible in Figure 2.1.15, below, of θ-1(t) if the system is run for a sufficiently long time. If θ_1 is sufficiently high, the system destabilises. This causes the resonant frequency to change slightly and so the actuator is now removing a little bit of energy and the maximum angle goes down, then returns to increasing as the system returns to stability. It is very hard for the system to achieve perfect resonance (especially for this small ϕ-0) because of this. The presence of these sub-peaks could cause a bit of trouble for some reactive algorithms the robot could be following, which could cause it to get trapped at the top of what the algorithm thinks is the true resonant peak, but it is actually only a sub-resonance.

Chaotic systems are very sensitive to initial conditions; change a parameter even slightly and there can be drastically different motion after enough time has elapsed. This can not only be the initial positions and velocities of the masses, but the driving frequency of the actuator as well. This is visible in Figure 2.1.16, which shows the motion for three very similar actuator frequencies. While the curves are identical for about 0.2rad, very similar for up to 20 seconds, or 0.4rad, the small change in the driving frequency soon produces some drastically different motion.

The model was also analysed using a version of an effort functional (see 2.1.4) to try to quantify the level of effort the person is putting into that motion:

$$Effort = Asin(\phi/\phi_0) - B\dot{\phi} + C\ddot{\phi} + D\tau \tag{2.1.13}$$

The point of developing an effort functional is to find the point at which the system puts in the minimum effort for the maximum achieved angle – this is one way of measuring efficiency. Ideally, the best point for the system would be the one where effort-per-maximum achieved angle would be minimised. Figure 2.1.17 examines the effort of the system around the point of greatest resonance. 'Effort' in this case is the total effort expended over the run of the system, which in this case was 50 seconds. The units are somewhat arbitrary as the coefficients of the terms in the effort functional are not absolutely defined; the simulation is more to observe the shape and approximate relative proportions of the terms. As is shown in Figure 2.1.17a, most of the effort functional is dominated by the torque term, and the other terms contribute only minor variations. This is because most of the system's 'effort' comes from the actuator doing work (exerting torque). Figure 2.1.17b shows total (*) and reduced total (+) efforts. 'Reduced total effort' in this case is the total effort divided by the maximum angle of the swing, θ-1, achieved in the time. There is a peak seen in the effort at the point of resonance; the torque is at a maximum here as this is the point where the actuator is doing most work, and more importantly, the most efficient work. It is clearly seen in the reduced effort that the minimum-effort-per-maximum-angle is located at the resonance of the system – the system has to put in a little more effort at this point, but it achieves far greater angles.

The square wave model exhibits somewhat similar behaviour, especially off-resonance, although none of the motions performed by the swing are perfectly smooth due to the discontinuous motion. However, the point of resonance is not as clearly defined for the square wave as for the sine; instead of a defined peak of close-to-perfect

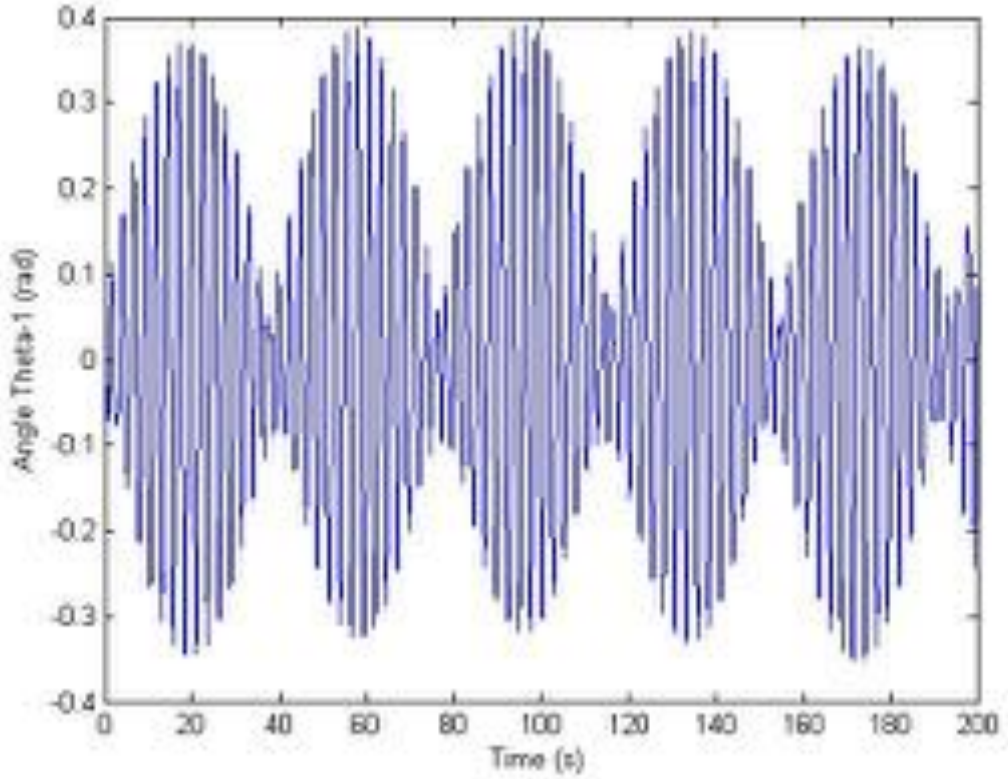


Figure 2.1.15: Demonstration of Envelope Waves

resonance, there is a much larger area (of actuator frequency) where growth of the swing's angle (θ_1) is fairly large and exhibits the driven-turning-to-parametric characteristics explained earlier, but the two lower masses do not move in tandem; they are much more likely to flip themselves over than perform ideal swing motion. This is probably a fault of the model itself, as the square wave is formed of discontinuities. Off resonance, the two appear to have very similar levels of effort, but close to and on resonance, the sine system appears to be putting in less effort than the square wave system but getting greater amplitude from it. This is contrary to intuition – the square wave would be the function expected to expend less effort as the swinger spends in uncomfortable transition points, and less time moving and thus creating torque, and is probably again a fault of the model.

The square wave function would be the ideal case to do most of the work with in a model designed for the robot, as this is closer to the motion the robot performs; however this actuator function is difficult to model accurately. Both the first and second derivatives are discontinuous and formed of Dirac delta functions, terms which are not possible to include in the code, which cannot cope with infinite numbers. These terms are thus approximated to be 0 for all t. While this is reasonable for a model to obtain the general idea of the behaviour of the system, as the two derivatives of ϕ are only non-zero at infinitesimal points, there is a certain amount of accuracy lost in dispensing with these derivative terms, as many terms in the equations of motion are now lost due to a coefficient of 0. The form of the actuator function is probably better off being replaced by a piecewise function formed of something like tanh which is continuous.

——————— The following was contributed by: Tom Crossland ———————

2.1.4 Effort Functionals

To determine which motions are the most efficient for the robot whilst operating the swing, rather than optimal to maximise some parameter (e.g. the amplitude of the swing oscillations), we must consider some form of 'effort' which the robot requires to undergo the motion. Additionally, if we hope for this effort to be lowest for

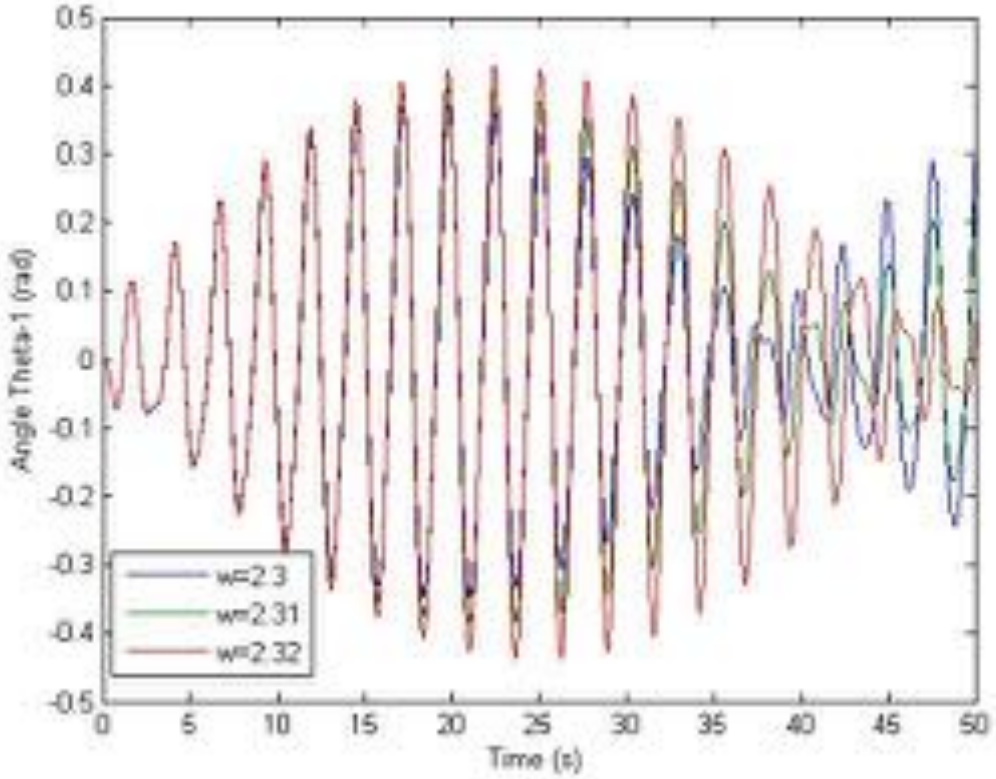


Figure 2.1.16: Demonstration of Chaos

human-like motions, we cannot only consider, say, the current usage of the robot's motors. We must instead consider the practicalities of human motion.

Little usable research could be found relating to this problem in the literature – that which existed went far beyond the scope of this project, considering complex biomechanics. Human muscles contract lengthways, with varying energy requirements relating to type of task and muscle type, hence requiring complex analysis if energy/effort considerations are to be undertaken. As such, an approximation was decided upon for use in this project, and is given by,

$$E_{ff} = A \sin\left(\frac{q}{q_{max}}\right) - B \mid \dot{q} \mid + C \mid \ddot{q} \mid^2 + D \mid F_q \mid^2. \tag{2.1.14}$$

The above considers a single degree of freedom, q, for the motion of the robot (e.g. angle made with swing supports), and consists of four terms, relating to: position, q, velocity, $\dot{q}$, acceleration, $\ddot{q}$, and force (torque in the case that q is an angular coordinate), F_q.

The position term is reasoned as follows: it is easier to maintain a position when held at an extreme. For instance, consider standing on a swing: holding oneself close to the ropes, or leaning back fully with arms extended, requires less effort than holding the body midway between these extremes. So, the term produces smaller values nearer these extremes, with a peak in the middle of the possible range of motion. q_{max} represents the maximum possible value that the coordinate q may take.

The velocity term considers that it is easier to be in motion than to be stationary. It places less strain on muscles to be moving slowly than it does to be static. As such, this term has a negative sign, producing a smaller effort for larger velocities. Ideally, this term would be of the form,

$$-B\dot{q}\,(a - \dot{q})\,,$$

where a is some constant, but any parameters chosen would have very little foundation. As such, the simpler, if less accurate version given above shall be used.

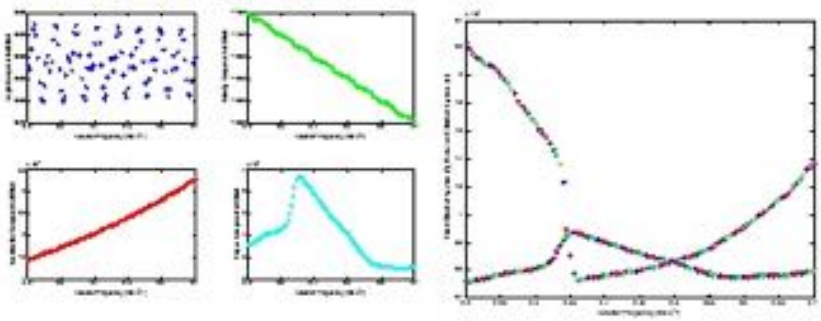


Figure 2.1.17: a: Components of effort. b: Comparing Effort and Reduced Effort

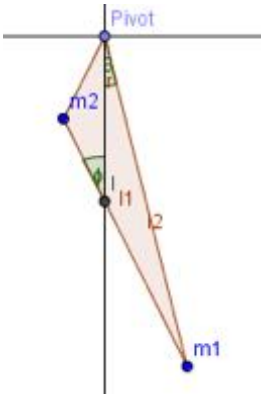


Figure 2.1.18: Initial diagram of the system before the first transition.

The acceleration and force terms use similar and simple reasoning: it is hard to accelerate a limb/body quickly. As such, effort increases quadratically with both acceleration and force, giving a large increase in effort for a small increase in acceleration or force.

The precise magnitude of the four coefficients, A, B, C, and D, will depend upon the specific system in question. However, it should make little difference to overall trends in effort requirements for the system.

If we wish to consider the efficiency of a given type of motion, we take some quantity we wish to maximise and divide by the average effort. The value produced is a measure of the efficiency of the motion. This may be used in reward-based learning algorithms, or simulations and predictions, to allow motions to be quantitatively examined.

——— The following was contributed by: Stuart Bradley ———

2.1.5 Starting Swing From Rest

The aim with this section of work is to show that it is possible to start a swing with rigid supports from rest, in the case of a simplistic mathmatical model. To model this we started with a compoud single pendulum, that transfoms into a double pendulum and then back to a compound pendulum once a certain distance between m2 and the pivot point has been achieved. This is representative of a humanoid on a swing, initially having their arms bent and being close to the supports and then relaxing their arms so they fall under gravity, then grasping the poles again when a maximum externsion of the arms. The framework of this idea is from Gore, B. F., 1970, "Starting a Swing From Rest," American Journal of Physics, 39, 347; however we arrive at different results.

It is important in the initial set up that attention be drawn to the detail that the swinger leans slightly backwards so that the supports of the swing at rest are not vertical. This is a crucial detail because otherwise with no external force the system would just sit at a position of unstable equilibrium.

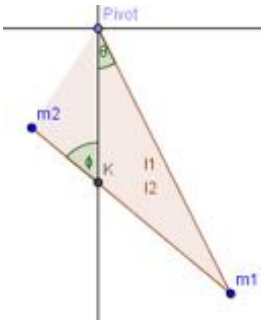


Figure 2.1.19: At a time after t=0, double pendulum

2.1.5.1 The Double Pendulum

At time t = 0, our pendulum becomes a double pendulum and the second mass begins to fall under the influence of gravity, we can describe its behavior with its Lagrangian.

$$\mathcal{L} = \dot{\theta}\dot{\phi}l_1 l_2 m_2 \cos(-\phi + \theta) + \frac{1}{2}(\dot{\theta}^2 l_1^2(m_1 + m_2) + \dot{\phi}^2 l_2^2 m_2) + g l_1 m_1 \cos(\theta) - (l_2 \cos(\phi) - l_1 \cos(\theta))g m_2 \quad (2.1.15)$$

By using the Euler-Lagrange equation and small angle approximations, we can form two simultaneous second order ordinary diffenential equations.

$$\ddot{\theta} l_1^2(m_1 + m_1) - \ddot{\phi} l_1 l_2 m_2 + g l_1 m_1 \theta + g l_1 m_2 \theta = 0 \quad (2.1.16)$$

$$-\ddot{\theta} l_1 l_2 m_2 + \ddot{\phi} l_2^2 m_2 - g l_2 m_2 \phi = 0 \quad (2.1.17)$$

If solutions to these equations are assumed to be of the form

$$\theta = \theta_0 e^{\alpha t} \quad (2.1.18)$$

$$\phi = \phi_0 e^{\alpha t} \quad (2.1.19)$$

, differentiating these trial solutions twice each with respect to time, gives expressions for $\ddot{\theta}$ and $\ddot{\phi}$. Subsituting these in addition to equations (4) and (5) into equations (2) and (3) gives a quartic equation in α, which allows us to solve for α^2. This gives two values for α^2, a positive and a negative one; the negative value of α^2 means an imaginary value for α. The behaviour of the system that we are looking for is a decay as opposed to an oscillation. This means that we want α to be real instead of imaginary, so we discard the negative α^2 value. The value for α^2 becomes important later on as we use it for comparison to a term in the angular momentum of the system.

$$\frac{(\alpha^2 l_1 + g)l_2}{\alpha^2 l_1^2} = \frac{\alpha^2 l_2^2 m_2}{(\alpha^2 l_1 l_2 + g l_1)(m_1 + m_2)} \quad (2.1.20)$$

$$\alpha^2 = -\frac{(g l_1 + g l_2)m_1 + (g l_1 + g l_2)m_2 - \sqrt{(l_1^2 - 2\,l_1 l_2 + l_2^2)m_1^2 + 2\,(l_1^2 + l_2^2)m_1 m_2 + (l_1^2 + 2\,l_1 l_2 + l_2^2)m_2^2}\,g}{2\,l_1 l_2 m_1} \quad (2.1.21)$$

By taking the appriopriate inital conditions, where there is no initial velocity, α is real and at $t = 0$, $\theta = \theta_0$ and $\phi = \phi_0$ we obtain that

$$\theta = \frac{1}{2}\theta_0(e^{\alpha t} + e^{-\alpha t}) \quad (2.1.22)$$

$$\phi = \frac{1}{2}((m_1 + m_2)/m_2 l_2)(l_1 + g/\alpha^2)\phi_0(e^{\alpha t} + e^{-\alpha t}) \quad (2.1.23)$$

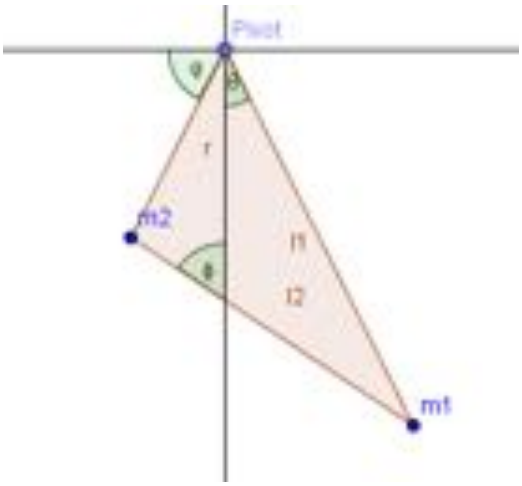


Figure 2.1.20: System at end of double pendulum phase, $r = r_{max}$

2.1.6 Angular Momentum of Double Pendulum

The total angular momentum of the system around the pivot point is the sum of the angular momentums of the individual point masses around the pivot point.

$$\Delta = \Delta_1 + \Delta_2 \tag{2.1.24}$$

$$\Delta = \dot{\theta} l_1^2 m_1 + \dot{\psi} r^2 m_2 \tag{2.1.25}$$

As none of the previous workings have involved $\dot{\psi}$ or r it is far more useful to find an equivalent expression of $\dot{\theta}$, $\dot{\phi}$, l_1 and l_2.

$$\Delta = \dot{\theta} l_1^2 m_1 + (\dot{\theta} l_1^2 - (\dot{\theta} + \dot{\phi}) l_1 l_2 + \dot{\phi} l_2^2) m_2 \tag{2.1.26}$$

Substituting in expressions for $\dot{\theta}$ and $\dot{\phi}$ gives the total angular momentum as a function of time.

$$\Delta = -\frac{((l_1^2 - l_1 l_2)\alpha^2 m_1 + (g l_1 - g l_2 (m_1 + m_2)) t_0 (e^{\alpha t} - e^{-\alpha t})}{2\alpha} \tag{2.1.27}$$

The importance of this comes from reorganising the equation so that you get a value for what α would have to be if we make angular momentum equal to 0.

$$\alpha^2 = \frac{(g l_1 - g l_2)(m_1 + m_2)}{(l_1^2 - l_1 l_2) m_1} \tag{2.1.28}$$

This is not the same as the value of α^2 that was calulcated earlier by the differential equations, this means that the angular momentum can not be 0 after $t = 0$. This is an important result because having a non-zero angular momentum implies that the centre of mass of the system is displaced from the centre line and there will be a moment or torque that makes the system start to oscillate or swing.

2.1.6.1 After maximum extension

After the arms or length r has reached its maximum, depending on whether you are considering it physically or mathmatically, the double pendulum becomes a compound pendulum and its shape becomes fixed. At this point without any external forces introduced to the system the pendulum will oscillate indefinitely about the equilibrium point with a period of,

$$T = \frac{2\pi}{\omega} = 2\pi \sqrt{\frac{I_{support}}{Mgh}} \tag{2.1.29}$$

,

where h is the distance to the centre of mass from the pivot point, and M is the sum of the two masses.

$$I = \sum_i m_i r_i = m_2(l_1^2 + l_2^2 - l_1 l_2 cos(\phi - \theta)) + m_1 l_1^2 \tag{2.1.30}$$

$$h = l_1^2 m_1^2 + (l_1^2 + l_2^2)m_2^2 - 2l_1 l_2 m_2^2 cos(\phi - \theta) + 2l_1 l_2 m_1 m_2 sin(\phi - \theta) \tag{2.1.31}$$

$$T = 2\pi \sqrt{\frac{m_2(l_1^2 + l_2^2 - l_1 l_2 cos(\phi - \theta)) + m_1 l_1^2}{(m_1 + m_2)g(l_1^2 m_1^2 + (l_1^2 + l_2^2)m_2^2 - 2l_1 l_2 m_2^2 cos(\phi - \theta) + 2l_1 l_2 m_1 m_2 sin(\phi - \theta))}} \tag{2.1.32}$$

Having knowledge of the period of the swing after it has started moving is important because it allows you to know when to make thrusts or introduce forces to start increasing the amplitude of the oscillations. It looks unusual having the variables ϕ and θ in the expressions for period, which intuitively would be a constant. However $\phi - \theta$ is constant, even the individual terms are not, as it corresponds to the angle between the two rods of the pendulum, which is now fixed.

Because the angle is now fixed it gives us a constraint in the system; this means the original Lagrangian is no longer valid for the system, and so we have introduce an extra term which will act as the constraint. The extra term that is added in has a Lagrange multiplier signified as λ, along with the constraint equation,

$$\phi - \theta = a \tag{2.1.33}$$

$$\mathcal{L} = \dot{\theta}\dot{\phi} l_1 l_2 m_2 \cos(-\phi + \theta) + \frac{1}{2}(\dot{\theta}^2 l_1^2 (m_1 + m_2) + \dot{\phi}^2 l_2^2 m_2) + g l_1 m_1 \cos(\theta) - (l_2 \cos(\phi) - l_1 \cos(\theta)) g m_2 + \lambda(\phi - \theta - a) \tag{2.1.34}$$

By doing a Euler-Lagrange equation on this new Lagrangian for each of he time dependant variables, ϕ,θ and λ, we get three simultaneous equations, which can be used to calculate the two new equations of motion, and the value of the Lagrange multiplier λ.

$$(m_1 + m_2)l_1^2 \ddot{\theta} - m_2 l_1 l_2 cos(\phi - \theta)\ddot{\phi} - m_2 l_1 l_2 \dot{\phi}^2 sin(\theta - \phi) + (m_1 + m_2) g l_1 sin(\theta) + \lambda = 0 \tag{2.1.35}$$

$$m_2 l_2^2 \ddot{\phi} - m_2 l_1 l_2 cos(\phi - \theta)\ddot{\theta} - m_2 l_1 l_2 \dot{\theta}^2 sin(\theta - \phi) + m_2 g l_2 sin(\phi) - \lambda = 0 \tag{2.1.36}$$

$$\phi - \theta - a = 0 \tag{2.1.37}$$

2.1.6.2 Summary

It has been shown that it is possible to start a swing without requiring an initial velocity, which was the aim from the start. However it should be noted that there are several limitations. The limitiations don't stop the example making physical sense, but would require more complicated mathmatics to iron out each simplification that was made. Firstly we made many small angle approximaions. This isn't a huge issue, the first oscillation of a swing starting from rest is likely to be small. Also all resistive forces have been ignored, mainly friction at the joints and air resistance; these would have to be accounted for in a physical system. We modelled the system as massless rods with point masses at the end of them, instead of at the centre of mass, this probably has more of an effect than the previous limitations, as no real system has a similar distribution of mass.

2.2 Computational Theory

The following was contributed by: Jack Ford

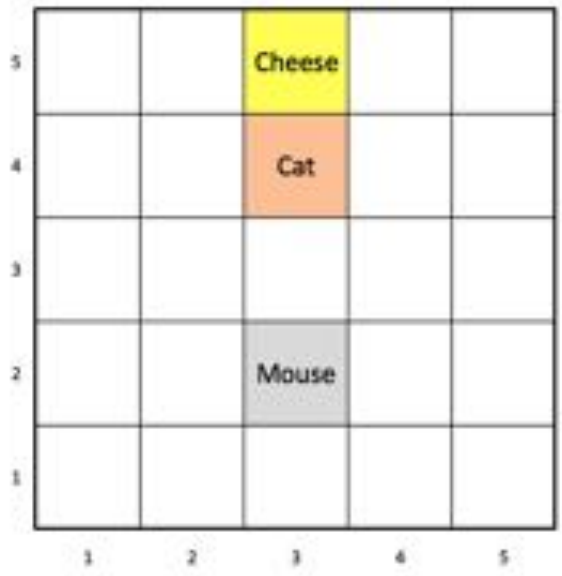


Figure 2.2.1: Cat and mouse example for machine learning

2.2.1 Machine Learning Approaches

2.2.1.1 What is Machine Learning?

Machine learning can take many forms, but in general machine learning is the ability of a program to use data to provide a solution to a problem without being explicitly programmed to do so. This means that a program, once it has gained experience, will be able to perform better at tasks and can be used to provide the optimum solution to various problems. This is particularly useful in scenarios where a numerical solution is required rather than an analytical solution. An example of machine learning can be found in Figure 2.2.1 where we have a mouse that is trying to obtain the cheese whilst avoiding the cat. If the mouse is on the same space as a cat it will receive a low reward score and if the mouse gets to the cheese it will receive a good reward score, if the cat is on the same space as the mouse and the cheese it will receive a score in between. The machine learning aspect of this is that as the system progresses, the mouse will learn what not to do and what to continue doing in order to maximise the score and eventually find a route to get the cheese and not be eaten.

Machine learning more generally has two main approaches; supervised and unsupervised learning.

Unsupervised learning is useful for finding patterns in data where the algorithm isn't given any labels and where there is no reward function for the algorithm. An example of this would be trying to find clusters in data. Supervised learning, like in our earlier example, takes input and output data and finds a mapping between them; sometimes these examples are already provided. After it has learnt from examples it can then test this on previously unseen data to try and predict the output. This utilises labelled data; that is, data which has already been assigned a type, or class.

Due to our experiment requiring the NAO to decide which motion will make it swing like a human being continuously throughout the swing we need to use a third type of machine learning called reinforcement learning. This is closer to supervised learning but because the input data is unlabelled it isn't exactly the same.

2.2.1.2 Reinforcement Learning

Reinforcement learning is based on behavioural psychology, similar to how animals can associate an action with a reward and then continue to perform that action. We can write algorithms that use a similar methodology to decide which actions to perform in order to gain a greater reward. Reinforcement learning, like supervised learning, learns from the data provided by performing an action. It then relearns based on a defined reward function for that motion.

In the example in Figure 2.2.2, where the system is trying to get a pendulum to stay upright after a torque is applied to the system (the action). If the pendulum moves closer to the centre its reward will be higher, if that

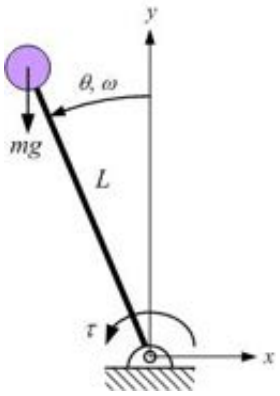


Figure 2.2.2: Inverted pendulum example [22]

torque makes it move away from the centre then the reward will be less. Eventually the system will balance in the centre and will have the maximum reward.

Most supervised machine learning starts by defining two things, the agent (in the above example the torque applied) and the environment (in the above example the rod) and these are both key in reinforcement learning. The agent is the thing that will be performing the actions that we want it to improve, and the environment is what it is applying these actions to.

In order to understand some theories of reinforcement learning we need to define 4 more features:

- Policy
- Reward function
- Value function
- Model

The policy determines which action states are mapped to specific states of the environment, determining the actions of the machine at a specific state and this is altered during the learning process.

The reward function determines the immediate reward of an action, so when an action is taken we can find out if it has a positive or negative effect on the system and as such can decide whether or not we need to change the policy. The reward function can be determined directly from the environment and is easy to calculate. It can be difficult to decide which action was the cause of the 'good' reward and is referred to as the 'blame attrition problem' in machine learning.

The value function determines the long term worth of an action, taking into account the subsequent motions and whether or not they provide a better end state. This takes into account that an action with a low reward may allow for other states to be accessed that overall have a greater net reward. Values are only estimates of the future and are much more difficult to calculate and these estimates will change with the number of simulations.

Finally a model of the environment can be created. This allows for planning the next motion so, rather than using trial and error to learn what is the best solution to the problem, it predicts what the next state and the next reward is. The model is an optional thing and while used in some reinforcement learning systems it is not used in all of them.

One of the issues with reinforcement learning and how the algorithm is formulated is how to balance the aspects of exploration and experience. Exploration is important to find which actions provide a greater reward, but too much exploration and the system will not be using the experience it gained from previous actions. If the system uses experience over exploration then it may not find the optimal solution; it may have found a solution that was optimal initially but has not taken into account the value of a different path of actions. Another issue with the experience is the amount of data required to be stored from previous iterations and how this is indexed.

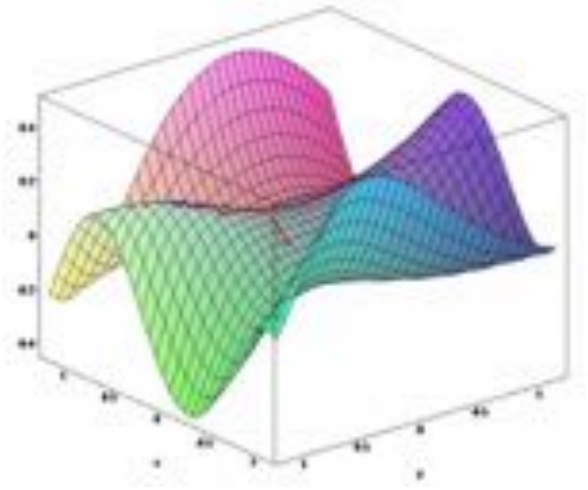


Figure 2.2.3: An example of a policy gradient [29]

2.2.1.3 The Algorithms Trialed

PGPE (Policy Gradient with Parameter Exploration) is one of the methods utilised from the pybrain package. In order to explain this we must define a policy gradient. Policy gradients are where the policy is defined by a combination of all the different parameters available and their projected reward; in the case of NAO the different parameters are angle and velocity of the legs and arms. Policy gradients can then be used by differentiating with respect to each parameter to estimate the value of the policy gradient for each parameter change [30]. Policy gradient methods allow for problems with continuous states and with a large number of dimensions. Because the reward is a part of the gradient there is no explicit value function. An example of how a policy gradient could work can be found in Figure 2.2.3.

PGPE methods try to overcome a large problem with the policy gradient method, which is that in a stochastic system the parameters are unknown, the parameter exploration uses previous episodes data to fill the parameter space making the new system deterministic.

The gradient of the policy is calculated using the finite differences method and is done by finding out the value of the policy gradient at a small perturbation of a parameter and finding the difference between this and the original policy gradient.

Chapter 3

Method

3.1 Investigations into the Swing

The following was contributed by: Chloé Smith

3.1.1 Why were modifications to the orginal swing design nessessary?

The swing given to the group initially (see Figure 3.1.1) was not fit for purpose. NAO's arms did not fit in between the poles that he would grip onto, and the swing was not designed to augment motion with limited effort. Therefore a re-design was necessary for both the standing and sitting swing set ups. The "Robot Testing" group were tasked with this.

When you make a fist with your hand, you can rotate it horizontally and vertically. The techinal terms for these motions are *pronation* and *supination*. This is thanks to the Distal radioulnar articulation (a joint) in between the radius and the ulnar. But, of course, NAO is not equipped with this complex anatomy. The "twisting" motion one makes when swinging with their hands on the chain (or, in our case, pole) is a critical part in increasing the amplitude of motion.

When someone is at maximum amplitude when swinging, they tend to throw their head back and bring their feet forward. The centre of mass of the user is somewhere low in their chest, below where the chains are suspended. When this motion is executed by the user, a positive torque is applied to their extremities (i.e. their head and feet). But everything in between and the swing itself experience a negative torque. To counteract this negative torque (which would "twist", or rotate the swing) there must be an opposite torque acting, which comes from the chains. The tension in the chains is normally below the centre of mass; so to counteract this torque the tension in the chain must increase. More specifically, the user pulls on the chain, and is lifted ever so slghtly. This is vital as it is this lift that raises the centre of mass, which results in an increase in potential energy. This potential energy is converted into kinetic energy at the bottom of the motion and they swing higher.

Figure 3.1.1: A CAD image of the swing intially give to the group.

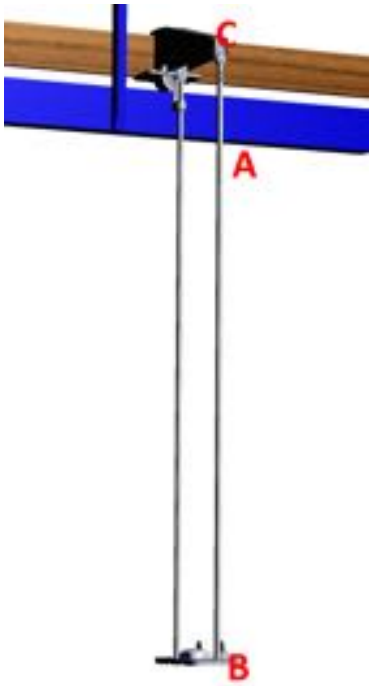


Figure 3.1.2: The initial swing design.

Figure 3.1.3: A comparison between the initial swing and the final standing design.

The basic physics here are compatible for both sitting whilst sitting, or standing.

NAO does not have rotating wrists, and therefore cannot pull on the chain in the way described. And even if he could, poles are being used, not chains, and therefore would not deform under this motion - making the whole action even more difficult.

This is where the pivots come in. The pivots in the swings' poles somewhat re-enact the complex motion described above, and increase his amplitude when swinging whilst reducing the net force NAO must exert.

——————— The following was contributed by: Joe Allen ———————

3.1.2 Swing Design

Due to the swing in Figure 3.1.2, not being suitable for the main project aim of achieving seated swinging it was necessary to formulate a new design. The main reasons for this are discussed in Section 3.1.1. This was of paramount concern as an initial project goal necessitated making the robot (NAO) swing in as close to human-like fashion as possible. This required NAO being able to sit on the swing and impart enough energy to reach a reasonable angular displacement. Consequently there were constraints which had to be placed on the design, and other external factors which drastically narrowed the scope of possible designs. For example, several avenues of design were rejected on the grounds that the swing would look abstract in comparison to your everyday playground swing, which initially was one of the key aims of the re-design plan.

One of the first concepts put forward involved the swing's poles being widened laterally in the region above and below NAO's shoulders, as this solved the problem of NAO's elbows being too wide for the poles. However, this would restrict the range of motion of the torso, resulting in less energy imparted to the swing.

At this point it became apparent that although the Physics Workshop produce equipment engineered to a high precision, they receive a lot of requests and subsequently the seated swing design was not made. Prior to this, the target had been to design two separate swing set ups: one for seated and one for standing. It was decided to partially combine the plans to decrease the workload on the Workshop. After various tests it was noted that to ensure the robot could lean as far backwards as possible, when seated, the hollows in the seat needed to be extended, otherwise NAO's motion was restricted by its anatomy.

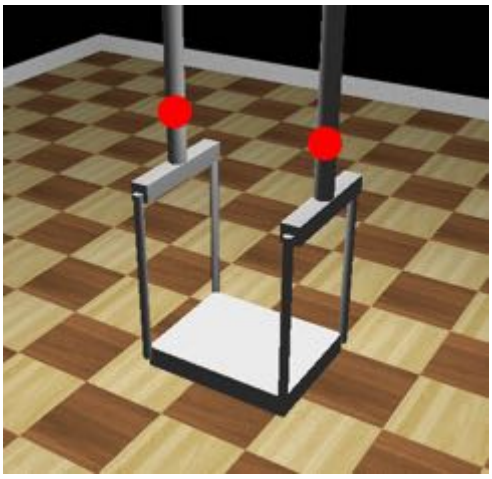


Figure 3.1.4: The final design of the seated swing, the pivot positions are indicated by the red circles, it has yet to be built.

The robot had some key flexibility issues, as described in Section 3.1.1, which became a problem when trying to create a human-like swing for it to use. For instance, NAO's wrists lack any degrees of freedom, other than rotational, and its elbows lock at approximately 90°. Both of these factors greatly restrict the design of the hand-holds: they must be flexible if the arms are not perpendicular to the line **AB** in Figure 3.1.2 and must be at least at a distance equal to the length of the forearm and hand to NAO's body.

It was decided to incorporate two pivots, one above and one below the robot's hand, to account for the lack of wrist flexibility, and to use this in a combinatorial design which essentially involved placing the seat at two different heights to allow standing and seated swinging, with the poles offset in front of the base in the seated conformation. This was later found not to be viable due to the shifted centre of mass of the system; however it caused the birth of the idea of using the same main pivot for both standing and seated swinging. Not only that but it had necessitated the calculation of the optimum distance of the handles from Nao's body for the seated design.

Armed with this newly discovered knowledge, the final designs were reached. The standing configuration was the same as the original swing, but with additional pivots above and below the handles, **1.** in Figure 3.1.3. The seated design, Figure 3.1.4 would use the same new top pivot as the standing design, with a rectangularly shaped set of rods attached which also attach to the seat. The rectangular shape extends in front and behind where the robot sits, which allows it to grasp the handles at the correct distance and also ensures the centre of mass of the system and robot is close to the line **AB** in Figure 3.1.2.

The updated standing swing was built and used successfully by the robot. It is a very good representation of a swing a human would use for the same task, although man-made swing supports are mostly made to be flexible; they are often metal chains or ropes, to allow users of differing heights to create pivots at their hands' points of contact with the supports. Due to the nature of the robot's grip and for ease of production, the designs produced were limited to inflexible rods and fixed pivots. Given that the swings were only to be used by the same sized robot, these restrictions did not matter so much.

The final seated swing design was initially rejected until later in the process when it became apparent that it was not possible to make a seated swing for the robot that looked like a conventional human swing. Overall, the tasks were completed satisfactorily and the designs fit for purpose. Time constraints did not allow the construction of the seated swing, but next year's group will no doubt be able to start the project by confirming the design and having it built.

——— The following was contributed by: Owan Haughey ———

3.1.3 Motion Capture Set-Up

One of the aims of this experiment was to make NAO swing like a human. To do this the way a human swings on a swing had to be found out. This could be done by looking on the Internet but it is often better to perform

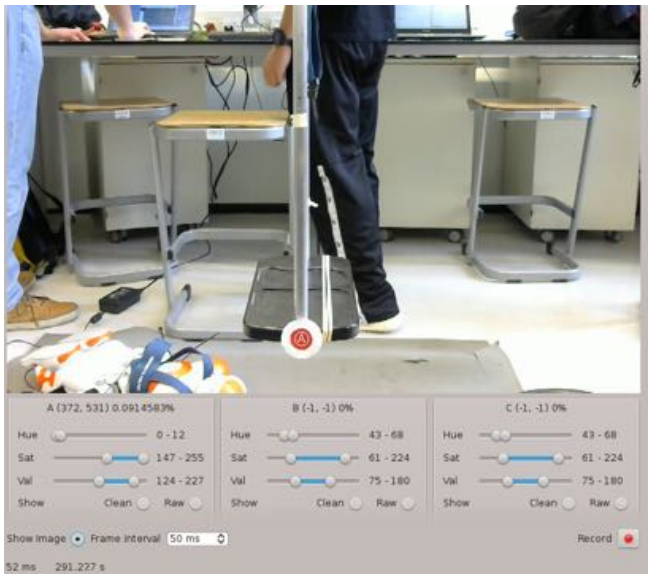


Figure 3.1.5: Printscreen of software

the research physically. To this end research of a person swinging was recorded. To take the research a swing was created in the lab and recordings taken of a person seated on it.

To use the motion capture software points in the body have to be picked out. To do this when a person was seated on the swing post-it notes of different colours where fixed to their body at the spots to be tracked. The motion capture software *Tracker* was used in this experiment. A limitation of the software was that it could only measure three points at a time. As a result of this multiple recordings were taken with differing points measured.

Another problem that the software suffers from is distinguishing colours. It struggled to focus on the desired colours and background colours often were mixed up with the tracking points. To counter this a white background was put into place behind the swing to block the extra exposure to colours not part of the swing system. To get the desired points recorded the hue, saturation and value were adjusted for each of the three points in the picture. These were altered until each tracker value picked up only one of the three points.

The positions data were taken for include: sitting down swinging both from rest and from initial displacement, natural motion of the swing and a person swinging from rest and initial displacement without using their arms. A print-screen of the Tracker software can be seen in Figure 3.1.5.

3.1.4 The Encoder

The following was contributed by: Will Etheridge

3.1.4.1 Theory

An encoder is a device that facilitates the conversion of one type of data into another. Typically a physical quantity, such as position, is transformed into a measurable electronic output to be interpreted by a computer. The outputted electronic signal is most commonly in binary or Gray code format depending on the required function. These number systems are crucial in understanding the theory of the encoder as they underpin the logic of their functionality.

Gray code, also known as Reflected Binary, is similar to binary in the sense that it is a base two number system. Gray code, however, has the unique property that two consecutive numbers in this system differ by no more than one bit. This property makes the use of Gray ideal for position encoders. In general, position encoders contain a piece of hardware that has a unique configuration assigned to each position it can record. These unique configurations are output as a digital electrical signal. The physical change of one configuration to the next is therefore measured and represented as a change in this output signal. When changing between two consecutive configurations using binary, errors can arise from bits changing at different times due to

manufacturing in-precision in the hardware of the encoder. Gray code, however, eliminates the possibility of this error arising due to the fact that from one configuration to the next only one bit will change.

In the case of this investigation, an 11-bit rotary encoder was used, which converts the rotational configuration of the encoder shaft into a digital Gray code signal and then to a digital binary output. This type of encoder has 11 fixed LED and light sensor pairs, each pair corresponding to one of the 11 bits of output, separated by a plate that is rotated with the shaft. The plate consists of a series of opaque and translucent windows. If an LED is blocked by being in front of an opaque window then its paired light sensor registers a low voltage or a 0 for the digital output of that bit. Conversely, if an LED is in front of a transparent window, its paired light sensor registers a high voltage or a 1 for the digital output of that bit. The plate is designed such that when moving from one configuration to the next, the state of only one light sensor will change and hence only one bit of the digital output will change. The signature for a given position of the shaft corresponds to the base-two Gray code number received by reading the digital output of all the 11 light sensors. Circuitry within the encoder then converts the Gray code signature into an 11-bit binary number to be read at the external output. The output consists of 25 pins; 11 pins correspond to the 11-bit binary number for the angle, 3 pins are to ground and 1 pin is for a 5V(DC) input. The pin map for the encoder is shown in Figure 3.1.2. The internal conversion of Gray to binary is a physical implementation of the conversion algorithm. The conversion algorithm for Gray to binary is as follows:

1. The most significant bit of the Gray number becomes the most significant bit of the binary number.
2. The second most significant bit of the binary number becomes the exclusive OR of the second most significant bit of the Gray number and the most significant bit of the binary number.
3. The third most significant bit of the binary number becomes the exclusive OR of the third most significant bit of the Gray number and the second most significant bit of the binary number.
4. And so on for as many bits in the gray number.

This is implemented in the encoder as a series of exclusive OR gates as shown in Figure 3.1.6 for a 4-bit Gray to binary conversion. The truth table for exclusive OR is shown in table 3.1.1.

Table 3.1.1: XOR truth table

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.1.2: Pin map for 11-bit encoder

Pin	Function	Pin	Function
1	1st DIGIT(MOST SIGNIFICANT)	14	SPARE
2	2nd DIGIT	15	OUTPUT GROUND
3	3rd DIGIT	16	CIRCUIT GROUND
4	4th DIGIT	17	SPARE
5	5th DIGIT	18	+ 5 VDC
6	6th DIGIT	19	SPARE
7	7th DIGIT	20	SPARE
8	8th DIGIT	21	LAMP MONITOR
9	9th DIGIT	22	SPARE
10	10th DIGIT	23	CASE GROUND
11	11th DIGIT(LEAST SIGNIFICANT)	24	GATE INPUT
12	SPARE	25	REVERSE COUNT
13	SPARE	—	—

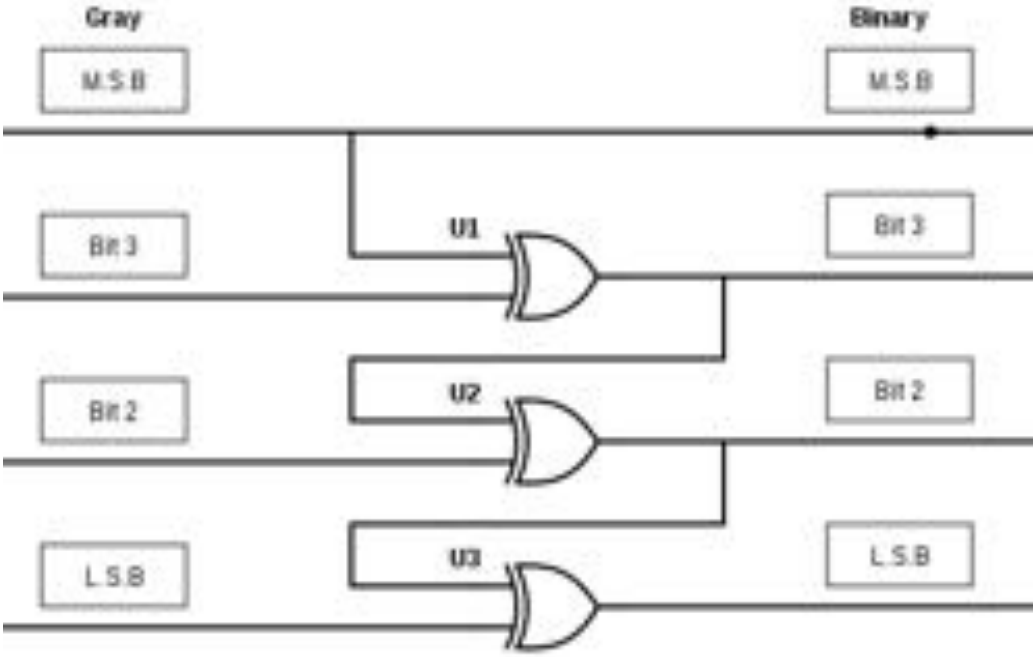


Figure 3.1.6: Logic circuit for conversion of 4-bit Gray number into 4-bit binary number

The binary output can then be sent to a computer via an interface to be expressed as a 16-bit integer. An interface is a link between two separate parts of a system that need to communicate. They facilitate the exchange of data.

3.1.4.2 Initial Considerations

Two alternative methods were considered to provide diagnostic information on the motion of the swing; a rotary encoder and an accelerometer. An accelerometer measures acceleration and could have provided the time period and points of maximum amplitudes of the swing via monitoring the linear acceleration of the seat. An accelerometer could have also been used to measure the pitch of the seat and this could have been used to get a value for the angle of the swing. These calculations would have introduced unwanted errors and the whole method would have become redundant if the structure of the swing was augmented to include a second pivot for example. A rotary encoder, if installed on the apex of the swing, could have directly provided the angle that the swing made to the vertical. From this the time period and the points of maximum amplitudes could have been obtained. In addition to this, the encoder could provide real time values for the angle and through a computer program these values could be logged and monitored. The ability to have these real time values was crucial as the mathematical models being developed for the swing intrinsically depended on the generalised coordinate of the system. For the swing, or a simple pendulum, this generalised coordinate is the angle the system makes to the vertical. The rotary encoder provided also boasted an accuracy of 0.175 degrees. The encoder was, therefore, decided to be the most accurate and logical diagnostic equipment for providing information on the motion of the swing.

——————— The following was contributed by: Will Edmondson ———————

3.1.4.3 Feasibility

In order to better understand whether the robot could move the swing from a static initial position a simple physical system using a stepper motor was designed to simulate the sitting robot's leg in motion.

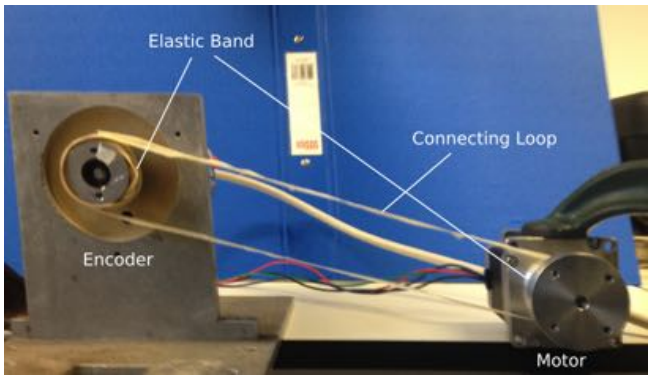


Figure 3.1.7: Image showing the experimental set up for testing the stepper motor output.

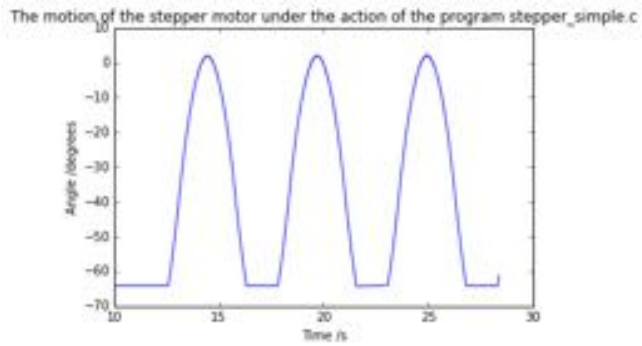


Figure 3.1.8: The motion of the stepper motor as part of the feasibility testing.

A Phidget stepper motor MKII-3306 with a holding torque of $9kgcm$ was chosen to perform the action . This motor was interfaced with a 1063 PhidgetStepper Bipolar 1-motor controller and it's assosiated C library --phidget21- which formed the basis the tests programming. [REF-Phidget library]

With instruction from the Phidget USB sensing and control API (see Legacy) the functions setCurrentLimit(), getAcceleration(), setVelocityLimit(), setTargetPosition() and getCurrentPosition() were implemented from the library in a program to control the motor entitled stepper_simple.c. When running the program the motor rotates periodically allowing the user to specify the maximum velocity, the acceleration, the number of cycles and the angle of rotation.

In order to understand the response of the motor when running this program the rotating shaft was coupled to the encoder with a "fan belt" style connecting loop (see 3.1.7. To prevent the loop from slipping rubber bands were placed at the points of contact and the loop was under sufficient tension.

The program was modified until the output was deemed an appropriate approximation to the robots knee motor swinging action. To produce the angle of motor rotation against time graph shown in Figure 3.1.8 the program was modifided such that the motor accelerates, reaching a target position of 60 degrees before reaching the maximum velocity. It then changes direction having the target position equal to the initial position where this repeats after a short delay.

A 25cm Aluminium rod of mass 135g was fixed to the motors rotating shaft at one of its ends to create a system with a moment of inertia of $8.4375x10^{-3}kgm^2$.

When attaching the motor to the swing, clamps were used to increase the load on the swing to $2.5kg$ this value was seen as appropriate as it is approximatly half of the robot's mass, $5.3kg$, and the system models only half of the propulsion, one leg.

With the experimental setup shown in 3.1.9 the system achieved small amplitudes of approximatly 2 degrees from rest. The result of the feasibility testing performed by other subgroups shifted the projects primary focus towards a soley standing robot swinging. This desicion was made before the encoder was mounted to the pivot as shown in Figure 3.1.11 and therefore the test was concluded before quantitative results could be obtained.

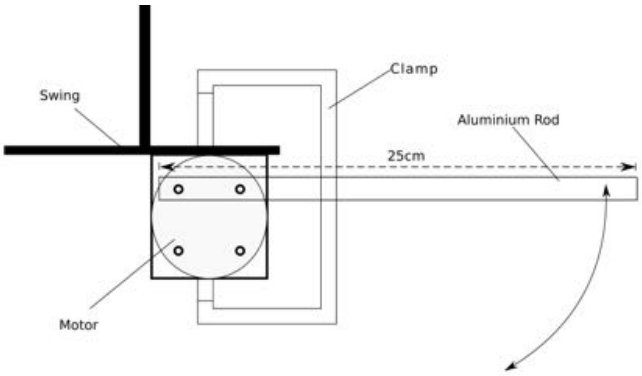


Figure 3.1.9: Diagram showing the experimental set up of the starting from rest feasibility study.

——— The following was contributed by: Will Etheridge ———

3.1.4.4 Method

In this experiment the encoder was interfaced to the computer via a USB-1208FS Management Computing data acquisition device. This device had 16 bits of digital I/O, 8 bits in port A and 8 bits in port B. The rotary encoder was wired up according to the encoder pin map shown in Figure 3.1.2 and was connected to the 1208FS device, configured for input, according to the the port map shown in Figure 3.1.10. The 1208FS device was then connected to a computer via a USB cable (1.0 to 2.0). A C program was written, called encoder.c, using the provided library for the 1208FS device to take the output of the encoder and convert into an angle. The library contained a function for getting the data from the device. The function, $pmd_digin(A)$ returned the eight bits connected to port A in the form of a 16-bit integer and $pmd_digin(B)$ returned the eight bits connected to port B also in the form of a 16-bit integer. The program created two 16-bit integers by calling the function with the arguments port A and port B. The output of the encoder was only eleven bits and therefore filled the eight inputs of port A and three from port B. These two integers were then combined into one by shifting the three bits from port B by eight to the right. This shifted integer and the integer formed from port A were combined by taking the logical OR of the two. This formed a new 16-bit integer containing the eleven bits of binary output from the encoder. A schematic of this process is shown in Figure 3.1.12. A calibration factor was then used to convert this into an angle. The program also contained logic to calibrate the angle to zero at a position chosen by the user.

The encoder was then mounted at the apex of the swing. The computer aided design for this mount is shown in Figure 3.1.11. To test the behaviour of the encoder, the encoder.c program was modified to log the angle and the time in an output file which allowed plotting. After running the program while the swing was swinging, the outputted data was plotted. This plot of angle against time, shown in Figure **??**, showed a correct sinusoidal shape, and also the attenuation due to damping. However, there were a few anomalous data points. The library used to gather the data from the 1208FS device was changed to a second library to eliminate anomalous readings. The reason for this step is explained in the diagnostics section. This new library allowed all 16 bits of the digital input to be gathered with one function call and placed straight into one 16-bit integer. Now fully functioning, the 1208FS device and connectors were housed in a casing for future use. The encoder was now ready for testing the damping coefficient of the swing.

For the sake of usability the C program was implemented as a Python module, *encodermodule.py*, for use in the motion programs to be run on the robot. The module provided, upon import, two functions; *getAngle*() and *calibrate_zero*(). The former returned the current angle of the encoder and the latter calibrated the current position of the encoder to zero. A test program was also provided, called encoder_example.py, which used the encoder module and produced an output log file with the angle and time.

The encoder module was then used to create a program called *network.py* that would allow the angle to be reached from any computer via a network connection. This was implemented in Python and a SyncManager was chosen as the method of networking the data. The SyncManager is a member of the Multiprocessing package in Python and allows a shared pool of variables to be accessed or manipulated by any computer connected to the

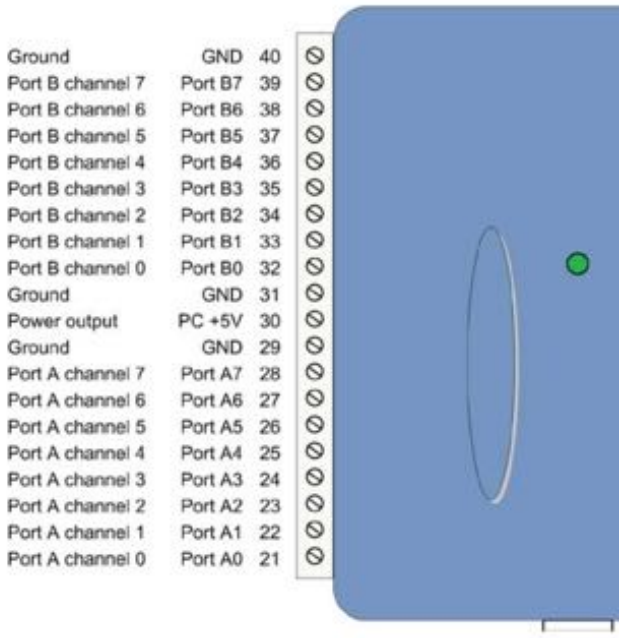


Figure 3.1.10: Map of USB-1208FS device for input functionality

manager. A client program called client.py was also provided to allow connection to the manager.

The following was contributed by: Will Edmondson

3.1.4.5 Diagnostics

This section is concerned with the significant challenges faced while building the hardware and software for the encoder. These challenges had a large effect on the decision making process throughout the method described above and, when unresolved, point to areas of focus for future further investigations.

Powering the Encoder The encoder was fitted to the swing pivot 2m above the ground of the laboratory and was connected to the pmd1208FS USB interface via a cable RS-232 serial cable 20 meters in length. This length of cable allowed access to all the laboratory computers. When used in the initial tests, the encoder was powered using the +5V DC output provided by pin 30 of the pmd1208FS USB device. When using the 20m cable the resistance was increased by 3.7Ω providing a current of 1.06A to the encoder which has its own internal resistance of 1Ω. This current proved insufficient to produce output readings. The problem was overcome by rewiring the voltage input to the encoder to and external power supply, as shown in Figure **??** which was required to supply between 6-7V for the encoder to produce an output reading. The output ground of the encoder, pin 15, needed to remain connected to the controller ground, pin 40, as its value is used as a reference of measuring the high and low voltages for the 11-bit binary inputs.

Computer Timing In attempting to record encoder angle against time within the program *encoder.c*, included in the digital appendix, a waiting function, usleep(), was initially used. This function was placed within a while loop along with an incrementing variable to measure the time each angle was recorded. The product of the number of loops, which the program had performed, and the length of the waiting time step provided a time stamp for each recorded angle. This is shown in the logic written in the C programming language below:

```
while(){
    int time_elapsed = 0;
        int i = 0;
        int increment = 1000; //1ms
```

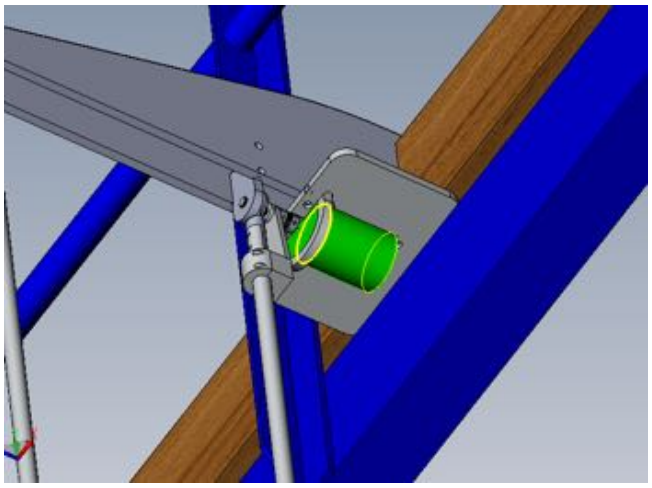


Figure 3.1.11

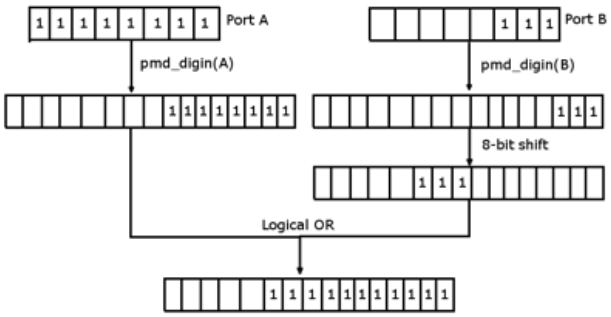


Figure 3.1.12

```
        usleep(increment)
        time_elapsed= i*increment;
        i++;
        //obtain angle, write angle and time_elapsed to file.
}
```

This method for recording time is based on the assumption that the computers processing time for the operations within the code and over all other programs is negligible when compared to the wait time. This was shown to be an unreasonable assumption when using timesteps of the order of milliseconds. The processor had an inconsistent processing time over the course of the tests and this made the recorded plots of angle against time distort, giving time periods of oscillation which were not consistant with observation. The initial resolution to this problem was to run the program with a higher task priority using the *nice* command. With reduced use of processing within other programs while *encoder.c* was running the processing time compared to the waiting time became less pronounced. Although this minimised the distortions it did not address the source of the timing issue.

Once the Python module was completed the method for acquiring time was changed within the program *encoder_example.py*. Implementing the Python standard library *time* the program was able to obtain the UNIX time stamp to sub millisecond accuracy. UNIX time is the universally used clock system for time syncronisation in communicating electronics, it is the number of seconds elapsed since the 1st of January 1970. The new time

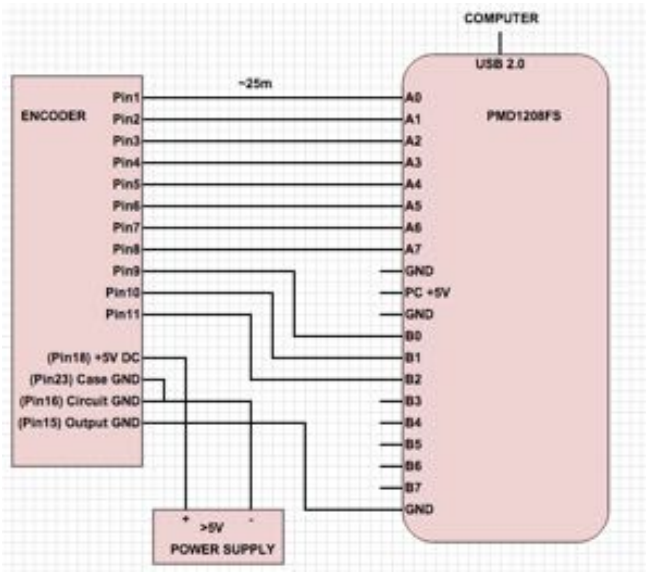


Figure 3.1.13: Schematic diagram showing the interface used to take measurements with the encoder.

measuring method logic is as shown below in the Python programming language:

```
import time
t0 = time.time()
for i in range(1, 30000)
    time_elapsed = time.time()-t0
    #record angle from encoder
    #write angle and time_elapsed to file
```

16 bit input The programs *encoder.c* and *encoder.py* were both built using the library pmd1208fs which permitted access to the function to read the bits from the USB controller. The function used to read the 11-bits from the controller was *digin('handle', 'port')* where the first argument for this function specifies which device is to be read and the second is specifies which 8-bit port in the device is to be read(see 3.1.10). To obtain the 11-bits from the USB device: port A, containing the least significant 8 bits, was first read and then in the next operation port B, in which the 3 most significant bits are contained, was read. The bits from port B were then appropriately shifted and the two bytes were summed to provide a 16 bit integer. This process is shown in Figure 3.1.12. It was found that while using a small timing increment and a high swing angular velocity the error show in Figure 3.1.14 was observed.

The error was a single point value and repeated at the same angle in the swings oscillation. The number of errors per oscillation increased at higher amplitudes and when the number of recordings per second was increased. For these reasons it was concluded that the error was systematic and the program encoder.c was modified to remove these values post collection using a simple *if* statement. Removal of these errors after they were recorded was acceptable when the data was being used to identify the damping coefficient after the test. However when using the encoder to provide real time feedback to the robot this data removal method was not applicable. Identification of the source of the error came from the analysis of the following raw binary data of the encoder over the following increments in which errors were observed.

As shown in the above table, the errors occurred during changes in the 3 most significant bits. The method of extracting the bits in two separate operations over the two ports was resulted in the summation of the port. A data form the first reading (either 255, 511 or 1535) and the port B data from the second reading (either 256, 512 or 768). As the error only occurred when the readings were taken between one of few consecutive angles this explanation accounts for the correlation between the number of errors per period and the amplitude. Similarly, as the error only occurred in the $< ms$ time between reading port A and port B, this explanation accounts for

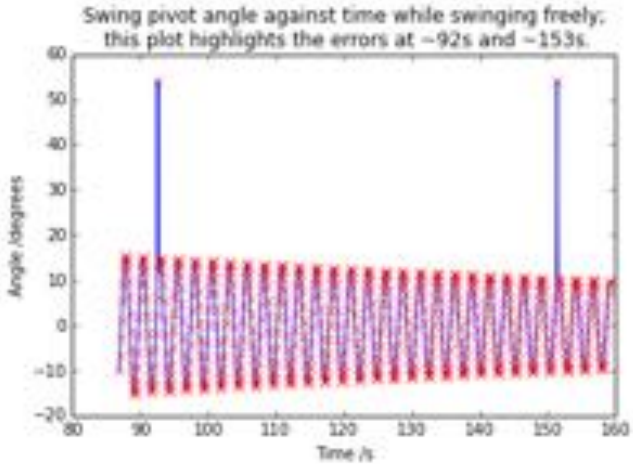


Figure 3.1.14: This data contains the error removed using the new pmd1208fs library function digin16()

Value errors occurred between	Binary output from encoder										
255	0	0	0	1	1	1	1	1	1	1	1
256	0	0	1	0	0	0	0	0	0	0	0
511	0	0	1	1	1	1	1	1	1	1	1
512	0	1	0	0	0	0	0	0	0	0	0
767	0	1	0	1	1	1	1	1	1	1	1
768	0	1	1	0	0	0	0	0	0	0	0

the inverse correlation between number of errors per period and time interval. To fix this problem, the library pmd1208fs was changed to allow a all 16 bits to be input in a single operation. The function for this operation was called *digin16('handle')* and took only one argument specifying the device.

Client/Server Speed The SyncManager developed for the encoder allowed access to a pool of methods and variables to those connected to the network established within the laboratory. The most applicable method to controlling the robot is the *getAngle()* method. When called from a client's computer over the network the return value of the encoder's current angle is presented at a maximum rate of approximately four times a second. This was deemed not fast enough for the feedback to the robot as the swing has a period of approximately 2 seconds and requires greater resolution of its position than 16 values per period. The SyncManager was rebuilt such that the method encoder.getAngle() did not search for the port containing the pmd1208FS device when called within a loop but this did not significantly improve the problem. The proposed and yet untested reason for the delay is the firewall active on the University computers. Disabling this firewall is unauthorised to students and the Sync-Manager can currently only be used for purposes with this time constraint.

3.2 Investigations into the Aldebaran Nao

The following was contributed by: Chloé Smith

3.2.1 Choregraphe

Choregraphe is NAO's "out of the box" software. It's a desktop application that allows the user to control NAO; and have it do low-level seemingly, simple tasks (until one discovers the "Tai-chi" movement). There are options to link these together, and therefore make the actions more complex. However, hard coding NAO's software *NAOqi* is a much more efficient way to do all these things, and has the added benefit of being cross language (i.e. one may code in C++ or Python).

It is strongly recommended though, even with its shortcomings, that students familiarise themselves with NAO via Choreographe. One would need to install Choreographe onto the laboratory computers or personal devices manually (it has multi-platform functionality).

A very helpful aspect of the software though is that you can control both the physical NAO (i.e. the one standing in front of you), and a virtual one on the screen. Both will do exactly the same thing. One group may need NAO for something, but another can test a Choreograph sequence on the virtual NAO to check functionality. There are options to move NAO's individual limbs, and rotate or elevate them using scrollers. In short it is a handy software to play with and see what the robot has to offer, but you will more than likely not be using it throughout the project as it progresses.

A guide to choreographing NAO and saving motion sequences via Choreograph is available in the **Legacy**.

——— The following was contributed by: Chris Smith ———

3.2.2 Connecting NAO to WiFi

NAO can be connected to both via an ethernet cable, or over WiFi. WiFi is the preferred connection method, as it allows multiple computers to connect simultaneously, and without any wires that may restrict NAO's movement.

In order to set up a WiFi connection for NAO, it first had to be connected to a computer via ethernet. The university's network does not allow NAO to be connected to either the main network, or individual computers. After a request to allow a method of connecting to NAO, a USB-Ethernet adapter (USB2.0 lan jp208) and router were obtained. The adapter allowed a single computer to make a wired connection to the router (without removing connection from the university network). The router was also used to make a wired connection to NAO, thus putting both the PC and NAO on the same wired network.

To then connect to NAO its IP address had to be obtained, which was achieved by pressing the button on NAO's chest, which causes it to say its IP address. When this IP address was entered into a web browser, this connected to NAO's web page. To access the page a user name and password are required, the default for both being 'nao'.

The WiFi connection was then set up from the networks page, by selecting the desired WiFi network from available networks, choosing connect and inputting the network password.

This then meant any computer that was connected to the same wireless network was able to access NAO, provided that they had its current IP address.

——— The following was contributed by: Vincent Fisher ———

3.2.3 Description of Strength Testing

One of the initial tests conducted on the robot was to examine its strength and see how it compares to what we would expect based on its technical specifications. This is an important investigation to carry out, ideally early on in the project, to ensure our aims and goals are feasible. It also gives an idea of how closely one can follow the theoretical values given by Aldebaran for this particular robot.

The aim is to test the motors which would be most used by NAO to perform swinging motions; these are the shoulder, elbow and knee motors. These motions were approximated into ones which we could test with the use of a newtonmeter clamped into position and executing them so the robot pulled on the meter, giving a reading of how much force could be exerted before a particular motor stalled. The motions were programmed in Choregraphe by setting the initial and final positions, then creating a box which simply moves between the two. Three separate tests were conducted:

1. Pull down test. This test was done using one arm, the set-up is as follows:
    - Create the desired motion in Choregraphe
    - Position NAO into a lying down position, with joints unstiffened.
    - Position the newtonmeter at the correct height above NAO, hanging vertically from a clamp.

- Lift NAO's arm up and attach his arm to the newtonmeter. Originally we planned on placing the hook of the meter through its hand when its grip was closed. However, we did not want to put too much pressure on the fingers, so instead we used a piece of string that tied tightly around the wrists and attached this to the hook instead.
- Stiffen all the joints and execute the movement, noting the results.

2. Pull up test. Similar to the previous test; however it attempted to get NAO to lift its own body weight, mimicking the pull-up exercise. The motion was programmed in the same manner and executed with NAO holding onto a pole which was suspended in the air.
3. Knee extension test. This tests the stall torque in the knee motor as it attempts to raise the lower leg from being perpendicular at the knee to the straight position. The set-up is similar to the pull down test:
   - Create the motion in Choregraphe
   - Position NAO into a sitting down position on the edge of a table, with its joints stiffened
   - Attach the newtonmeter horizontally under the table, so it is roughly parallel with the upper leg. We did this by clipping one end of the meter to a stool underneath the table.
   - Attach the other side of the meter to the bottom of NAO's leg. This was done using rope that tied around the ankle joint and connected back to the meter.
   - Execute the motion ensuring that the stool is adequately fastened so the motion does not pull it forward, ensuring all the force is exerted through the newtonmeter.
   - When the motor has stalled, record the force shown on the meter

3.3 Computational Methodology and Software

The following was contributed by: Peter Suttie

3.3.1 Choice of Programming Language

One of the first tasks for the Robot Programmers sub-team was to decide which programming language to use to program the NAO robot. The NAOqi software allows for several programming languages to be used, including C++, Python, .Net, Java, Matlab, and Urbi. However, the NAO robot can only run C++ and Python code natively, so realistically it was a choice between these two languages.

3.3.1.1 C++

The first inclination of the group was to use C++ as the main programming language for the project. This was because all of the group members had previous experience in writing code in C++. In addition, the example simulations and models in the Webots software are all written in C++, and this gave the group a good code base to build upon.

There are several technical reasons as to why C++ is a good language to use to write programs to run on the robot. C++ is a compiled language. Initially high-level source code is written by the programmer. A program called a compiler then translates this high-level code into machine code that an operating system can execute directly (known as object code) [27]. This process is shown in Figure 3.3.1.

The main advantage of using a compiled language like C++ is its speed; running programs written in C++ gives much better performance than those written in Python. This is because lots of optimisations can be made to the code as it is compiled that save time when the program is executed. For example, a compiler can verify the location of data once only during the compilation process and then safely use this location throughout, without having to repeatedly verify the location whenever the data is needed. In addition, once the source code is compiled, the machine code can be executed many times (it does not need to be compiled every time it is run). This saves time if the same program needs to be executed repeatedly, for example in performing multiple simulations [27]. The documentation for the NAOqi software states that the C++ framework is "the only framework that lets you write real-time code, running at high speed on the robot" [7]. However, for the purposes of this project, such high speeds would not necessarily be required.

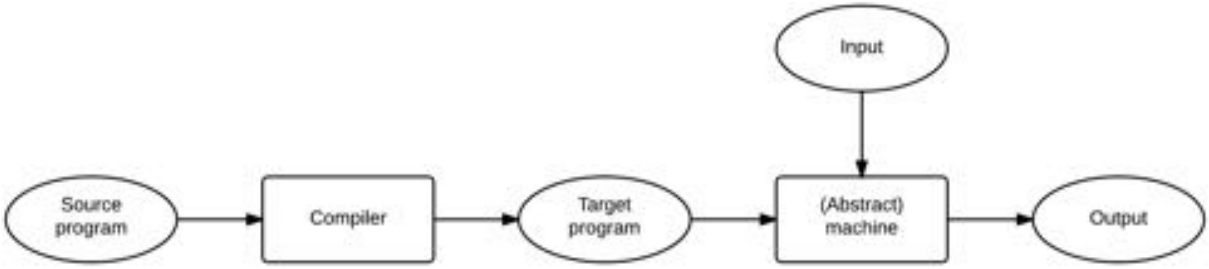


Figure 3.3.1: Compilation process [28]

3.3.1.2 Python

Python differs from C++ in that it is an interpreted, rather than compiled, language. The interpretation process is shown in Figure 3.3.2. Like C++, high-level source code is initially written. However, rather than being compiled, a program called an interpreter reads in the source code line by line and executes it on the fly.

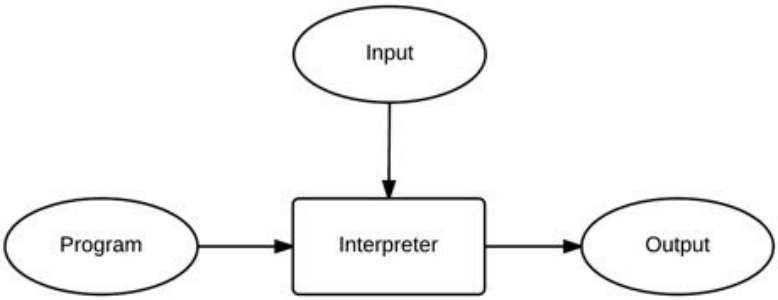


Figure 3.3.2: Interpretation process [28]

Interpreted languages like Python generally run slower than compiled languages like C++. This is because many of the code optimisations that a compiler can make by looking at the entire source code can not be made by an interpreter when it is executing code line by line [27]. However, there are several advantages to using an interpreted language like Python.

One of the main advantages is cross platform compatibility. When C++ code is compiled, it can typically only be executed on one operating system. Identical Python source code, however, can be run on any operating system for which a Python interpreter exists. This is very useful as it means any operating system can be used to develop and test code. Debugging is also a simpler process when using an interpreted language like Python, as the line by line execution of the code allows errors to be located easily [27].

A huge advantage of writing code specifically in Python compared to C++ is that the actual high level source code is significantly simpler and faster to write. This is due to the syntax of Python being less convoluted. For example, variable types do not need to be explicitly declared in Python; the type of a variable is evaluated by the interpreter as the program is run (variables are 'dynamically typed'). This slows down the program at run time but makes writing the program more efficient [24]. In fact, the Python documentation claims that equivalent programs written in Python and C++ are typically 5 to 10 times shorter in Python [11], and this was also found to be the case in this project.

3.3.1.3 Decision to Use Python

Whilst writing code in C++ would have given speed advantages, the group encountered problems in installing the NAO C++ SDK and compiling C++ code to run on the robot. A lot of time was being spent trying to get the installation to work, and eventually it was decided that it would be more productive to switch to programming in Python, as the installation of the SDK and setup was a lot simpler. Using Python also allowed the group to write a lot more code than if C++ was chosen as the code is a lot simpler to write in Python.

Migrating the code written by this group from Python to C++ would not be too complex if future years' groups wished to use C++. This is because the provided APIs for Python and C++ are close to identical. For example, all the class names and method names are the same whether programming with the C++ or Python SDK. This would allow future years' groups to write programs that could be executed faster than the Python programs that this group has written.

——— The following was contributed by: Joe Preece ———

3.3.2 Webots

Webots, developed by Cyberbotics Ltd., is a piece of software that enables the user to simulate a variety of robots in a virtual environment. The software is available for all major operating systems, making it accessible to numerous demographics. It allows for the user to program the simulations and environments in a number of mainstream programming languages, such as MatLab, C++, and Python.

The software comes with preinstalled robot models, but also gives the user the choice to design their own. The Aldebaran NAO is one of the preset robots available, making it easy to insert into any virtual world. There are demo worlds, but new environments can be built from scratch. The interface makes it easy for the user to visualise the world, whilst the scene tree allows for precise modifications to the physics and dynamical systems of anything within the world. The scene tree is a hierarchial list of all the objects in the world, which are named as nodes. Webots was designed with many different types of nodes, which may be used for different purposes (e.g. hinges, stiff walls, lights). Each node has settings that may be modified to make it suitable to the environment, and has the ability to attach child nodes, which in turn will be intrinsically linked to the sole parent node, as opposed to the entire world.

Webots uses an ODE (Open Dynamics Engine) library for the simulation of rigid-body objects. These objects may be a part of a robot, or part of the scenery and objects in the environment. The library allows for Webots to accurately simulate fundamental physical properties, such as inertia, velocity, and friction.

A robot controller is a script that can be associated to a robot within the virtual environment which tells it what to do. The Aldebaran NAO robot within Webots contains libraries that simulate the robot's detectors to access virtual telemetry. This information is crucial in simulations within virtual environments to detect if there may be any discrepancies in the real world, and can potentially prevent damage to an actual robot. Alongside access to these virtual detectors is a library of pre-coded motions, which can be used in robot controllers. New motions can be added to make more complicated scripts for the robot to perform. The more advanced controllers will utilise the appropriate libraries to collect required data prior to real life tests. Furthermore, Webots has the ability to take .png screenshots of the simulations and the ability to record .mpeg or .avi movies in addition to the data it collects.

The educational version of Webots, which was used in this project, was capable of allowing the group to design a primitive swing with rudimentary physical principles, and enabled us to explore more complicated possibilities, such as double pendula, and modified swings. The challenge was to design the swing in software that was completely new to us, and to then attach the robot to the different swings and perform tests. It would require experimenting with different programming languages, different Scene Tree formulations, and utilisation of complex programming techniques to ensure that webots matched our specifications precisely.

——— The following was contributed by: Michael Wright ———

3.3.2.1 Creating the Virtual Environment

In Webots, the Scene Tree contains a list of all objects in the world. In order to adequately simulate the system, the Scene Tree must first be assembled. Certain objects are provided pre-made and can be simply added. Custom, dynamic objects like the swing must be constructed. [16]

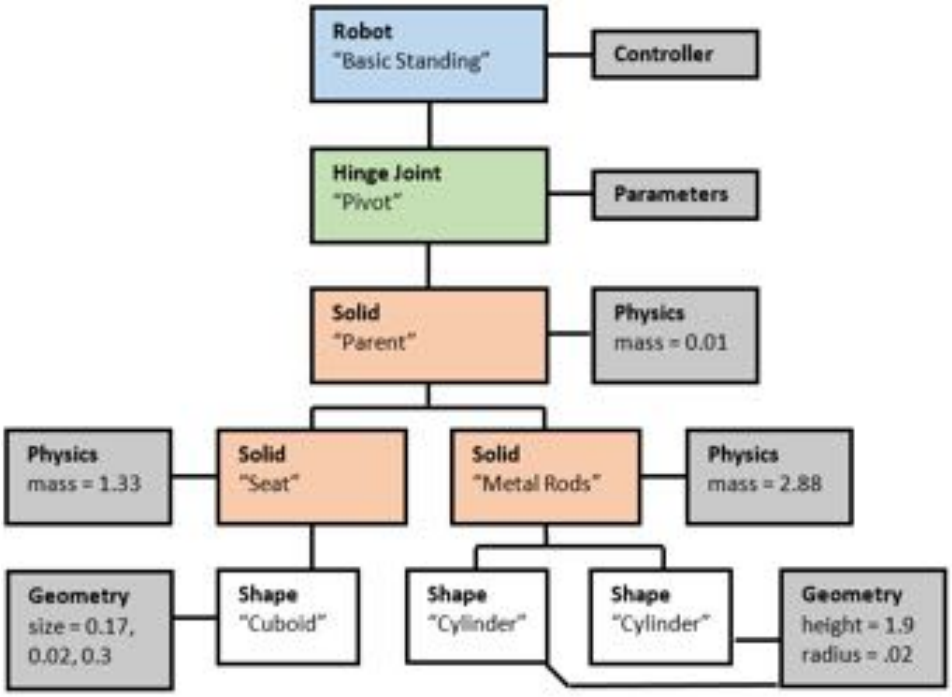


Figure 3.3.3: Scene Tree for the basic standing swing. For each node the node type and name is given. And up-down line represents a parent-child relationship, whereas a sideways node represents property setting.

Before constructing the environment specific to the project, it was first necessary to familiarise ourselves with Webots, and create the foundation world to build in; requiring NAO, floor and lighting. These steps can be found in the appendix - the Webots Guide.

The Swing Webots provides the tools for the construction of custom made robots, with these it is possible to create a dynamic swing object very similar to the real object, complete with hinges, friction and even the encoder.

Complex objects need to be constructed in the Scene Tree, making use of various types of nodes, each in a parent-child relationship with one another. The following types of nodes were used in constructing the Scene Tree: [15]

- **Solid** A node which can be given physics. Base class for all nodes which can interact with the environment.
- **Shape** Usually child to a solid node. Can take the form of a cuboid, cylinder, sphere etc. The dimensions and appearance can be set.
- **Robot** Derived from the solid node, but with more parameters. Can be given robot device nodes and a controller.
- **Joint** A node which links solid nodes together, but allows them to move with a specified number of degrees of freedom. The project makes use of HingeJoint nodes.
- **Device** An abstract node type, which can only be added as a child to robot nodes. Device nodes such as sensors and other robot parts which the controllers can make use of.

Three different swings were created in the virtual environment. The simplest of these was the basic standing swing provided from the start of the project. Weighing and measuring the individual components, allowed for an accurate recreation of the standing swing.

Figure 3.3.3 shows how the basic standing swing can be constructed and in Figure 3.3.4 it can be seen how this appears in the virtual environment. Originally the top “Basic Standing” node was a solid node, but it was

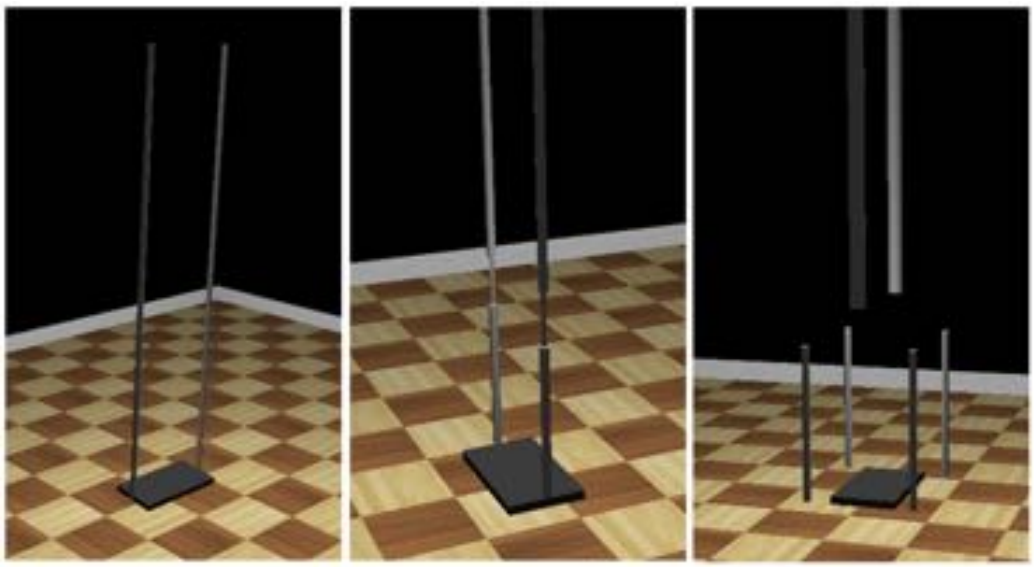


Figure 3.3.4: The 3 different swings designed in the virtual environment. From left to right; the basic standing swing, the new standing swing (triple pendulum) and the sitting cage swing.

later realised that the swing could make use of device nodes to represent the encoder. Giving this node no physics properties causes it to be suspended in the air, recreating the pivot point.

The Parent node keeps all individual parts of the swing rigid and stops them moving relative to one another. A shape node is required for each part of the swing that should be visible and interactable with the robot. For this to work, each solid's `boundingObject` needs to be set to be equal to the child shape. This is not shown in Figure 3.3.3 to avoid clutter.

This simple model is a good approximation of the real swing that was used, but there are a number of small differences. The seat of the real swing contains slot specifically designed to hold NAO's feet. This swing can be considered "perfect"; the joint only allows for a single degree of freedom. The real swing can oscillate sideways slightly, and can bend under weight. Finally, as seen in the next section, Webots only supports linear friction which is not completely realistic. [15]

One other method used to create the swing, was without the use of the hinge joint nodes. Instead various solid and shape nodes were used to create a mechanical joint (ie. a cylinder inside a torus). This method was shown to be unreliable; occasionally solid objects can slip through each other and it was also much more computationally intensive.

Given the designs for the modified sitting and standing swings, it was possible to construct them virtually in a similar manner to before, this time containing more pivot points. Since these new models were effectively double and triple pendulum swings, the resulting Scene Trees were much more complex. However Webots was still able to compute these without runtime problems. Seen in the caged swing above, certain parts of the swing were not included since they shouldn't interact with the robot and would only add to the processing requirements.

Once the new standing swing arrived, we were able to get the precise masses from the workshop data. However the seated swing masses had to be approximated because of its cancellation. It was still possible to do measurements on the seated swing virtually however, the Scene Tree designs for these 2 swings and their appearance in the VE can be seen in Figures 3.3.4 and 3.3.5.

Naturally these double and triple pendulums behaved very chaotically. It was hoped that with addition of friction terms and the robot's driving force they would behave more orderly.

Artificial Friction In order to accurately model the swing's behavior, a degree of friction must be added to the swing. Fortunately investigations were made by the hardware subgroup, making use of the encoder to see how the amplitude of the oscillation decayed over time. From this it is possible to calculate the linear friction coefficient. Since all of the joints used in the swings were similar, the assumption was made that all pivot had approximately equal friction values. However, the other subgroup took measurements of the amplitude decay

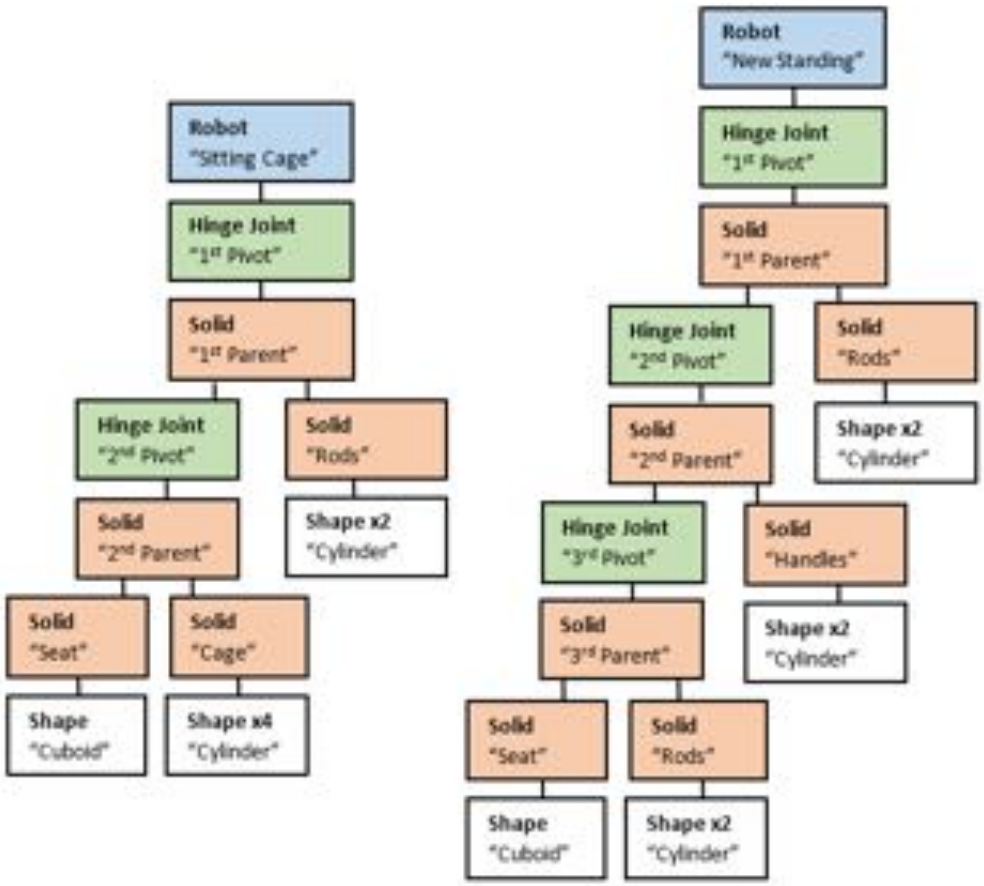


Figure 3.3.5: The Scene Tree constructions for the sitting cage swing and the new standing swing. The parameter nodes this time have not been shown to reduce clutter.

rate γ. So this needed relating to the linear friction coefficient μ. This can be shown with the following method.

The equation of motion for a pendulum with linear friction is given by the following;

$$F = ma = -mgsin(\theta) - \mu v \tag{3.3.1}$$

By making the small angle approximation, converting to radial coordinates and rearranging we get the following quadratic;

$$0 = \ddot{\theta} + \frac{\mu}{m}\dot{\theta} + \frac{g}{l}\theta \tag{3.3.2}$$

A decaying oscillation takes the form;

$$\theta = Ae^{-\gamma t}cos(\omega t) \tag{3.3.3}$$

Which we can substitute in to get a solution.

$$\dot{\theta} = Ae^{-\gamma t}[-\omega sin(\omega t) - \gamma cos(\omega t)]$$

$$\ddot{\theta} = Ae^{-\gamma t}[-\omega^2 cos(\omega t) + 2\omega\gamma sin(\omega t) + \gamma cos(\omega t)]$$

$$Ae^{-\gamma t}\left[\left(\gamma - \omega^2 - \frac{\gamma\mu}{m} + \frac{g}{l}\right)cos(\omega t) + \left(2\omega\gamma - \frac{\omega\mu}{m}\right)sin(\omega t)\right] = 0 \tag{3.3.4}$$

In order for this equation to be true, we know that the multiplier before the *sin* term must be zero, allowing the following relation to be deduced.

$$\mu = 2m\gamma \tag{3.3.5}$$

This tells us the two terms, γ and μ are linearly related, with a coefficient m, the mass of the swing system.

The value γ was provided, but this is only an estimate. The actual value is dependent on the current angle at which the swing is at. This is not something we cannot easily replicate in the virtual environment due to the fact we have made the assumption of only linear friction, but there is likely to be constant and quadratic factors.

We took the value of $\gamma = 0.012 \pm 0.002$ for when the swing is at about 15° as a sensible upper limit, as it is about this range where NAO's upper amplitude is achieved, and the friction will limit the motion. Setting $m = 9.41 \pm 0.1$, gives a value of $\mu = 0.226 \pm 0.044$. This value was to be used in all simulations.

——————— The following was contributed by: Fred Grover ———————

Recreating Encoder Similar to the real life set up, initial simulations showed an underlying problem with using the robot's built in gyro. This arose due to the positioning of the sensors in the body node meaning that when the robot began performing a swinging motion the gyro's sensor would pick up the movement leading to chaotic outputs. To make sure the robot only swung at the apexes of the swing as required the real life encoder would have to be replicated. This would be done by the conversion of the swing into a robot node, this was required to allow the addition of sensors to various parts of the swing. To replicate the encoder the swing would require the addition of a gyro attached to the seat to determine the angle. To allow the information of the swings current position to be communicated to the robot a transmitter would also have to be added. This would also necessitate a linked receiver built into the NAO robot to allow it to receive the data.

The emitters and receiver were set to communicate via radio communication. This would replicate the time delay expected by the use of radio communication so the distance between the emitter and receiver would have to be minimised to reduce the lag related to the system to the smallest value possible. The time delay related to the set up was measured by recording the time on the emitter's side and sending it to the receiver. When this was received it would then retrieve the current time. By then comparing the values that were recorded it was possible to see the delay attributed to this communication method. The value was shown to be precisely one time step of the wbt file, this would allow this delay to be accounted for and prevented from having any effect on the data retrieved.

Robot on the Swing The first task once the virtual environment was created involved the placing of the robot on the swing. This was done by the use of a controller file which would set up the robot's initial joint positions; after this was conducted the robot was then manually moved on to the swing via the adjustments to its translation points. Similar to what was experienced with the real robot, at high enough amplitudes the simulated NAO robot was unable to maintain grip on the swing. This occurred at different points on the two swings.

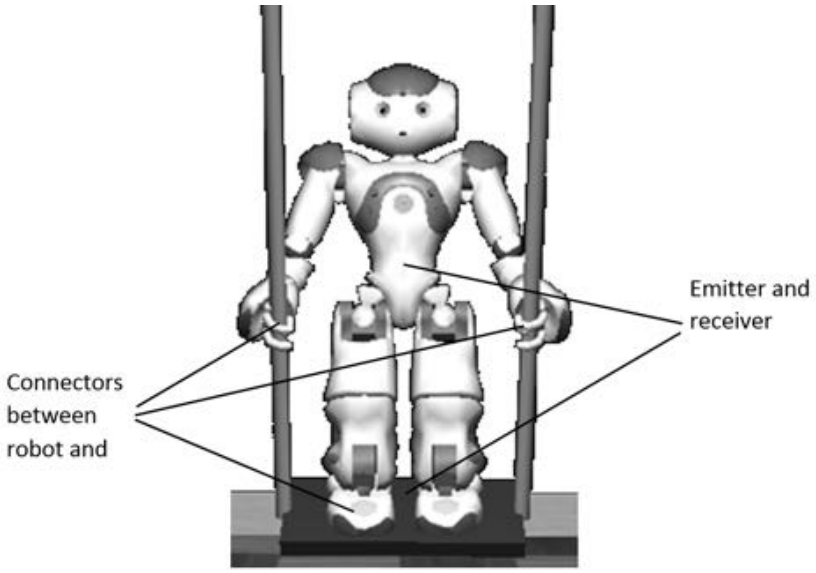


Figure 3.3.6: Robot and position of additional nodes

Sitting Swing For the sitting swing the main problem occurred with the robot sliding on the swing at higher amplitudes. To reduce this the friction between the seat and NAO was increased artificially high; this reduced any motion achieved by the robot to the swing. To reduce any lifting of the robot a lap bar was also placed over it, these factors would have to be replicated in real life when the seated swing is constructed in future years to prevent similar motion of the real robot.

Standing Swing On the standing swing loss of grip was experienced in two key points on the robot. The first part of the robot which was seen to have undesirable motions was the hands. The exact value at which the hands would lose grip would vary between simulations but was generally experienced at the limit of 0.18-0.22 radians. The loosening of the fingers around the swing's bars would cause the robot to flail and ultimately stop any swinging motion from occurring. The second part of the robot's connection to the swing which would fail was the position of the feet on the seat; this regularly occurred when the simulation would approach angles above 0.5 radians. At this point the robot would commonly exhibit slipping of the feet or being momentarily lifted off the swing. While these characteristics were solved by the addition of simple material, in the real set-up for the Webots simulation something more refined would be required. The simplest solution for this would be the use of connectors; these would be created as nodes which were children of both hands and feet as well as the corresponding objects on the swing to which the joints would be fixed. Each connector would activate as soon as its corresponding connector was present within in the 0.1 metres range of the device. At this point the connectors would fix the parent's position, locking the robot to the swing. The physical link formed was given an infinite strength; though unreasonable compared to the strength of the actual objects used to fix the robot to the swing, this was a necessary compromise for maintaining stability in the Webots' environment. The positions of the connectors, the emitters and the receivers can be seen on the robot and the swing in 3.3.6.

——————— The following was contributed by: Fred Grover ———————

Controllers All data gathering in the virtual environment and motions involved in the swinging of the robot were controlled by corresponding C files known as controllers. Separate controllers would be required in the initial set-up of the robot, swinging of the robot and data acquisition of the swing. Each controller would include the necessary device tags; each device tag corresponded to a child node which would be required to be utilised at run time.

To synchronise the run time of the controllers to each other and to the Webots' simulator a function wbrobotstep was used. This maintained the stability of the simulator by making sure each controller performed necessary actions at the correct time and allowed the controllers to successfully maintain communication between

each other. The majority of code in the controllers were set up to be executed in a single while loop. This loop would be performed every simulation step, and would retrieve data from the devices as required as well execute any motions or update any controllable objects.

Three controllers had to be made for the simulation. The first, the RobotPosition controller was used to set the position of the robot's joints so that it would be possible to place the robot. To do this the positional motion file was used along with opening of the hand joints. This motion sets the robot's joints to the positions that were determined by setting up the real NAO robot on the swing. This script was only need to be run initially; once the robot was placed on the swing the controller would no longer have a use and could be disregarded. The controller was the only script to be non-automated; instead to aid in the setting up of the robot each command was controlled by a key press - this allowed the robot to be moved into the correct position before the hands were and closed and the connector placed in the locked position.

The second key script was the NAO controller script. This script was involved in the process of making the robot perform the required motion. It was linked to the three motion files; one for positioning and the two for the swinging extremes. The initial part of the script involved removing the stiffness of the joints not involved in the swinging motions. This was needed to reduce the torque required to overcome each joint's movement, allowing a smoother and more efficient movement. The script's main purpose was the controlling of the robot and the correct timing for when it should begin its motion. To do this, the script was required to loop around; every simulation step it was required to check to see if any new data had been received from the swing controller, if this had occurred then the controller would begin the correct motion dependent on the data received.

The final and most complex of the scripts was related to the swing controller. This controller was in charge of the calculation of when the NAO robot should begin its motions as well as outputting the data achieved by the simulation. The swing controller worked by repeating a loop on each time step, with each iteration the angle would be recorded by the gyro sensor. The swing controller would use this data to determine its position in the period; when the correct criteria were met, the swing would send a signal to the robot controller with the corresponding information or record the amplitude achieved. The reason for doing the calculations on the side of swing controller rather then on the robot controller is because this minimises the amount of data that needs to be communicated, thereby increasing run speed which is vital in simulating the swing.

Controlling the robot was done via the use of small scripts called motion files. The motion file was simply a text file which stored a list of the joints which angles would be adjusted and how they would be modified. This was done by stating their initial and final positions and the duration which it would take to move from one to the other. This allowed the speed of motions to be adjusted easily and was the least intensive method for applying movement to the robot.

3.3.3 Python SDK

The following was contributed by: Daniel Tyrer

3.3.3.1 NAOqi [6]

NAOqi is the software that is executed on the robot and starts when NAO is switched on. It allows the user to control the robot by providing libraries with a large amount of built in functionality.

NAOqi Framework A programming framework is where the functionality of the software can be changed to fit the needs of a user. The NAOqi framework:

- **is Cross Platform** The software can be used on Windows, Linux and Mac, although only Linux operating systems allow for code to be run on the robot as well as the computer.
- **is Cross Language** The framework allows code to be written in both the C++ and Python languages.
- **provides Introspection** Introspection means that the robot knows exactly what methods are available in the API. Methods can be added to the API by using BIND_METHOD which is found in the ALModule header file. The main advantage to this is that you can tell methods to wait and stop, or check if they're running at a certain time. The API can be found by entering the robots IP address and port number into a web browser e.g. http://192.168.1.6:9559.

NAOqi Process When the robot is started, the NAOqi executable is run. This executable is what is known as a broker. A broker is responsible for dealing with the communication of a distributed software system. It co-ordinates messages sent between multiple remote objects that may have to interact synchronously. The NAOqi broker loads the autoload.ini preferences file upon starting. This file tells the robot which libraries to load into memory. The libraries contain a number of modules whose methods are then advertised by the broker. This is illustrated in Figure 3.3.7.

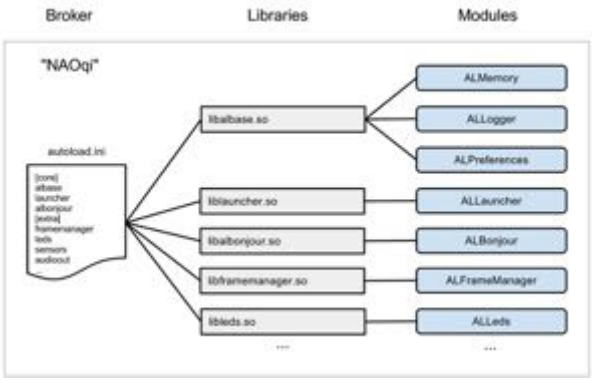


Figure 3.3.7: Diagram of the connection between the broker, libraries and modules

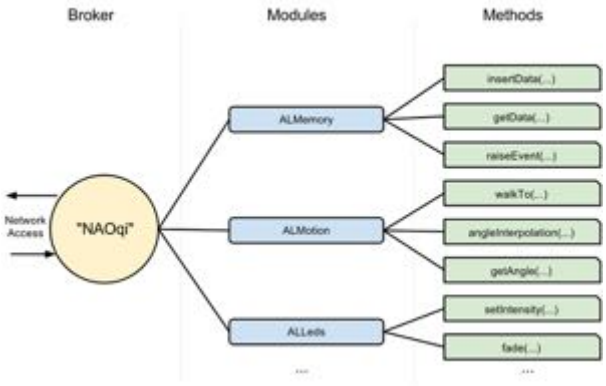


Figure 3.3.8: Diagram of the lookup tree of methods

The broker can be used to look across the network. This means that each module has the capability to find another module's methods, either in the module tree or across the network. Figure 3.3.8 depicts this.

Modules A module is usually a class, with methods, inside of a library. Modules can be either remote or local; this difference will be described in detail in the next section.

Remote Modules A remote module is compiled and run (or interpreted) from outside of the robot. These modules need to be connected to a broker via a network, to be able to communicate with other modules. Remote modules are easier to use and write, with errors being dealt with easily. The downsides to using a remote module are that it is much slower and uses more memory. Another negative is that direct memory access cannot be used. Remote modules can be connected to each other in two different ways, either by connecting their brokers directly or by connecting via a proxy. A direct connection allows for communication both ways between modules. A proxy to broker connection, on the other hand, only allows the proxy access to the broker's modules. The broker does not have access the other way.

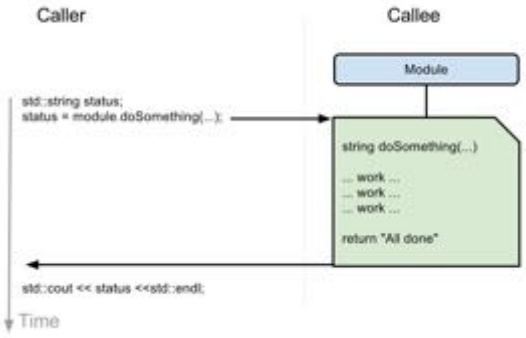


Figure 3.3.9: Diagram showing how blocking calls are executed over time

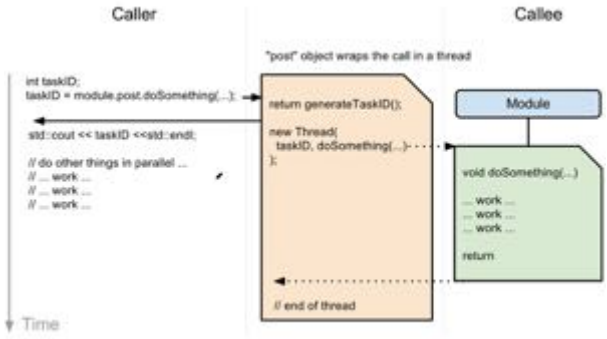


Figure 3.3.10: Diagram showing how non-blocking calls are executed over time

Local Modules Local modules communicate using the same broker. These modules can access each others variables and methods without connection to a network or serialization. This is because they are launched in the same process. Local modules are compiled as libraries and are only used on the robot. They are much faster and more efficient than remote modules. If any enslavement was looking to be achieved, local modules must be used.

Blocking and non-blocking calls A thread of execution is defined as the shortest sequence of instructions that can be managed by a scheduler. [13] NAOqi uses threads to allow two different ways of calling methods. They can either be blocking or non-blocking.

Blocking Calls Blocking calls are similar to a normal method call. The method calls are executed one by one after the previous method call has returned. This is shown in Figure 3.3.9. The method doSomething() is executed from the main thread. Before the std::cout command can be executed the program must wait for the doSomething() method to return “All done”.

Non-blocking calls Non-blocking calls use parallel threading. This allows more than one method to be run at a time. A post object is used to achieve this. When a post object is called, it creates a new thread for the task and generates a task id to identify it. The task id can be checked to see if a task is still running at any point. Figure 3.3.10 is an example of a non-blocking call.

The doSomething() method is called using the post object. This creates a parallel thread for the doSomething method to be executed in. Other jobs can be done in the main thread whilst the doSomething() method is being executed in the parallel thread. Once the doSomething() method has returned, the parallel thread ends and the program returns to being singly threaded.

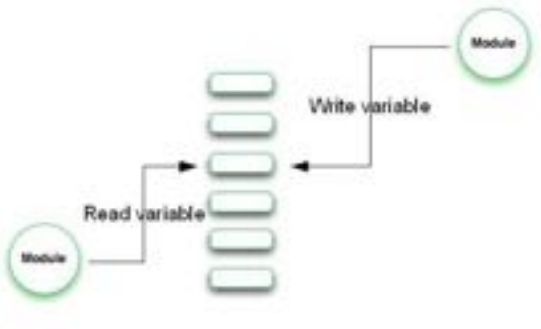


Figure 3.3.11: How memory can be read/written to ALMemory

Memory The robot's memory can be accessed by all modules. They can both read and write data into memory, shown in Figure 3.3.11. NAOqi also allows modules to subscribe to events so that the modules are called when the event is triggered. NAOqi stores data as instances of the ALValue class [2]. ALMemory is the array where these objects are stored. Manipulation of data can only be performed if it guarantees correct function over multiple threads. ALMemory stores; data from sensors and joints, events and micro-events.

——————— The following was contributed by: Peter Suttie ———————

3.3.3.2 NAOqi API

In this section the main features that were used from the NAOqi API will be described. As previously stated, the API is the same whether C++ or Python is chosen as the programming language. A full description of the API can be found in the online documentation and the disk supplied with the robot [5].

ALProxy The ALProxy class allows connections to be made to specific modules on the robot's operating system over a network. When an instance of this class is made, the name of the desired module on the robot, the robot's IP address, and the port number must be specified. The computer making the connection can then call methods contained in that module on the robot. Multiple connections can be made from several computers to the same module, which is useful in laboratory testing as several programs can be run on the robot at any one time.

ALMotion The ALMotion module contains methods which allow the robot to be moved. The main methods that were used were:

setStiffnesses() This method allows the stiffness of a specified joint or list of joints to be changed. Joints must be set to be stiff to be able to move them. It was also necessary to control the stiffnesses of joints when getting the robot into position on the swing.

setAngle() This method allows the angles of joints to be changed at a specified proportion of the maximum speed possible. This was used to control the motions of the robot required for swinging.

closeHand() This method was initially used to enable the robot to grip onto the swing.

getBodyNames() This method can be used to get the names of joints in the robot. This was useful for identifying the required names to move each part of the robot.

These methods were combined to build up motions for the robot.

ALMemory The ALMemory module contains methods which allow data to be taken from the robot. This module was used in the project to read gyroscope and accelerometer values, which could then be plotted and recorded, or used in the programs to identify when to perform certain motions. It was also used to read the battery level of the robot. The method used from this module was:

getData() This method allows data to be received from a specified file path on the robot.

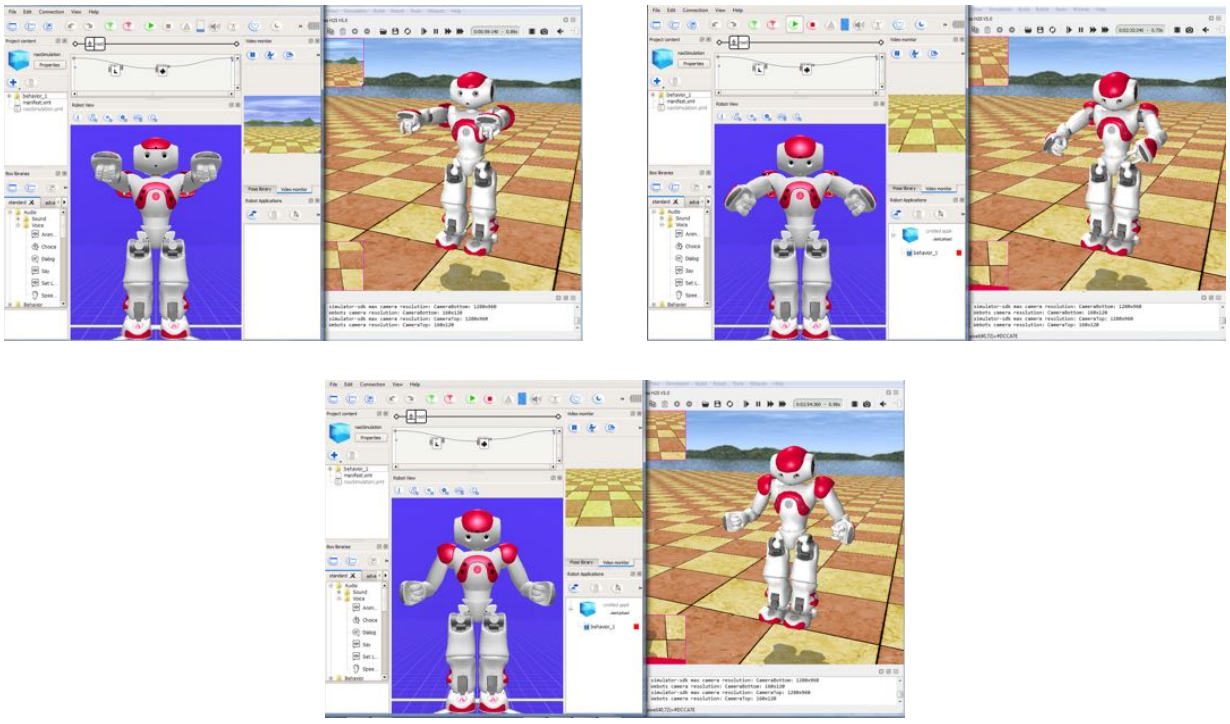


Figure 3.3.12: Stages of the custom `swingStandingInit()` motion running simultaneously in Choregraphe and Webots. This was originally coded in NAOqi and successfully executed on the robot

ALTextToSpeech The ALTextToSpeech module contains methods to control the vocal output of the robot. Methods in this module were typically used to audibly update the group on the status of the robot, for example by having the robot say 'top' when it was at the top of its swing, or by having the robot read out its battery level when it was getting low. The method used from this module was:

say() This method can be used to make the robot say a specified string.

——— The following was contributed by: Christopher Hounsom ———

3.3.3.3 Using Naoqi in Webots

The initial aim of designing and modelling the physical swing in Webots was to provide a simulation environment to test any NAOqi motions programmed for NAO. If set up, the environment would provide the means for a new motion to be tested significantly faster than if the real NAO were to be used. Additionally, an accurate simulation would ensure limited risk to the real NAO, as any dangerous motions would be identified on the virtual NAO.

There is, however, little correlation between the code used to write a Webots controller and that used in NAOqi. The Webots controller, for example, contains code exclusive to simulations (definition of time-steps etc.). In addition, as Webots motions began to be programmed using matrices in `.motion` files, the differences became too great for it to be viable to convert between the two environments.

As the Choregraphe boxes contain Python code [8], the use of Choregraphe as an intermediary between NAOqi and Webots was investigated. A connection could easily be made between the virtual NAO in Choregraphe and that in Webots [14]. After connection, pre-defined boxes in Choregraphe could be run and the Webots NAO would mirror the actions of the Choregraphe NAO.

Despite the use of Python in both Choregraphe and the NAOqi SDK, minor changes still had to be made to NAOqi code before it was executed correctly in Choregraphe and, consequently, Webots. Appendix A.2 describes how to set up a successful connection between Choregraphe and Webots and how NAOqi code should be integrated.

Although Figure 3.3.12 demonstrates that this method of simulation was successful, it was decided that this would not be investigated further. With the focus of the project switching to machine learning, the intended use of the Webots simulation was to provide an environment for the learning to take place without having to use the real NAO. Importing non-default Python packages in the Choregraphe boxes was not a possibility without further work, a feature that would no doubt be required for machine learning.

In addition, concerns were raised that the execution of the code in Choregraphe would suffer from slow-down once passed through Webots, a drawback that would have had serious implications on the effectiveness of any machine learning algorithm. As a result, it was decided to focus efforts on machine learning as its complexity was unknown.

It is not known how the virtual NAO would behave if placed in a Webots environment containing a model of a swing, as this was not tested. Further work could indeed show that the interaction between the virtual robot and swings in Webots is unsatisfactory. Nevertheless, the linking of Choregraphe and Webots could prove useful for the groups that continue this project. Undoubtedly, more complex motions will be programmed for NAO in the subsequent years of this project. Being able to visualise these on a virtual NAO instead of constantly requiring the actual robot is the main benefit of this investigation.

——— The following was contributed by: Daniel Tyrer ———

3.3.3.4 How the Code Works

Object-oriented Programming Object-oriented programming is the most widely used programming paradigm in the world. [32] It centres around the idea of objects. An object, in programming terms, is a data structure that consists of a number of different field variables or attributes. Objects can also contain multiple methods (or functions) that perform an operation to alter the state of the data structure. It is common practice for objects to be implemented as instances of classes. Every class must have a constructor. The constructor sets up the object to be used. Member variables are usually assigned from the arguments that it receives. To instantiate a class its constructor must be called.

There are many advantages to object-oriented programming [1]. The ability to reuse objects in multiple programs is a major benefit. This means that code will not have to be duplicated in more than one place. Using the concept of inheritance, objects that are closely related can reuse the same code that is derived from a parent (or super) class. An example of this is shown in Figure 3.3.13. The triangle and quadrilateral sub-classes both inherit the features of the shape super-class. The rectangle class can only inherit features from direct super-classes. It inherits features defined in both the shape and quadrilateral classes but does not derive anything from the triangle class. A variable that could be inherited from these classes is one called numberOfSides. Values assigned to this variable would be 3 for the triangle and 4 for the quadrilateral. This is then inherited by the right triangle and the rectangle which have the values 3 and 4 respectively.

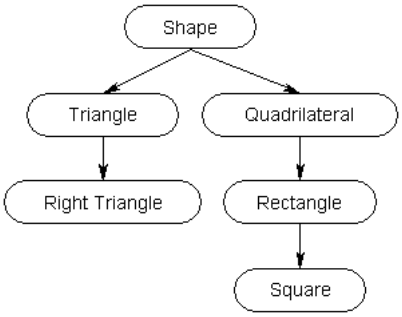


Figure 3.3.13: The inheritance of some shapes [3]

The shape class is what is known as an abstract class. A shape in itself is an abstract idea with no exact implementation or definition. The idea of an abstract method can be used to illustrate this. An abstract method is a method in a parent class that must be implemented in a child class derived from it. An example of this is could be one called calculateArea(). Every shape must be able to have it's area calculated. This cannot be done for a shape with no definition so cannot be implemented in the shape class. The RightTriangle and Rectangle

```
#import the Nao.py module
import Nao
if __name__=="__main__":
    ip="192.168.1.6"
    port=9559
    #initialise the nao1 object
    nao1=Nao.RemoteNao(ip,port)
    #initialise Nao to move
    nao1.motion.moveInit()
    #get the gyroscope data in the x direction
    nao1.memory.getData("Device/SubDeviceList/InertialSensor/GyrX/Sensor/Value")
    #make nao speak
    nao1.txtToSp.say("This example shows you how to initialise movement, get the gyroscope data and speak")
```

Figure 3.3.14: RemoteNao Example

classes have definite variables so the method can, and must, be implemented here. Functions in child classes can also override functions in parent classes. The name given to this concept is Polymorphism [18]. This is used when a method in a parent class is no longer functional for use in a child class and must be changed accordingly. Encapsulation is another great advantage of this approach, meaning that no knowledge of the implementation is required in order to use an object and its member functions. Documentation that is provided alongside each class helps to describe what each method does, but not how it does it. An object-oriented program is much easier to change and maintain than any other paradigm. [1] This means that a lot less effort has to be put into modifying the code to account for changes, in the long term.

RemoteNao Class One of the main goals of the project was to provide usable libraries that subsequent years can extend upon. The benefits of object-oriented programming are ideal for this goal. To implement this, the RemoteNao class was created. Inheritance can be used by succeeding years to extend the class and build more and more functionality. Polymorphism can be used to overwrite methods that have to be written differently for different tasks. It was decided that the Nao.py module that contains the RemoteNao class would be a remote module. As the project only runs for a single term, it was decided that the loss of efficiency, in speed and memory, was worth the time gained from easier use and debugging.

Using the Class In order to use the RemoteNao class the Nao.py module must first be imported. The constructor must then be called with the ip address and port number as arguments. These are stored as member variables. Required modules are then acquired by creating an ALProxy object. The ALProxy object mimics the module that it represents. Its methods can be accessed in the same way that a member function of a class instance is. These are stored as member variables so that any method of the module can be accessed when necessary. Methods and variables of the RemoteNao class can be accessed as shown in Figure 3.3.14. This also shows how to instantiate the RemoteNao class and import the Nao.py module.

Extending the Class How to extend the RemoteNao class is shown in Figure 3.3.15. The constructor for the RemoteNao super-class should be called first to re-use the construction process. Member variables that are assigned in the super-class constructor can be accessed as shown in the first line of method(). Extra NAOqi modules can be added to the member variables by creating a proxy in the usual way. Access to the super-classes methods is also shown in method(), with getGyroscopeValues() being called. Methods can be overridden by simply calling them the same name as the method you wish to override. This is shown with the swingStandingInit() method. The class is instantiated in main in the same way as before, with the ip and port given as arguments. Super-class methods can still be accessed from outside with the getGyroscopeValues() again being called in main. It also shows how new and overridden methods are called from outside the class.

——— The following was contributed by: Christopher Hounsom ———

3.3.4 PyBrain

In order to implement reinforcement learning, the Python package PyBrain was used. This section will document what PyBrain is and how it works as well as how it was utilised in order to create a reinforcement learning

```
import naoqi as nq
from Nao import RemoteNao

class RemoteNaoExtended(RemoteNao):

    def __init__(self,ip,port):
        #call the constructor of the RemoteNao class
        RemoteNao.__init__(ip,port)
        #get a new module using ALProxy that RemoteNao doesnt have
        self.module=nq.ALProxy("SomeALModule",ip,port)

    #Accessing a method from Super class to inherit code
    def method(self):
        #Accessing a field variable of the superclass
        RemoteNao.txtToSp.say("I can still access this variable")
        print RemoteNao.getGyroscopeValues()+2

    #overriding a method in SuperClass
    def swingStandingInit(self):
        #...
        #initialise the standing swing orientation differently here
        pass

if __name__=="__main__":
    ip="192.168.1.6"
    port=9559
    #initialise the nao1 object
    nao1=RemoteNaoExtended(ip,port)
    #call the getGyroscopeValues method from the super class
    nao1.getGyroscopeValues()
    #call the new method that has been written
    nao1.method()
    #call the overridden swingStandingInit() method
    nao1.swingStandingInit()
```

Figure 3.3.15: Extending RemoteNao

project that could be applied to NAO.

3.3.4.1 What is PyBrain?

PyBrain, as an acronym, stands for Python-Based Reinforcement Learning, Artificial Intelligence and Neural Network Library [17]. It provides the tools necessary to create a reinforcement learning (RL) system with as little difficulty as possible and, as its name suggests, can be used for many projects other than RL.

The system of a robot on a swing is very complex from an action/state point of view. The robot can exist in one of a continuous set of states (i.e. any combination of swing angle and robot motor positions) and can perform many actions (compound motion of any of NAO's motors). This results in a vast state-action map that requires significant computational power to access and edit. PyBrain's use of neural networks allows it to perform this computationally heavy task with relative ease [17].

3.3.4.2 How PyBrain works

The components that make up a PyBrain RL program are: an Experiment, Task, Environment and Agent. Figure 3.3.16 shows how these components are inter-linked and send/receive data to each other [10]. PyBrain also offers the ability to be used in conjunction with the Open Dynamics Engine, a physics engine that allows for a solid body system, and it's learning process, to be simulated. As successful machine learning often requires a significant period of learning, being able to simulate the process eliminates the dependence on the real system being set up, and will often save a considerable amount of time.

The reinforcement learning process occurs over several interactions. An interaction describes the process of an action being performed and the reward calculated. After an interaction, the system is reset and the process repeats.

Environment The Environment contains the definition of the 'world' that the project exists in. It describes the system and is altered by any actions carried out.

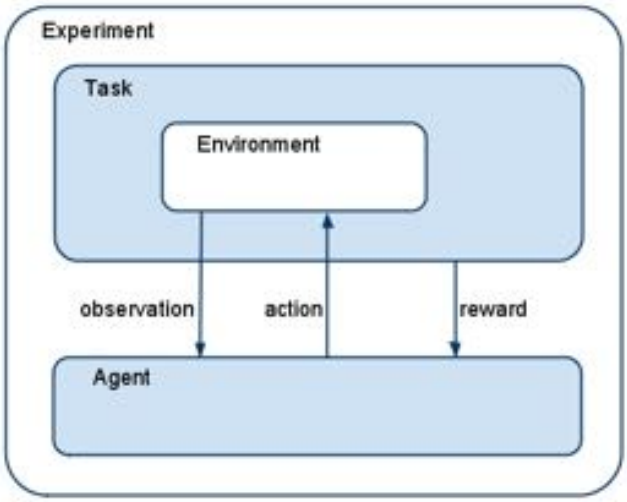


Figure 3.3.16: How the individual components of a PyBrain experiment communicate with each other

Agent The Agent is the component that performs actions on the Environment. In an attempt to maximise the reward that it receives after each interaction, the Agent alters its actions. An Agent is made up of three modules:

- **Controller** - Performs actions on the system
- **Learner** - Alters the Controller as a result of the feedback received from the reward function. The behaviour of the Learner is determined by the type of learning algorithm chosen.
- **Explorer** - Introduces the possibility of the Controller performing an exploratory action (i.e. an action that may not have been recommended by the Learner). This prevents the Learner from becoming fixated on a particular action that has returned a positive reward, as it is likely there will be other actions that return greater rewards.

Task The main role of the Task is to assess the Agent's performance. The Task defines the goal of the project and, through the reward function, evaluates whether or not the Agent is achieving this goal.

Experiment The Experiment oversees the whole project, ensuring continued communication between the Environment, Task and Agent. It passes the Environment's observations of the system to the Agent and notes the Agents corresponding action, returning this to the Environment. The Experiment decides when the interaction is finished, consequently taking the final reward from the Task and passing it the Agent so that it knows whether the previous action was effective or not.

Open Dynamics Engine (ODE) For physical systems such as pendulums, ODE can be used to model the Environment. As opposed to defining what states the system can exist in and the actions that can be carried out, PyBrain derives this information from the ODE model. This, in conjunction with the visual representation of the learning process, results in a simplified method of incorporating machine learning into what may at first seem like an impossibly complex system.

Several examples of using ODE were included with PyBrain. The most useful of these were the `johnnie` example, shown in Figure 3.3.17, and the `balance` example, where a driven pendulum learnt how to invert and balance itself in the upright position. These provided the basic structure required for a complete RL experiment.

3.3.4.3 Using PyBrain

A complete PyBrain project incorporating ODE requires the creation of 4 classes:

Figure 3.3.17: The `johnnie` example, one of many programs included with PyBrain that incorporate ODE. The robot learns how to position itself in order to maintain a stable stance.

MakeXODE (extends `XODEfile`). This class is where the ODE model is built, creating a `.xode` file. ODE models are created out of combinations of boxes and cylinders that can be attached together by a variety of joints. This process is detailed in Appendix A.3.

Environment Class (extends `ODEEnvironment`). The `.xode` file is imported into the environment and sensors applied to each body and joint. The sensors provide feedback on the Environment to the Task, thereby providing it with the means to calculate the reward for the Agent. In addition to sensors, actuators are also applied to each joint. Forces applied to the actuators provide the actions carried out on the system. The fraction of the total torque of the system (defined in the Task class) is specified for each joint as well as the high and low ranges for the angles that an applied torque can move an actuator through.

Task Class (extends `EpisodicTask`). The Task class defines the maximum torque of the system as well as simulation properties such as time step interval and interaction/episode length. The reward function is also contained within this class.

Main Class This combines the Task and Environment classes, creating and running the final experiment. The neural network, required for the state-action map, is initialised here along with the specification of the type of learning algorithm the Learner should use. Following the `johnnie` and `balance` examples, the PGPE learning algorithm was used for the Learner.

The comments in the RL files submitted alongside the report address the specifics of each class and their functions.

With all the classes complete and the `.xode` file created, the main class of the program can be run with Python, initiating the learning process. To observe the system whilst it is learning, the ODE Viewer must be run (details of this are found at the end of Appendix A.3).

It was decided that it would be beneficial to create a simpler reinforcement learning system first, in order to understand how to set up all of the components necessary for a PyBrain experiment. The knowledge gained from this simple example could then be transferred to the creation of the much more complex robot and swing system.

3.3.4.4 Simplified System - Flywheel Pendulum

The simplified system was chosen to be a flywheel pendulum as shown in Figure 3.3.18. The pendulum was created in ODE out of 3 objects: `arm` - a small box of unit mass fixed to the environment; `swing` - a long cylinder of unit mass attached to the `arm` by a hinge; `bob` - a long cuboid of mass 10 attached to the `swing` by a hinge. A torque could be applied at the joint between the `arm` and `bob`. Details of the creation of the model can be found in `pendMakeXODE.py` submitted alongside this report.

This system bore a close resemblance to the dumbbell pendulum examined theoretically. As the ideal motion for this system had been derived, it served as a point of comparison for any learnt motion that the flywheel system developed.

The main aim for this system was for it to learn when torques should be applied to the flywheel, and with how much force, in order to maximise the amplitude of the pendulum oscillation.

To reduce the time required for learning, the timestep was set to $dt = 0.01$ and a total of epiLen $= 2500$ timesteps were carried out each episode. This resulted in the simulation running for ~ 3 oscillation periods. This

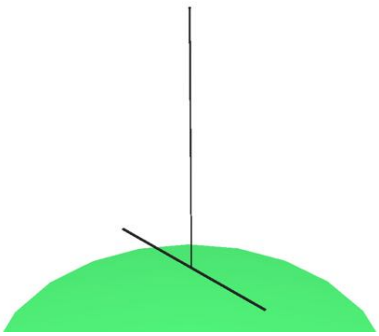


Figure 3.3.18: The flywheel pendulum ODE model undergoing the learning process

was favoured over simulation episodes that lasted many oscillations as the 'ideal' motion could be found for this shorter window in a much quicker time. If a longer experiment was required, the results from using the shorter window could be fed into the new experiment and used to provide a starting point for the longer investigations.

The simplest reward function tested was

$$R = |\theta|$$

where θ is the angle that the `swing` makes with the vertical. The PGPE parameters set were to `learningRate=0.2, sigmaLearningRate=0.1, epsilon=50` to encourage exploration. After a significant period (10000 episodes) no noticeable learning was observed.

The system developed a behaviour of applying a large initial torque to the flywheel and then ceasing to apply any further torques. The motion identified theoretically, involves applying a torque in the opposite direction to the motion at the extremities of the pendulum oscillation. The pendulum would have to apply torque in phase with the resonant frequency of the pendulum.

The exploration of the system infrequently resulted in the application of additional torques. Unfortunately, these torques only succeeded in removing energy from the system thereby giving low rewards. It is believed that this is due to the fact that, in order for the amplitude of the oscillations to increase, the subsequent torques must be applied at a precise time otherwise the effect is to reduce the amplitude.

In an attempt to overcome this fact, the learning rates were decreased so as to encourage the system to try more actions before settling on what it believes to be the ideal. Several different reward functions were also tested, and used various combinations of pendulum angle, velocity, position of the `bob` and angular velocity of the `bob` about the `swing`. One alternative reward function used was

$$R = |\theta| + 10v$$

where v is the angular velocity of the `bob` about the `swing` due to the applied torque. The v term was included in an attempt to encourage the system to apply torques as often as possible so that it would converge to the ideal motion faster. In spite of this, the system still displayed the same behaviour as before: applying a large initial torque and remaining stationary thereafter.

——— The following was contributed by: Peter Suttie ———

3.3.4.5 Standing NAO Swing in PyBrain

In order to implement machine learning on the robot, it was decided that a model of the robot and swing would be made using PyBrain. The aim of this was to model the physical robot and swing as closely as possible, and then apply a suitable learning function which would result in the robot learning to swing in the simulation. The resulting motions would then be recorded and supplied to the physical robot so that it could perform the actions.

Physical Model The first task was to create models of the swing and robot in ODE. This involved consultation with the Hardware group to gain information on the dimensions and masses of the parts of the swing, as well as looking in the documentation for NAO to find this information for the robot. The dimensions and masses can be

Table 3.3.1: Table showing dimensions and masses of swing components. Sizes are in metres but multiplied by 10 to make the bodies more visible in the simulation software.

Part name	Type	Size	Mass (kg)
seat	Box	(4, 0.2, 8)	2
lRodLower	Cylinder	(0.1, 3.3)	0.19
rRodLower	Cylinder	(0.1, 3.3)	0.19
lRodMiddle	Cylinder	(0.05, 1.2)	0.01
rRodMiddle	Cylinder	(0.05, 1.2)	0.01
lRodUpper	Cylinder	(0.1, 15)	0.81
upperSupport	Box	(4, 0.1, 2)	0.1

Table 3.3.2: Table showing dimensions and masses of NAO components. Sizes are in metres but multiplied by 10 to make the bodies more visible in the simulation software.

Part name	Type	Size	Mass (kg)
head	Cylinder	(1.2, 1.35)	0.64454
body	Cylinder	(1.1, 2.1)	1.18185
rPelvis	Cylinder	(0.9, 0.6)	0.140075
lPelvis	Cylinder	(0.9, 0.6)	0.140075
rUpperLeg	Box	(0.9, 1, 0.75)	0.459945
lUpperLeg	Box	(0.9, 1, 0.75)	0.459945
rLowerLeg	Box	(0.9, 1.029, 0.75)	0.3685
lLowerLeg	Box	(0.9, 1.029, 0.75)	0.3685
rFoot	Box	(0.9, 0.4519, 1.63)	0.23892
lFoot	Box	(0.9, 0.4519, 1.63)	0.23892
rUpperArm	Cylinder	(0.55, 1.05)	0.236705
lUpperArm	Cylinder	(0.55, 1.05)	0.236705
rLowerArm	Cylinder	(0.45, 0.5595)	0.110025
lLowerArm	Cylinder	(0.45, 0.5595)	0.110025
rHand	Cylinder	(0.45, 0.9)	0.18533
lHand	Cylinder	(0.45, 0.9)	0.18533

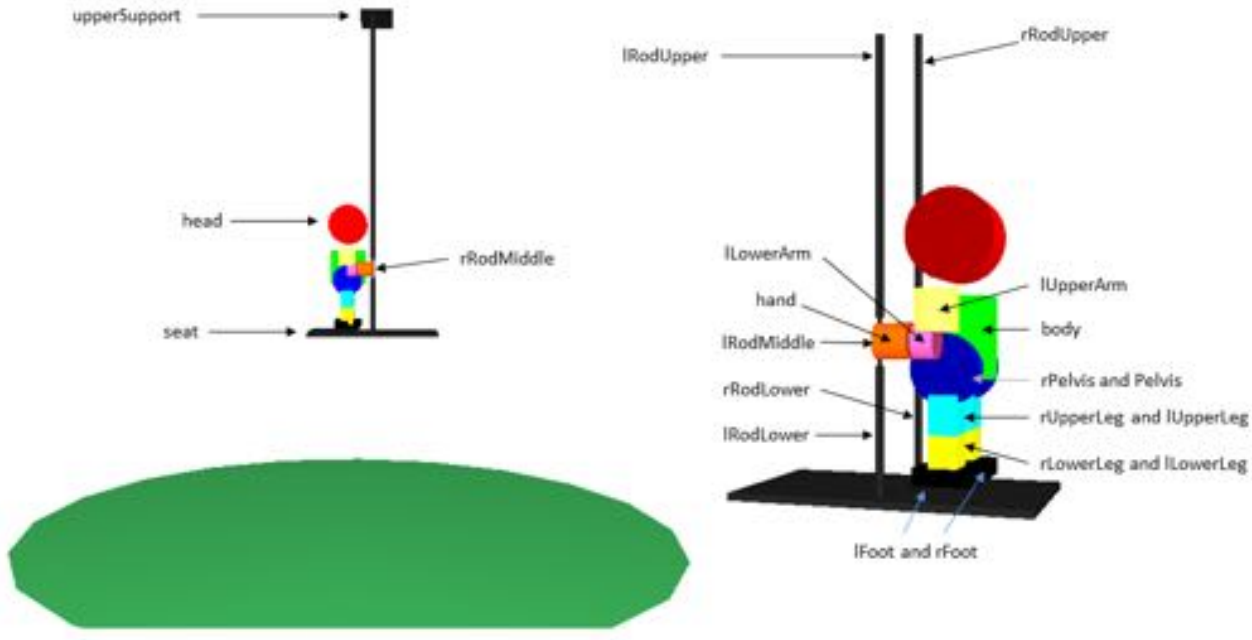


Figure 3.3.19: Diagram showing components of ODE model of swing and NAO

seen in Tables 3.3.1 and 3.3.2 respectively. The pivoting handle for the robot was included in the model; this can be set to be fully stiff if a fixed swing is required.

In ODE only boxes and cylinders are available to build models, which greatly limits how close the simulation model can get to the actual physical objects. The size of the swing seat also had to be enlarged in the ODE model to enable the NAO model to attach its feet in the simulation. The correct mass, however, was retained. Figure 3.3.19 shows the resulting ODE model created. Joints were made between each body to allow the model to move, with angle limits set to those of the actual robot.

Learning Once the model had been created, a task was developed with the hope that the robot would learn how to swing. Actuators were applied to each of the joints in the robot so that its constituent parts could be moved. PyBrain then effects random forces to random combinations of actuators to try to maximise the returned value of the reward function, shown in Equation 3.3.6. In this equation, rsa refers to the angle between rRodUpper and the vertical; lsa refers to the angle between lRodUpper and the vertical. The intention of this reward function was to maximise the displacement of the swing (either in the positive or negative direction, hence the top line of the fraction) whilst making sure that both of the uprights of the swing moved in the same direction (hence the division by the difference between the two angles). Movement of the uprights of the swing in different directions is possible due to the pivoting handle that the robot holds on to (see 'lRodMiddle' in Figure 3.3.19).

$$reward = \frac{|rsa + lsa|}{1 + |rsa - lsa|} \tag{3.3.6}$$

Results Currently only very slight movements of the robot in the simulation have been observed, with negligible swinging the result. The robot seems to be able to only move each of its joints very slightly which drastically limits to amount of swing movement that it achievable; it is currently unknown as to why this is the case as the code written to try to perform the learning is very similar to the fully functional example code supplied with PyBrain.

Due to the simulations not performing as well as expected, no results from simulations have been tested on the physical robot. It is hoped that groups in future years will be able to use the ODE models that have been built in this project to accomplish this.

The following was contributed by: Christopher Hounsom

3.3.4.6 Limitations

There are several factors that must be considered if results using PyBrain are to be transferred to the real NAO robot.

Firstly, the simulated ODE environments do not include friction at the joint or air resistance. In the `johnnie` example's Task class there exists the parameter `FricMu`, suggesting the friction of the system can be set. Fixing the `bob` and initially displacing the pendulum from the equilibrium position allowed the effect of this parameter to be investigated. The amplitude of the oscillations remained unchanged regardless of the value of `FricMu` (even when defined as a `string` the program ran suggesting that this parameter isn't used at all), thus indicating that no damping was occurring. The reluctance of the flywheel pendulum system to try any behaviour other than applying a large initial force is believed to be as a result of the lack of friction. As there is no damping, the system can apply the maximum possible torque initially and the system will continue to oscillate at that amplitude. This results in a consistent reward. Unless the reverse torque is applied at exactly the right moment, any additional torques will only reduce the energy in the system and hence reduce the angle made with the vertical. The system then concludes that it is more beneficial to continue with applying one initial torque.

In addition, the robot and swing models are not exact replicas of their real life equivalents. Therefore even if the simulation robot can be shown to learn how to swing, the results may not be immediately transferable to the real NAO. The simulation models could be improved further or, if significant progress is made allowing the real NAO to learn using PyBrain, the simulation results could be used as a starting point for NAO to improve upon.

The methods `saveWeights, saveResults` and `loadWeights` were found in the class `ExTools.py` and suggest learnt results can be saved and re-imported later. Additionally, the classes `NetworkWriter` and `NetworkReader` were found included in PyBrain and possibly present an additional way to save and import previously learnt behaviours. The use of both of these was briefly investigated but the ability to store and use the results from an existing reinforcement learning experiment was not accomplished.

If learnt behaviours can be saved, it is still unclear how the results can be interpreted and converted into a form that NAO can understand and act on. For example, the actions in ODE are random torques applied to the simulated actuators. It is not known how these actions can be related to the NAOqi motions contained in the `RemoteNao` class.

Instead of comprehending exactly how PyBrain and it's neural networks function, both the flywheel and robot swing systems borrowed heavily from the included PyBrain examples. If a greater understanding of PyBrain had been established initially, it is possible that the reasons why both systems aren't functioning as intended could be identified. Although the use of ODE makes PyBrain appear ideal for the task of teaching a pendulum-like system to maximise it's amplitude (easy implementation of laws of physics), the difficulties experienced in this aspect of the project have been numerous. An alternative machine learning library, not necessarily for Python, could prove more suitable for the task at hand.

Chapter 4

Analysis

The following was contributed by: Chris Smith

4.1 Results of NAO Strength Test

4.1.1 Arm Strength Results

The arm strength test showed that using one arm NAO was able to pull with a force of 20.5±0.5 N when its motors stalled, which is equivalent to lifting 2.1 Kg. Since NAO's mass is 5.2 Kg, this would suggest that using both arms NAO would be unable to hold its own weight.

However, since it was subsequently shown that NAO was capable of performing a pull up, this sets a lower limit on its strength of 25.5N per arm, as this is the minimum force required for NAO to lift itself off the ground.

To calculate a theoretical value for the force applied by NAO's motors, the following formula was used,

$$F = \frac{\tau r}{l} \tag{4.1.1}$$

Where F is the total force output, τ, is the motor torque, r is the motor speed reduction ratio, and l is the distance between the motor and the and the point at which the force is applied.

The error for this calculation is given by,

$$\sigma F = \sqrt{(\frac{r\sigma\tau}{l})^2 + (\frac{\tau\sigma r}{l})^2 + (\frac{\tau r\sigma l}{l^2})^2} \tag{4.1.2}$$

No error value for the speed reduction ratio is given however, removing the σr term in the final calculation.

Since the force measured in the experiment was when NAO's motors had stalled, τ is the stall torque of the motor. The motion in the experiment used 2 motors, the shoulder roll and elbow roll, the total force is the sum of both of their forces. However, the motion of the lower arm changes the distance between the shoulder and hand, causing l to decrease, and have a constantly increasing force.

The simplest value to calculate is when the arm is straight, which will also give the lowest force output, and thus calculate the lower bound fro NAO's arm strength.

Both of motors have a stall torque of 14.3 mNm ±8%, and a speed reduction ratio of 173.22. [9] NAO's lower arm is 126 mm and it upper arm 120 mm, giving a total arm length of 246 mm.

Putting these values into equations 4.1.1 and 4.1.2, gives a total force of 29.7±1.8 N per arm. This would allow NAO to lift 6.1 Kg, which is sufficient for NAO to hold itself up.

Possible reasons for the discrepancy between the strength test results, and the theoretical value could be that the newton meter used to measure the force was connected to one of NAO's fingers, which can be bent slightly even when tensed.

A final note should be made that when making NAO perform pull ups, it's hands lost grip in some of the latter attempts, causing it to let go of the bar it was holding. It is believed that this was due to NAO's hand motors overheating, which sufficiently degraded the motor's performance, such that NAO was unable to maintain it's grip.

4.1.2 Arm Strength Effects

The results of the arm strength tests, showed that NAO is capable of lifting its own weight using its arms alone, meaning that there were no special considerations for NAO's strength that needed to be made when creating a swinging motion, as it can do any necessary motions under its own power.

However the loss of grip in the hands required that NAO's hands must be fixed to the swing poles to prevent it from falling off. As its hands are fixed in position, this means that any action needing the hands to move could not be performed.

4.1.3 Leg Strength Test Results

The results of the Leg strength test showed that one of NAO's legs was able to pull with a force of 19.5±0.5 N, equivalent to being able to lift 2.0 Kg.

The theoretical value for leg strength is calculated in the same way as for the arm strength, but only one motor is used in this case, the knee pitch. The stall torque of the knee pitch motor is 68 mNm ±8%, and a speed reduction ratio of 130.85. The lower leg length is 115 mm.

Putting these values into equations 4.1.1 and 4.1.2, gives a total force of 77.4±6.3 N. This value is significantly higher than the measured value, though still of the same order of magnitude.

Some of the difference could be accounted for in the string attached to NAO's leg stretching, though the effect would be very small. Also, the arm test had the newton meter pulled in straight line, whereas the leg would have moved in an arc, but the newton meter remained in the same location, so the direction the of force applied may not have been parallel to the newton meter, which would also have resulted in a lower reading.

4.1.4 Leg Strength Effects

The results from the leg strength tests were planed to be used as part of the feasibility study for the robot swing whilst sitting, as well as any simulations of the robot. However, since the sitting swing modifications were never finished, this proved unnecessary, so had no impact on the project as a whole.

——————— The following was contributed by: Tom Crossland ———————

4.2 Analysing Fundamental Physics of Operating a Swing

Whilst operating a swing, a human performs several different actions simultaneously. The body and legs pivot about the swing seat, and the exact form of this motion causes a change in the centre of mass. Indeed, when a swing is operated by a standing human, the changes in centre of mass relative to the swing pivot appear to be far more significant to the motion of the swing than any rotation about the swing seat. To gain a better understanding of the physics involved in the successful operation of a swing, the movements of the operator will be broken down into the rotation about the swing seat, and change in centre of mass, mentioned above.

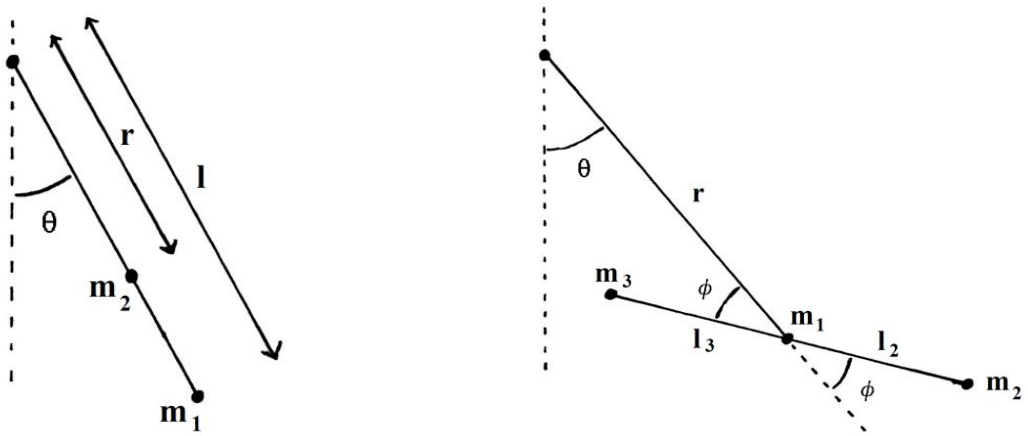
To simplify the models we shall use to examine these motions, the swing itself will be treated as a mathematical pendulum, i.e. a point mass attached to a light, rigid, inextensible rod. It is considered that this approximation will not affect the underlying physics of a playground swing, as it will hold for small oscillations where the rope used for the swing supports experiences little curvature.

The varying centre of mass model will be constructed as a mathematical pendulum, with a second point mass which may move along the pendulum rod. The rotation model will use a dumbbell comprised of light, rigid rod with a point mass fixed at either end. This dumbbell will then be affixed, somewhere along its length, to the bottom of a mathematical pendulum by a pivot (this setup has been used by others to model the human body on a playground swing [20]). For illustrations of these models, see Figure 4.2.1.

4.2.1 Mathematical Setup

4.2.1.1 Oscillating Mass Model

First we shall consider the varying centre of mass: If the length of the pendulum is l, and the distance of the second mass from the pendulum pivot is $r(t)$, then for the system shown in Figure 4.2.1a, the Lagrangian is



(a) Oscillating mass model.

(b) Rotating dumbbell model.

Figure 4.2.1: Illustrations of the models to be mathematically examined in Section 4.2.

given by,

$$L = \frac{1}{2}m_1 l^2\dot{\theta}^2 + \frac{1}{2}m_2\left(\dot{r}^2 + r^2\dot{\theta}^2\right) + g\cos\theta\left(m_1 l + m_2 r\right). \tag{4.2.1}$$

The Lagrange multiplier approach will be used to find the equations of motion, treating r as a generalised coordinate, and using the constraint equation,

$$G = r - r_{driven}\left(t\right) = 0.$$

From this setup, we find that the equation of motion for θ is,

$$\ddot{\theta} = \frac{-g\sin\theta\left(m_1 l + m_2 r\right) - 2m_2 r\dot{r}\dot{\theta}}{m_1 l^2 + m_2 r^2}. \tag{4.2.2}$$

The Lagrange multiplier, λ, is also used to find the generalised force on r, given by,

$$\tau = -gm_2 cos\theta + m_2\ddot{r} - m_2 r\dot{\theta}^2. \tag{4.2.3}$$

It is noted that as r changes, both the centre of mass and the moment of inertia of the pendulum as a whole vary. The moment of inertia of the pendulum is given by,

$$I\left(t\right) = m_1 l^2 + m_2 r^2,$$

and the time derivative of this is given by,

$$\dot{I} = 2m_2 r\dot{r}.$$

It may be seen that this expression appears in the equation of motion above. Also, there exists a relation,

$$r_c M = lm_1 + rm_2,$$

where $M = m_1 + m_2$ and r_c is the distance from the pendulum pivot to the centre of mass of the system at a given time. If these two are substituted into Equation 4.2.2, we get,

$$\ddot{\theta} = \frac{-Mgr_c\sin\theta - \dot{I}\dot{\theta}}{I}. \tag{4.2.4}$$

This simplified form allows us to see which properties of the system (centre of mass and moment of inertia) affect the individual terms in the equation of motion, allowing for a greater understanding of the physics involved. Notably, it is not only the position of the centre of mass which affects the motion of the pendulum, but also the way in which the moment of inertia changes (or the speed of the centre of mass, $\dot{r}$).

4.2.1.2 Rotating Dumbbell Model

Considering the system shown in Figure 4.2.1b, if the length of the pendulum is r, the distances from the mass, m_1, at the end of the pendulum to the masses m_2 and m_3 are l_2 and l_3 respectively, and the angle between the pendulum rod and dumbbell is given by ϕ, as in the figure, then the Lagrangian for the system is given by,

$$L = \frac{1}{2}Mr^2\dot{\theta}^2 + \frac{1}{2}I_0\left(\dot{\theta}+\dot{\phi}\right)^2 + \rho\left(\dot{\theta}+\dot{\phi}\right)\dot{\theta}r\cos\phi + Mgr\cos\theta + g\rho\cos\left(\theta+\phi\right), \tag{4.2.5}$$

where,

$$I_0 = m_2{l_2}^2 + m_3{l_3}^2,$$
$$\rho = m_2l_2 - m_3l_3,$$
$$M = m_1 + m_2 + m_3.$$

Again using Lagrange multipliers, with the constraint equation,

$$G = \phi - \phi_{driven}\left(t\right) = 0,$$

we find that the equation of motion for the θ is given by,

$$\ddot{\theta} = \frac{-Mgr\sin\theta - g\rho\sin\left(\theta+\phi\right) - I_0\ddot{\phi} + \rho r\dot{\phi}\left(2\dot{\theta}+\dot{\phi}\right)\sin\phi - \rho r\ddot{\phi}\cos\phi}{Mr^2 + I_0 + 2\rho r\cos\phi}. \tag{4.2.6}$$

However, the equation above is too complex for simple analysis, and so we shall simplify the system by assuming that the dumbbell is symmetric about the end of the pendulum, such that,

$$m_1 = m_2 = m_3 = m,$$

$$l_2 = l_3 = l.$$

With these changes, the equation becomes,

$$\ddot{\theta} = \frac{-Mgr\sin\theta - I_0\ddot{\phi}}{Mr^2 + I_0}, \tag{4.2.7}$$

with the generalised force (torque) on ϕ being given by,

$$\tau = \lambda\frac{\partial g}{\partial\phi} = \frac{6m^2l^2r\left(\ddot{\phi}r - g\sin\theta\right)}{Mr^2 + I_0}. \tag{4.2.8}$$

For this system involving the symmetric dumbbell, the centre of mass of the pendulum and dumbbell remains at a constant distance from the pendulum pivot, i.e. it always lies at the same point as the mass m_1. Also, the moment of inertia about the pendulum pivot remains constant, and is given by,

$$I = 2ml^2 + 3mr^2.$$

Hence, for this setup, it is only the motion (specifically, the acceleration, $\ddot{\phi}$) of the dumbbell relative to the pendulum which affects the motion of the pendulum, not the orientation of the dumbbell. Naturally, for an asymmetric system, the orientation of the dumbbell will affect the equation of motion, but for our purposes these effects will be dealt with using the changes in centre of mass and moment of inertia of system described in Section 4.2.1.1.

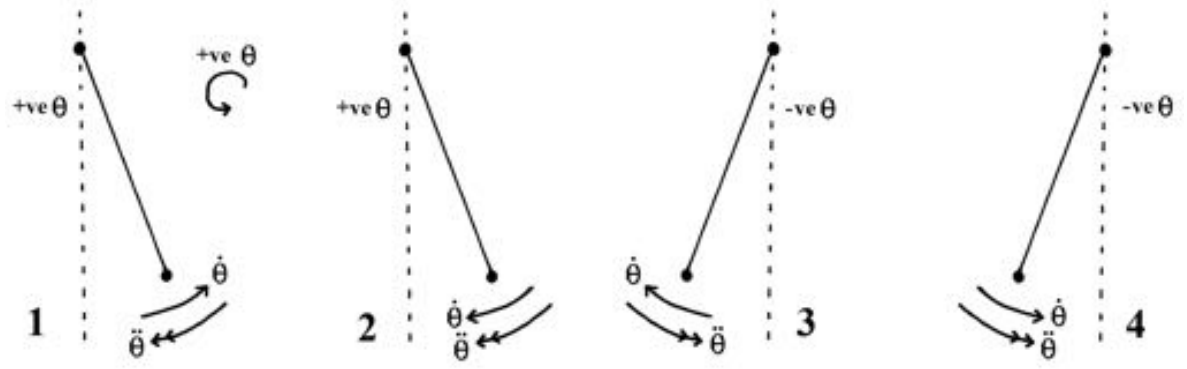


Figure 4.2.2: Possible states of motion when operating a swing, along with the sign convention for angular displacements (anti-clockwise displacement is considered to be positive).

4.2.2 Physical Analysis of Mathematics

When operating a swing, there are four distinct phases to the motion, as illustrated in Figure 4.2.2. Upon inspection, we see that 1 & 3 and 2 & 4 are symmetric, and hence we need only consider two stages of motion – swinging towards and away from equilibrium.

If we assume that the individual operating a swing wishes to undergo the largest possible oscillations, then we may determine the motion which will produce such oscillations. If we take the sign conventions given in Figure 4.2.2, we see that in stage 1, for an non-driven pendulum, the angular acceleration of the pendulum ($\ddot{\theta}$) will be negative. However, as the pendulum is still in an upwards swing, the operator should move so as to reduce the magnitude of this acceleration (or even produce a positive value) so as to achieve the greatest amplitude possible. During stage 2, $\ddot{\theta}$ will still be negative. However, the operator will now wish the magnitude of this acceleration to be as great as possible, as they will have entered the downward swing of their motion, and wish to have the greatest possible velocity upon reaching the lowest point of the swing. From this, we can deduce the optimal forms of motion for the operator, with regard to the two models described in Sections 4.2.1.1 & 4.2.1.2.

4.2.2.1 Analysis of Oscillating Mass Model

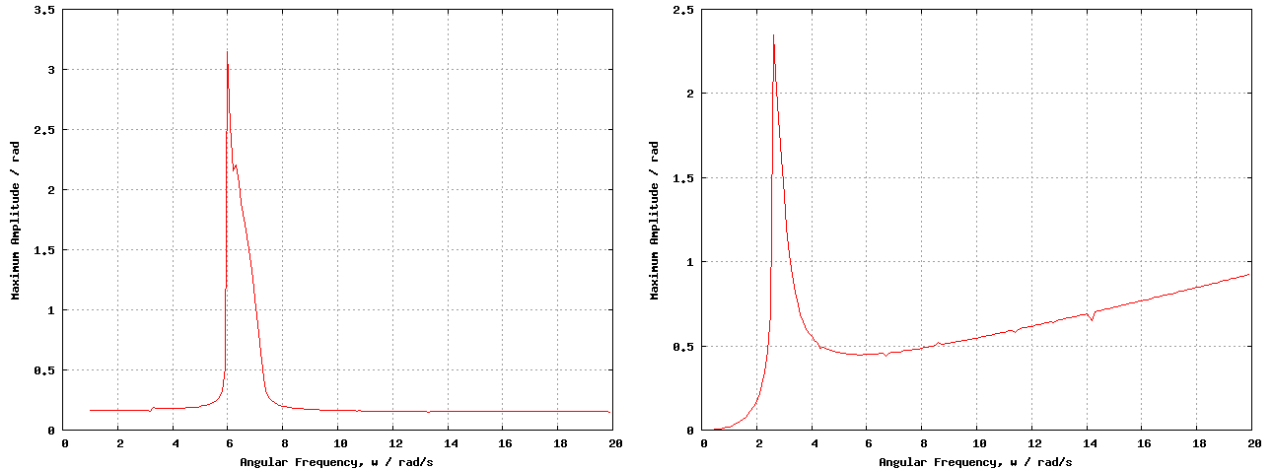
If we take the case of the system described in Section 4.2.1.1, where a mass moves along the rod of a mathematical pendulum, we have the equation of motion given by Equation 4.2.2. If we consider this equation to be of the form,

$$\frac{A+B}{C}$$

then we may see that term C will always have the same sign (positive), and that we cannot influence the sign of term A, as the only variable under our control (r) will always be positive. Hence, term B, and specifically the $\dot{r}$ variable, is the only parameter under our control which will have a sizeable impact on the nature of the motion.

If we consider stage 1, we see that term A is negative, and hence we desire term B to be positive (to reduce the deceleration). This suggests that $\dot{r}$ should be negative, i.e. the mass should be moving towards the pendulum pivot (note the sign of $\dot{\theta}$). If we next consider stage 2, we see that the first term is again negative, and hence we desire $\dot{r}$ to also be negative once again.

Unfortunately, this appears to indicate that the moveable mass always wants to be moving towards the pendulum pivot to increase the magnitude of the oscillations. This is an unhelpful answer, but one to be expected: consider the figure skater pulling in their arms to spin faster. Such a motion will always produce a greater angular velocity, regardless of position, due to conservation of angular momentum. This particular problem shall be addressed in Section 4.2.4, using a numerical simulation of the system.



(a) Oscillating mass model with linear actuator.

(b) Rotating dumbbell model using a rotational actuator.

Figure 4.2.3: Resonance curves showing maximum achieved amplitude against angular frequency of driving actuator.

4.2.2.2 Analysis of Dumbbell Model

If we next consider the system described in Section 4.2.1.2, with a dumbbell pivoted symmetrically on a pendulum shaft, we again encounter an equation of motion of the form,

$$\frac{A+B}{C},$$

which is given by Equation 4.2.7. In this instance, we see that term C is a positive constant, and that the only term we may influence is term B, through the $\ddot{\phi}$ variable.

Considering stage 1, and using similar logic to before, we see that we desire $\ddot{\phi}$ to have a negative sign. In stage 2, we desire $\ddot{\phi}$ to have a positive sign. Noting the state of the system during these stages, this suggests that the largest oscillations are achieved when,

$$\begin{aligned} +\dot{\theta} &\rightarrow -\ddot{\phi} \\ -\dot{\theta} &\rightarrow +\ddot{\phi}. \end{aligned} \tag{4.2.9}$$

This shall be examined further in Section 4.2.4 using numerical modelling techniques.

4.2.3 Numerical Modelling of Simplified Systems

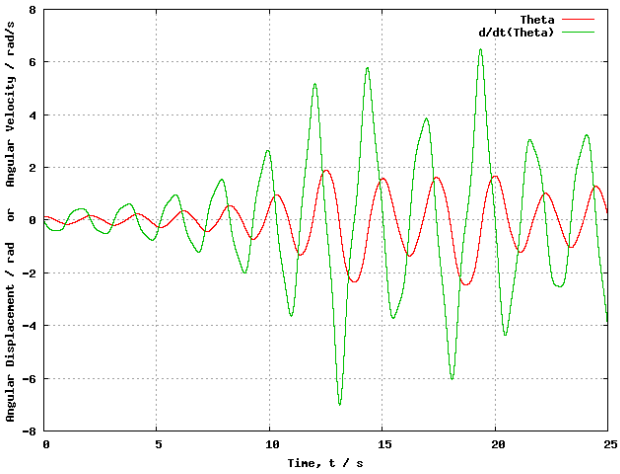
Models were created using the equations of motion found in Section 4.2.1. This allowed for verification of the mathematics, by producing both graphical data, and animated diagrams of the motion (see attached digital materials), and examining these for overtly unphysical behaviour.

To begin, sinusoidal functions are chosen to describe the motions of the dumbbell ($\phi(t)$) and point mass ($r(t)$), of the form,

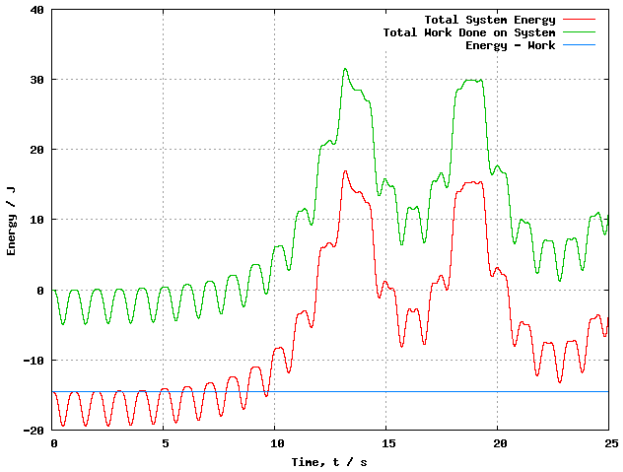
$$x(t) = A\sin(\omega t) + x_0,$$

where A is the amplitude of the motion, ω is the angular frequency, and x_0 is some constant displacement term. Once the models were built and tested using these functions, resonance plots were produced for both systems, whereby the angular frequency, ω, of the sinusoidal functions was varied, and the maximum amplitude achieved during a $25s$ run recorded. These produced the plots shown in Figure 4.2.3. It is interesting to note that each resonance plot shows only one strong peak, indicating that resonance cannot be achieved at multiples of some base frequency.

The angular frequency used in the simulation runs which produced the largest angular displacement was recorded, and used to conduct single runs of the simulations (note that numerical values included here are for



(a) Plot showing time evolution of θ and $\dot{\theta}$.



(b) Plot showing time evolution of total kinetic and potential energies of the system, total work done on the system, and the conserved total energy of the system.

Figure 4.2.4: Data collected from oscillating mass model simulation, from a 25s run.

interest only, and do not, in fact, have any physical significance). These individual runs were used to record data about the motion, energy, and work done in the system (recall the use of Lagrange multipliers to calculate the work done by the "motors").

Figure 4.2.4 show the data collected from the oscillating mass model. Figure 4.2.5 show the data collected from the dumbbell model.

An interesting feature of both runs is the growth and then decay of the oscillations. It is particularly noticeable in Figure 4.2.4a, where the steady increase in amplitude may be seen, followed by a sudden drop in achieved displacement. If the corresponding energy-time graph is examined (Figure 4.2.4b), we see that this rise and fall is mirrored by the energy of the system. The cause of this can be seen in the corresponding decrease of total work done on the system – implying that work is being done to take energy out of the system. The motion of the dumbbell/mass has fallen out of phase with the motion of the pendulum. We may understand this be considering that no small angle approximations were made in the creation of these models, and hence the period of the pendulum will begin to depend on displacement once the pendulum achieves a certain amplitude. As such, the angular frequency is no longer a resonant frequency for the system, and the motion decays back down to a state in which resonance can be achieved and the oscillations begin to build once more.

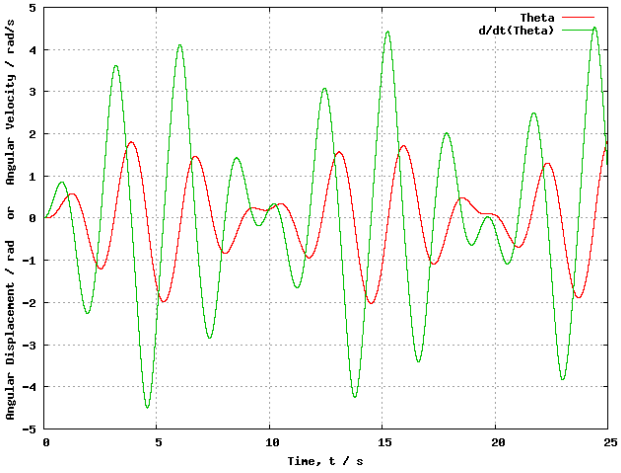
4.2.3.1 Error Analysis of Numerical Models

Error plots were produced for both simulations, showing the time evolution of the discrepancy between initial and current energy, as described in Section 2.1.3.3. Figure 4.2.6a shows the resulting plot for the oscillating mass simulation, and Figure 4.2.6b the plot for the dumbbell model. No simulation incurred an error greater than 0.5%.

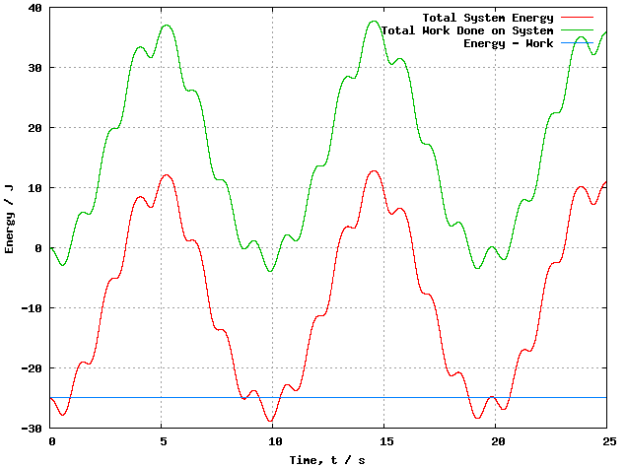
4.2.4 Motion Conclusions and Algorithms

4.2.4.1 Oscillating Mass Model

Upon examination of a visual simulation of the oscillating mass model, it was noted that the resonant frequency found previously produced distinct behaviour relating to the stages discussed in Section 4.2.2. Specifically, it was noted that $\dot{r}$ is negative during stage 1, but positive during stage 2. The exact reasoning for this relation is unclear, but it is suspected that it pertains to the dominance of the gravity term in the equation of motion (Equation 4.2.2) during stage 2, i.e. the optimum motion (negative $\dot{r}$) has a greater impact when conducted during stage 1.

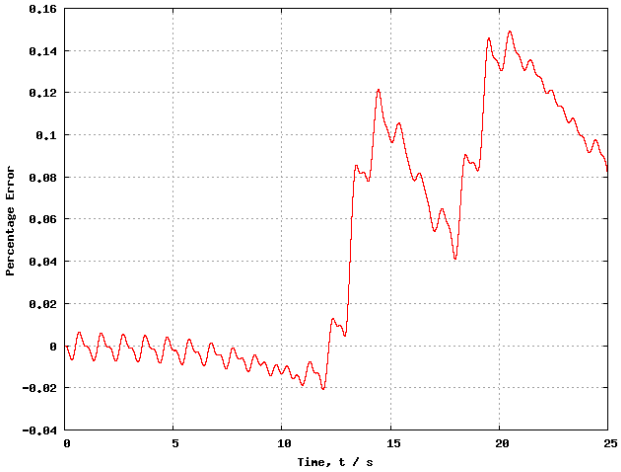


(a) Plot showing time evolution of θ and $\dot{\theta}$.

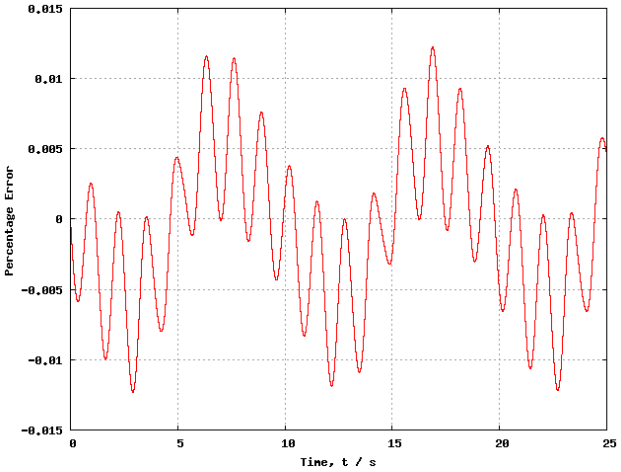


(b) Plot showing time evolution of total kinetic and potential energies of the system, total work done on the system, and the conserved total energy of the system.

Figure 4.2.5: Data collected from rotating dumbbell model simulation, from a 25s run.

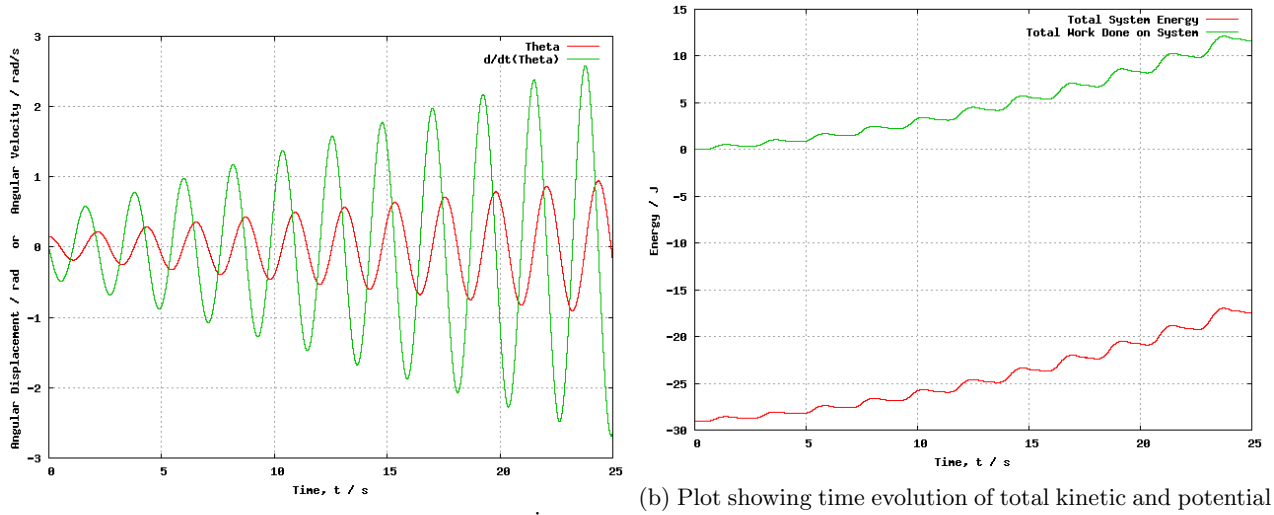


(a) Oscillating mass model, Figure 4.2.4.



(b) Rotating dumbbell model, Figure 4.2.5.

Figure 4.2.6: Plots showing time evolution of percentage error for the 25s simulation runs shown in Figures 4.2.4 & 4.2.5.



(a) Plot showing time evolution of θ and $\dot{\theta}$.

(b) Plot showing time evolution of total kinetic and potential energies of the system, and total work done on the system.

Figure 4.2.7: Motion and energy plots for 25s simulation run for the rotating dumbbell model, using the relation given by Equation 4.2.9.

(a) Schematic diagram of double pendulum model.

(b) Screenshot from the visual simulation.

Figure 4.2.8: Diagrams pertaining to double pendulum model used to examine the relation given by Equation 4.2.9.

4.2.4.2 Rotating Dumbbell Model

Additionally, the relation found in Section 4.2.2.2 for the dumbbell model, relating $\ddot{\phi}$ and $\dot{\theta}$, was included in a simulation run and examined, whereby the acceleration of the dumbbell was set to take some value whose sign was dependant on the velocity of the pendulum, such that,

$$+\dot{\theta} \rightarrow -\ddot{\phi}$$

$$-\dot{\theta} \rightarrow +\ddot{\phi}.$$

The motion of the pendulum for this simulation is shown in Figure 4.2.7. The simulation in question showed a maximum error of 0.0003%. The continual growth in amplitude and velocity is noted, and appears to support the hypothesis that the relation found leads to larger oscillations. Note that if linear friction (of the form $\mu\dot{q}$) is included, the system achieves and maintains a stable amplitude.

One further test of this principle was conducted, using a new model consisting of a double pendulum with an actuator at the joint (see Figure 4.2.8). The same algorithm was applied, and the results visually examined. Once again, the amplitude of the pendulum oscillations increased with time. This indicates the general application of the relation to different rotating systems.

It is concluded that this arrangement causes the motor to be continually pumping energy into the system, without ever doing "negative" work (i.e. work which removes energy from the system).

4.3 Analysis of Swinging

The following was contributed by: Ashe Morgan

4.3.1 Analysis of Human Swing

4.3.1.1 Introduction

Two different programs were used throughout the duration of the Group Study, which were both named Tracker. To differentiate between the two programs, the first program used, that was pre-installed for the Group Study and used on the laboratory Unix computer, shall be termed *Tracker*. The second program, which was downloaded later on during the Group Study and installed on a personal laptop, shall be termed *Tracker (D. Brown)* [19].

4.3.1.2 Data Analysis

MATLAB R2009b was used to create a script (see Appendix A.1) to import, process and export the tracking data for analysis using a separate MATLAB script. The data obtained via *Tracker*, as detailed in Section 3.1.3 was in the form of a tab delimited .txt file. The file consisted of a time column and nine further columns specifying the x-position, y-position and percentage of the total image occupied for each of the three markers. The percentage columns were not relevant and were deleted. The *Tracker* program occasionally lost track of a marker and recorded the position as -1.0000 by default. Any row of data that contained a value of -1.0000 for any co-ordinate was deleted to make sure only correct data was analysed.

The placement of the tracking markers on the person being tracked can be seen in Figure 4.3.1. The position of the swing was subtracted from each marker position to obtain the position of the two markers relative to the swing as a function of time. The time stamp and x and y positions of each marker were exported to a .csv file to be analysed.

A new MATLAB script was created to calculate θ_1 and θ_2, as well as creating an animation of the human motion relative to the swing. The lengths L_1, L_2 and L_3 were measured experimentally. The distance L_1 was calculated in pixels and used to calculate a calibration scale to convert between pixels and metres. The lengths r_1 and r_2 were calculated in pixels using MATLAB, and converted to meters using the calibration scale. The angle ϕ was known from θ_1, as L_2 and L_3 were measured (within error) to be the same length, thus giving an isosceles triangle. The angles θ_1 and θ_2 were calculated using Equation 4.3.1 and Equation 4.3.2 respectively and exported to a .csv file.

$$\theta_1 = \cos^{-1}\left[\frac{{L_2}^2 + {L_3}^2 - {r_1}^2}{2L_2L_3}\right] \tag{4.3.1}$$

$$\theta_2 = \cos^{-1}\left[\frac{{r_1}^2 + {L_1}^2 - {r_2}^2}{2L_1r_1}\right] - \phi \tag{4.3.2}$$

The animation of the motion relative to the swing was exported as a video file, as can be seen on the legacy CD. A comparison of a video of a human swinging and this animation was created and exported as a separate video file.

4.3.1.3 Results and Discussion

The relationship between θ_1 and θ_2 was emphasised as being an important factor to investigate by the Mathematics and Simulation sub-group leader. The *Tracker* data was processed and a graph of θ_1 and θ_2 produced, as can be seen in Fig 4.3.2 (θ_1 corresponds to the 'Hip Angle' and θ_2 the 'Underneath Knee Angle') . This was for a human swinging naturally on a rope swing from rest.

It appears that θ_1 and θ_2 are in phase so that a human could be considered to 'open' and 'close' using one co-ordinate to represent the 'straightening' of the body. Towards t=20s the hip angle starts to become difficult

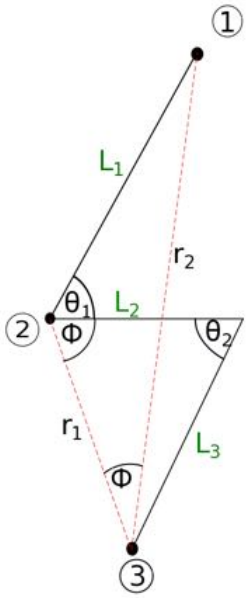


Figure 4.3.1: The placing of tracking markers on the human to be tracked whilst swinging. Marker 1 was placed on the side of the head above the ear. Marker 2 was placed on the side of the swing in front of the hip bone. Marker 3 was placed in the middle of the side of the shoe nearest the camera.

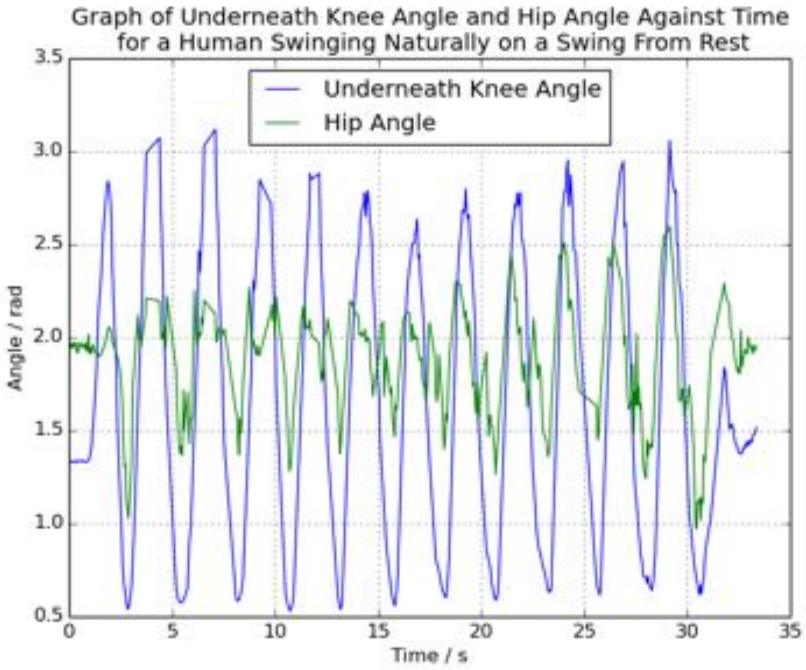


Figure 4.3.2: Relationship between the Underneath Knee angle and Hip Angle for a human swinging from rest on a rope swing

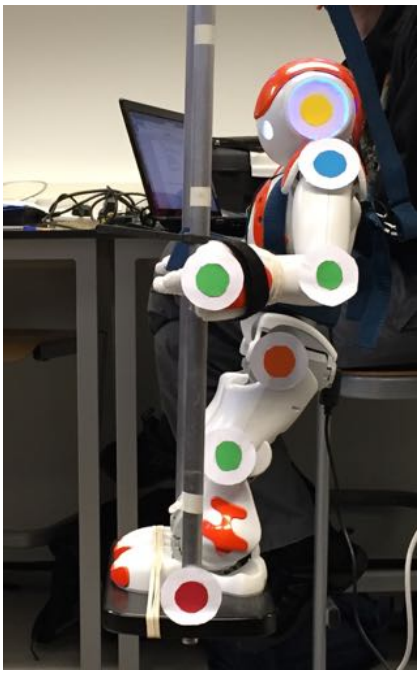


Figure 4.3.3: The placing of tracking markers on the NAO robot for capture and analysis by *Tracker (D. Brown)*

to interpret and the phase relationship between the two angles also appears to change. This was most likely due to the person on the swing attempting to 'brake' when swinging in order to not reach too great an amplitude.

Significant noise can be seen in the data as a result of the approximations made during analysis. As only three markers were available, in order to calculate θ_1 it was necessary to assume that the length L_2 (corresponding to the upper leg) remained parallel to the swing seat throughout the motion. In reality, this was not the case and the upper leg was observed to move over time.

The data capture of the human swing was an area that could be improved in future. Using more tracking marker points could significantly improve the data obtained. Using a swing with stiff metal rods instead of rope could also make the data easier to interpret. Lastly, taking into account the parallax error involved in tracking the markers was something which was overlooked due to time constraints but could significantly improve the data analysis. This is an area explained in more detail in Section 4.3.2.2.

4.3.2 Analysis of NAO Robot Swinging

4.3.2.1 Method

After processing data from a human on a swing as detailed in Section 4.3.1.2 and Section 4.3.1.3, the *Tracker* program was identified as having weaknesses . The raw data (the video file) was not recorded by the program and the output was a .txt file containing the positions of each of the tracked points alone, with no contextual video to inform further data processing. This made interpretation of data difficult. Secondly, the program was limited to tracking three marker points as alluded to in Section 4.3.1.3.

Tracker (D. Brown) was found to be a program which met the needs of the Group Study in terms of tracking the NAO robot on the swing. *Tracker (D. Brown)* was able to import a previously taken video and track certain 'unique' points in the frame. To make this more effective, tracking markers were created from paper and placed on the NAO robot as shown in Fig 4.3.3.

Video of the NAO robot swinging was recorded using an HD video camera on a phone. The phone was secured using a clamp stand at a height in line with the NAO robot's hip. The clamp stand was used as a rudimentary tripod in order to keep the camera fixed and stable to enable later tracking of the movement. A calibration scale (masking tape on the swing rod as seen in Fig 4.3.3) was created to enable pixel to length conversion within *Tracker (D. Brown)*.

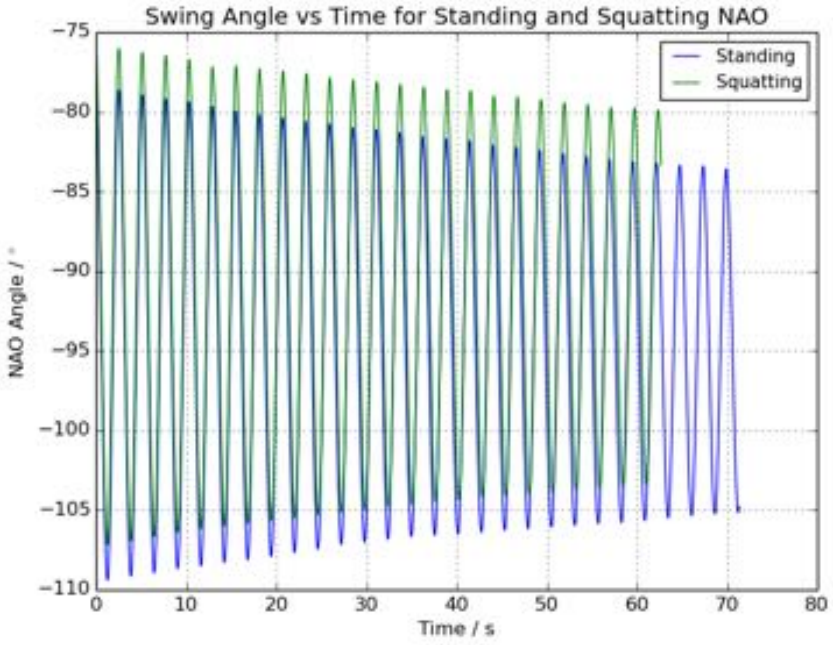


Figure 4.3.4: A comparison of the angle of the swing (obtained from *Tracker (D. Brown)*) for the NAO standing and squatting motions

Tracker (D. Brown) was used to import and analyse the video of the NAO robot. A simple user guide was created for the program for future cohort's familiarisation. This can be seen in Appendix A.1. The full process of tracking the markers can be seen in the user guide. For each marker, the x and y components of position, velocity and acceleration were calculated automatically as well as the angle and angular velocity of the marker with respect to the user defined swing joint (the origin). The calculated variables were exported to a .txt file for analysis. A MATLAB script was created to import the *Tracker (D. Brown)* data and assign the relevant columns of data to the relevant variables.Desired outputs were then calculated, exported to a .csv file and plotted.

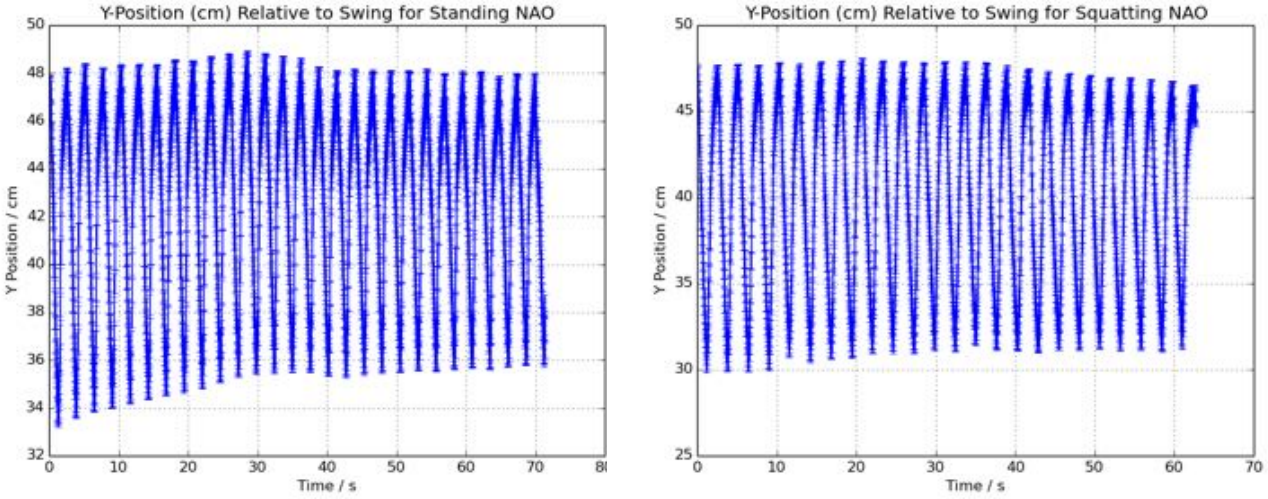
Video was taken for the NAO robot maintaining swinging from an initial height. NAO motion a programmed reaction to the swing angle which was fed back to the NAO robot from the rotary encoder. Further information on the encoder can be found in Section 3.1.4. Two standing motions were used. A motion where the robot remained standing and pivoted from the ankles (this was referred to as 'standing') and a motion in which the robot squats down and stands up periodically (this was referred to as 'squatting'). Separate video was recorded for both motions and analysed using *Tracker (D. Brown)*.

4.3.2.2 Results and discussion - Standing NAO Motions

The relevant potential outputs from the processed data were discussed with the Mathematics and Simulations sub-group leader and Dr. Theis. It was decided that the most relevant quantities would be the change in position of the centre of mass of the robot and the swing angle obtained throughout the two motions.

The swing angle over time for the standing and squatting motions can be seen in Fig 4.3.4. This data was calculated automatically by *Tracker (D. Brown)*. Fig 4.3.4 shows that the standing motion did not maintain the large initial amplitude of the swing but reached a more stable amplitude of approximately 20 degrees ± 2 degrees. The squatting motion appeared to give a slightly higher stable amplitude of 24 degrees ± 2 degrees. The error on the tracking of each marker, due to the programme evolving the tracking template image over time, appeared to be ±1cm in both the x and y directions. This corresponded to an angle error of ± 1 degree for the 2m swing, giving an angle amplitude error of ± 2 degrees.

The movement in the y-direction, relative to the swing, of the marker placed on the head of the NAO robot can be seen in Fig 4.3.5. This head movement was taken to represent the movement of the centre of mass of



(a) Y-position of the NAO head relative to the swing for the standing motion

(b) Y-position of the NAO head relative to the swing for the squatting motion

Figure 4.3.5: Comparison of the y-direction movement of the NAO head relative to the swing for the standing and squatting motions. The apparent 'noise' on the curves are the y-axis error bars

the NAO robot in the y-direction, as the torso and head of the robot remained fixed relative to each other throughout both motions. The standing motion gave an amplitude of 11cm ± 2cm (Fig 4.3.5a) and the squatting motion gave an amplitude of 14cm ± 2cm (Fig 4.3.5b). This appears to confirm that the greater amplitude of the swing angle during the squatting motion is due to the greater change in the height of the centre of mass of the NAO robot. This confirms the research and hypothesis from the Mathematics and Simulations sub-group. The vertical movement calculated from the video of the squatting motion appeared to be correct from visual observation, however, the standing motion appeared to give a much smaller vertical movement of the head than the calculated 11cm ± 2cm.

It can be seen in Fig 4.3.4 that the angle of the swing for the two motions are not both symmetrical about -90 degrees. This implied that the angle obtained from *Tracker (D. Brown)* possessed errors (other than the auto-tracking error) which had not been accounted for. To investigate this, the angle of the swing obtained from *Tracker (D. Brown)* was compared against the angle of the swing obtained from the encoder, which was known to be of a very high accuracy. This can be seen in Fig 4.3.6.

From this comparison, the calculated amplitude from *Tracker (D. Brown)* is clearly significantly larger than the actual amplitude of the swing. However, it appears that up to angles of just under 10 degrees, *Tracker (D. Brown)* is a suitable method for determining the angle of the swing.

The errors which caused discrepancies in the calculated swing angle and the y-position of the head marker were most likely due to parallax. For large amplitudes of the swing motion, the x position of the NAO robot markers relative to the swing were significantly altered due to this. To determine the swing point, a calibration scale was created using the tape markers in Fig 4.3.3 and the origin was placed in accordance with this. This swing point was incorrectly calibrated as parallax in the y-direction was not taken into account. This caused the calculated swing angle to be too large and also the y-direction movement of the head for the standing motion to be too large. In the future, parallax could be accounted for in both the x-direction and the y-direction to significantly improve the accuracy of using the *Tracker (D. Brown)* software. The camera used for data capture could be set-up to be exactly in alignment with the swing joint and to be facing exactly perpendicular to the swing motion. Together with accounting for parallax, this should adjust the origin so that the swing angle is calculated as symmetric about -90 degrees.

——— The following was contributed by: Joe Allen ———

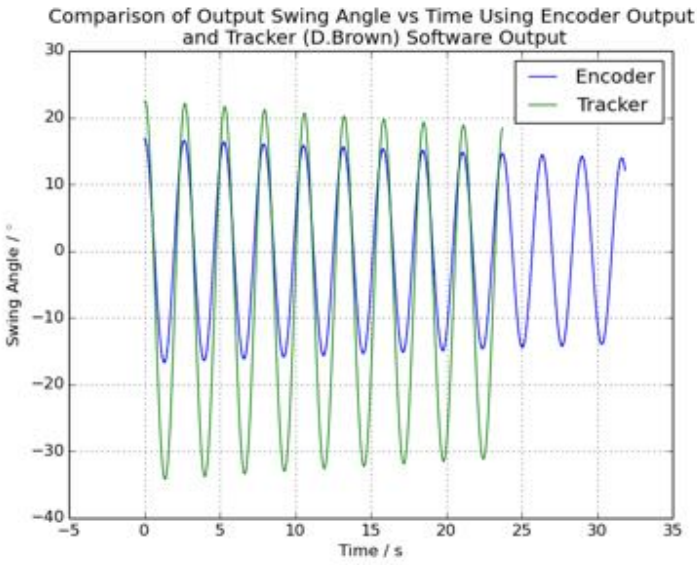


Figure 4.3.6: Comparison of the swing angle obtained from the encoder and from *Tracker (D. Brown)*

4.4 Swing Analysis

4.4.1 Initial Considerations

The encoder's precision, discussed in Section 3.1.4.2, made it ideal for extracting information about the specific swings used in the laboratory. Several parts of the project used this information. The Webots simulations needed to be designed to be as close to the real system as possible, and this required an accurate damping coefficient to be used in their model. In addition, a value calculated for the time period of the system was used in the final program with which the robot (Nao) drove the swing. It was planned to try and ascertain how the damping and the time period of the system changed depending on amplitude. The code used for this analysis is included in the digital appendix.

Data recorded by the encoder was very clean, with little random noise, however there were some noticeable systematic errors, described in Section 3.1.4.5, whose origin needed to be worked out in order to negate their affects. There was an additional unexplained error discovered late on during this analysis which sometimes occurred as the swing passed through an angular displacement of approximately $-52°$ or $+38°$, at which point the gradient changes sign for the subsequent 5 - 10 data points, after which it reverts back to recording the actual displacement of the swing. It was hypothesized there may be some imperfections internally in the encoder, as these errors were not found when a second encoder was tested, as these which would cause this systematic error. Given that the encoder angle was normally recorded every 7 milliseconds, there is a sufficient amount of data points to all but nullify the affect of this error on any ensuing analysis, also because they were not always present so re-recording the data set usually got rid of these errors.

After some initial tests and basic analysis, it was discovered that the period of oscillation of the straight-rod swing, shown in Figure 3.1.2, was approximately 2.5 seconds. Moreover, when extra weights were added to the swing, to assimilate the mass of the robot, the swing support, indicated by **C** in Figure 3.1.2, flexed. This meant that the swing was not moving in a constant vertical plane and that the damping due to friction in the pivots was increased. To minimise this flexing, the top support was tied firmly to the metal bars of the balcony.

In addition, if the seat was not secured carefully and correctly, the swing's rods would not be perfectly parallel. This also had an adverse affect on the system, as the top joints were no longer in correct alignment, causing the friction in the pivots to increase, which warped resulting data by increasing the damping in the system.

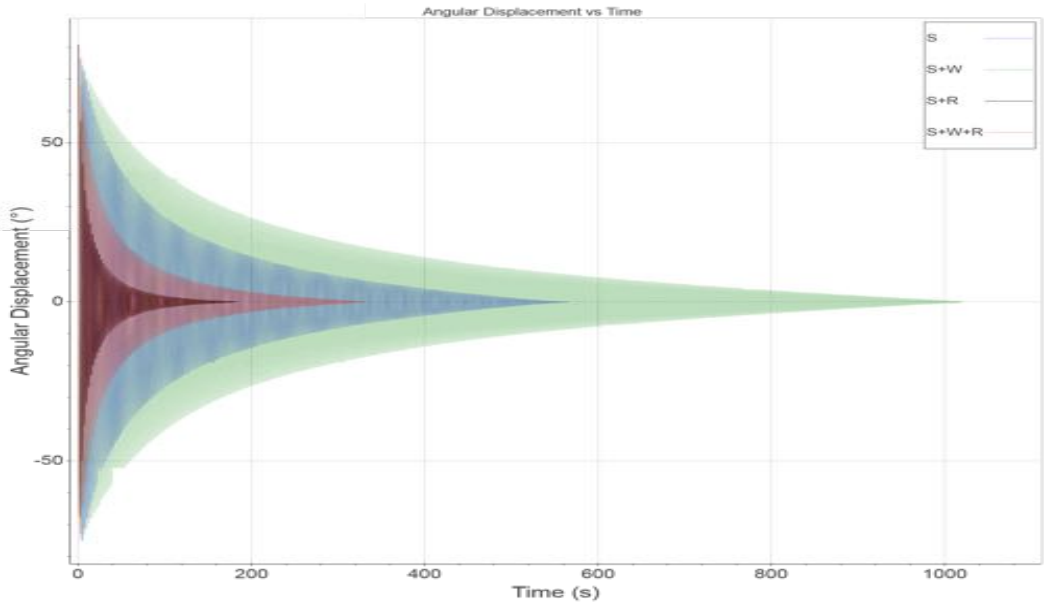


Figure 4.4.1: The oscillations recorded for the four different swing set ups, showing, at a high level, the affect of air resistance and added mass to the swing system. Understanding the key: S denotes the straight-rod swing, W denotes added weight and R denotes added air resistance.

4.4.2 Analysis

With the swing undergoing damped harmonic motion, it was expected that a function of the form in Equation 4.4.1 would describe the oscillatory system, where y is the angular displacement, t is time, γ is the damping coefficient, ω is the angular velocity of the swing and A and ϕ are necessary constants. By approximating the encoder data to this function using a curve fitting routine, values of γ and the time period T, $T = 2\pi/\omega$, of the system could be extracted. The curve fitting in question is included in *Scipy* for Python, and uses the Levenberg-Marquardt method. [12]

$$y = Ae^{-\gamma t}cos(\omega t - \phi) \tag{4.4.1}$$

However, due to the physical nature of the swing, the time period and damping coefficient change over the course of its motion. Originally the time period had been calculated from the Fourier transform and power spectrum of the oscillatory data, but this method was deemed insufficient as it could not show the variation of time period with amplitude. The curve fitting method was able to circumvent this issue, as it could be reliably performed on a small chunk of the data. Then, for each chunk, γ and the time period could be calculated. The length of the chunk was normally 10 seconds, because each one would contain over three complete oscillations, allowing an accurate fit.

In order to investigate the properties of the swing, experiments were carried out using four different set ups, with different combinations of weight and air resistance to see how these factors affected the motion of the system. Figure 4.4.1 shows the oscillations of the swing in these four experiments. All the experiments were started at an angular displacement above 77° and data was recorded until the system reached rest. The weight added was 5.27 kg, approximately the mass of the robot: 5.18 kg, and the added air resistance was due to a 53×31 cm cover, compared to the robot's height of 57.3 cm. No doubt as the swing poles are only separated by 30 cm the rectangle provided greater air resistance than the robot, however in a comparison test the robot was only in motion for $\frac{1}{3}$ longer so the two configurations do not have wildly differing effects.

The data was taken from the straight-rod swing, with added weight and air resistance, set up (SWR) and the damped oscillation function fitted to it in approximately ten second sections. The fits were all very good, with an average reduced χ^2 value of $\chi^2 = 1.09514$ (d.f.= 1429). The time period of each chunk was calculated from the angular velocity and plotted against the average maximum amplitude of the chunk, displayed in Figure 4.4.2. Because the fits to obtain this data were good, the error bars of the points were mostly too small to be seen on this graph, and so the line of best fit does not pass through all the points' error bars. Possible reasons for this

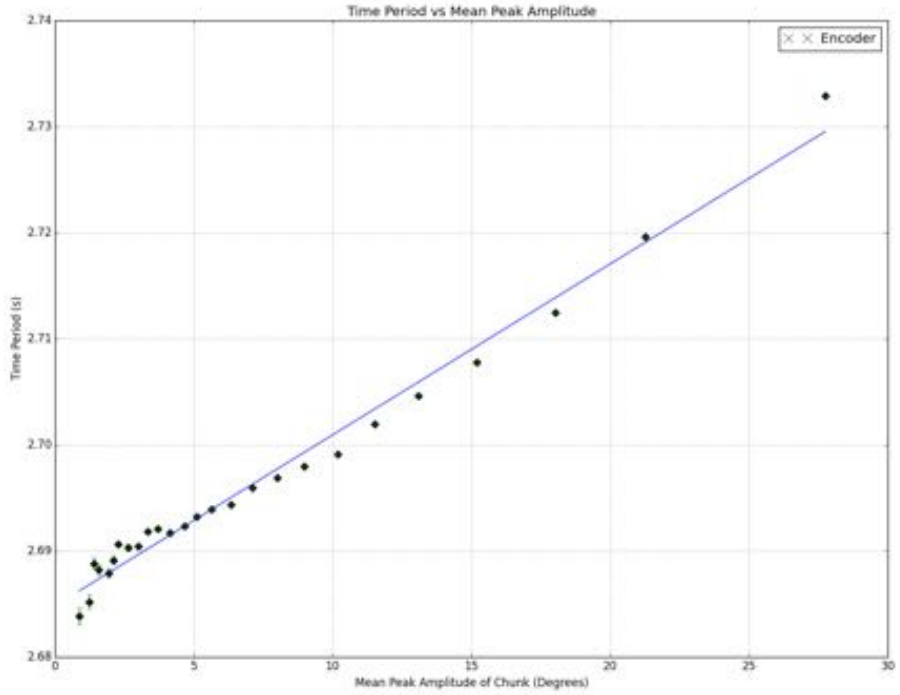


Figure 4.4.2: Time Period versus Mean Maximum Amplitude for the straight-rod swing set up with additional weight and air resistance.

will be discussed later.

If the chosen linear function for the data in Figure 4.4.2 is correct then the time period for the SWR set up varies with average amplitude as in Equation 4.4.2, where $\bar{A}$ is the average amplitude of a data chunk. Air resistance is the main factor that causes time period to vary with amplitude, because the swing will move with a higher velocity overall when oscillating at higher amplitudes. This increase in velocity increases the air resistance acting on the swing, thus slowing it down slightly and increasing the time period.

$$T = (1.6117 \pm 0.0424) \times 10^{-3}\bar{A} + 2.6848 \pm 0.0004 \tag{4.4.2}$$

The next experiment involved using the robot stationary on the standing swing, recording data from both the encoder and the robot's internal gyroscope contained in its Inertial Unit [4], letting the system oscillate from above 25° until rest. One of the main project aims was to get the robot to swing using only its own sensors, so it needed to be ascertained how accurate it was and whether it was fit for purpose. It is not clear on what scale the gyroscope outputs data, but with the introduction of a linear multiplying factor it was clear that what it records is equivalent to the encoder. The fits to the encoder data produced a reduced χ^2 value of $\chi^2 = 0.84676$ (d.f. $= 1429$), and the equivalent curve fits to the gyroscope data resulted in a goodness of fit of $\chi^2 = 50.8976$ (d.f. $= 1429$. The data from the gyroscope was harder to fit because there were often discontinuities in the data, however the general profile of the damped oscillation was obvious enough to be fitted reasonably well.

After the same process was carried out on the encoder and gyroscope data from this experiment, time period was again plotted against the mean amplitude of the chunk for both the data sets. Again linear trends were observed, the time period recorded by the gyroscope varying as in Equation 4.4.3, where $\bar{A}$ is the mean amplitude of a section of data.

$$T = (1.3531 \pm 0.0500) \times 10^{-3}\bar{A} + 2.5710 \pm 0.0005 \tag{4.4.3}$$

The encoder data from the same experiment gave the relation in Equation 4.4.4, showing that the robot's gyroscope is in fact accurate enough to produce results which are within error bounds to those of the encoder.

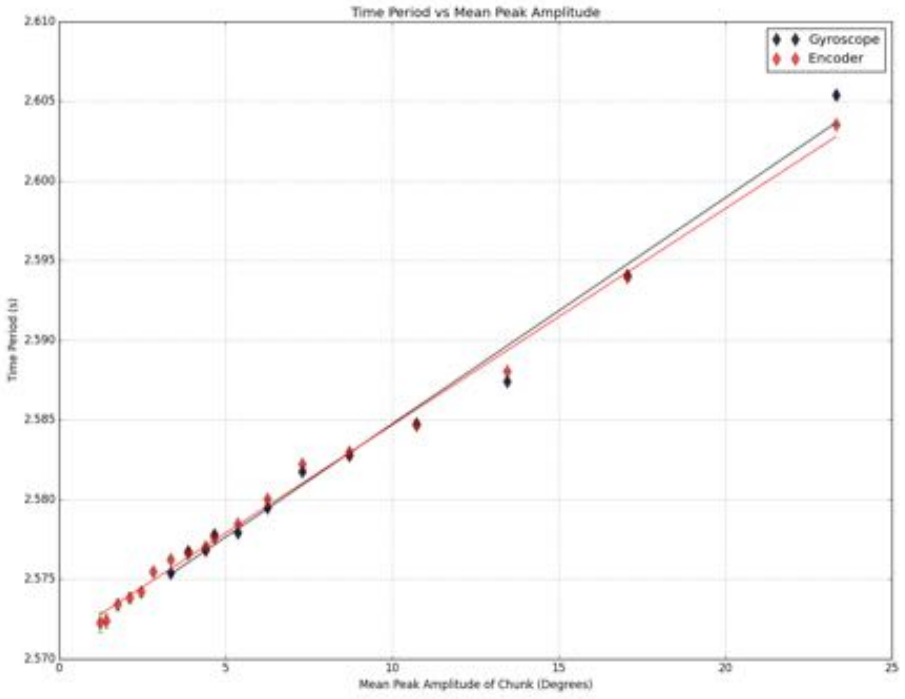


Figure 4.4.3: Time Period versus Mean Maximum Amplitude for the encoder (red) and gyroscope (black) data recorded with the robot standing unmoving on the swing. The encoder data fit has a χ-squared value of $\chi^2 = 1113.95$. The gyroscope data fit has a goodness of fit of $\chi^2 = 32.38$.

The chi-squared values are rather large for both the fits in Figure 4.4.3, there are two potential reasons for this: either the model fitted to the data is wrong, or the error bars of the data points are too small. The relations in Equation 4.4.3 and Equation 4.4.4 are certainly of the same order as Equation 4.4.2, however the noticeable differences in the results can only be due to the differing air resistances in the two experiments.

If the robot is assumed to be a rectangle of height 57.3 cm and width 27.5 cm, the width of its shoulders, it has approximately 75 cm^2 less surface area to provide air resistance, in reality the difference is larger. It is this difference that must cause the change in how time period varies with mean amplitude, increasing the low angular displacement time period, because the increased air resistance further reduces the velocity of the swing which increases the period, and thus shifting the data points systematically upward. On top of this, the increased air resistance also has a greater affect at larger velocities, which occur at larger amplitudes, which makes the line fitted to the SWR data have a greater gradient.

$$T = (1.3889 \pm 0.0282) \times 10^{-3} \bar{A} + 2.5703 \pm 0.0004 \tag{4.4.4}$$

These experiments were also used to determine the low angle time period of the swing, 2.5703 ± 0.0004 seconds, which was used to allow Nao to start swinging from rest, as at very low amplitudes it is not possible to use the encoder or gyroscope readings to work out when Nao should perform a forward or backward motion.

4.4.3 Conclusion

The reason that few of the time period versus mean amplitude points encompass the line of best fit with their error bars is thought to be because of two reasons. Firstly, the initial curve fits on the damped harmonic data chunks were very good and so the errors on the data points were rather small, which makes it harder to fit a second curve to the calculated points.

Secondly, the amalgamation of data into approximately 10 second chunks has a large affect, because time period is plotted against the average of the maximum amplitudes in the chunk, half the sum of the largest and smallest peak amplitudes. This means that there is potentially quite a large error in each of the points' x-axis position, for example if a given chunk began just past a maximum peak such that such that the first maximum only appears almost a whole period into the chunk, then this will skew the point. For points at higher amplitudes this error will be augmented because over a set time the percentage change in amplitude is greater than at lower amplitudes, therefore there is a greater uncertainty in a point's x-position.

A possible way to get around these problems would be to analyse the damped harmonic motion period by period, rather than in an arbitrary chunk containing several oscillations, as this would achieve a much more accurate representation of how time period varies with amplitude. Partly because the change in time period over a single oscillation could certainly be neglected, and partly due to the fact that there would be no error associated with the value of the initial amplitude for the point. If this is an unrealistic solution, then a method to calculate the horizontal errors, in average peak amplitude, for the data in Figures 4.4.2 and 4.4.3 would be implemented to get a better idea of the accuracy of the fit.

With hindsight, if the analysis were done again, much more data would be collected with the robot on the swing and far less time spent devising and analysing the other set ups with added weight and/or air resistance. Perhaps a method to analyse the gyroscope data, when the robot is not stationary on the swing but is performing motions to create movement, could be used, as if the robot is ever to swing using only its own sensors it seems likely that the gyroscope will be the most useful for this purpose.

——— The following was contributed by: Michael Wright ———

4.5 Results from Investigations on Webots

4.5.1 Virtual Environment: Investigations

Once the environment had been set up, complete with; the swing, it's friction, Nao positioned and written controllers, it was possible to start running some simulations and collecting data.

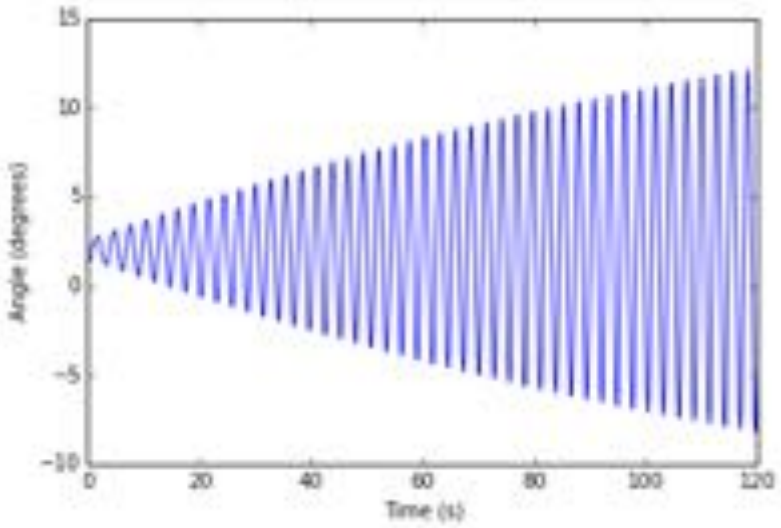


Figure 4.5.1: Standing Swing from Rest over 100s

4.5.1.1 Initial Investigations

The first virtual investigation is not unlike one run by the robot programmers on the actual Nao robot. From watching videos provided by the Human Swing subgroup, it was decided that each swing iteration should have a forward and a backward motion. The video they provided demonstrated that these motions were roughly used at the apex of each swing. This was used to produce our first simulations.

Initially the basic standing swing was used, and the aim was to record the maximum amplitude reached by the swing. Nao would repeat the 2 motions at the corresponding apexes. The first simulations required continuous modifying of the environment so that Nao could continue to achieve higher and higher amplitudes without falling off the swing. As Nao swung higher, the larger forces experienced could cause its hands or feet lose grip with the swing. Eventually this issue was resolved using the methods described earlier and the true maximum amplitude of the system could be determined.

Figure 4.5.2 shows the first 100 seconds of the investigation. The increase in amplitude slows over time, as it becomes harder and harder to add more energy to the system. However Nao eventually achieves angles as large as 18.9° after 10 minutes, or roughly 0.1m in height.

It can be seen that that the swing doesn't start or oscillate about 0. This is because the rest position of Nao on the swing isn't perfectly flat. In order for Nao to hold the swing, it must stand outside the rods, moving the centre of mass slightly to one side.

A similar investigation was performed using the sitting cage swing. Using a slightly different set of motion files, and a combination of 2 gyro units (1 per pivot). The sitting swing was able to achieve greater amplitudes, and considerably quicker when compared to the virtual standing swing. Over 10 minutes the standing swing achieved a max height of 0.102m, whereas the sitting swing achieved this in 80s, and a max height of 0.198m in 120s.

The system as a whole was much more unstable however. The lower and upper parts of the pendulum would oscillate in phase but at very different magnitudes. The upper part of the pendulum would oscillate to between small values, but the lower part of the swing tilted as far as 49.3°. Nao did not achieve a true maximum amplitude, because he quickly became unstable and fell off the swing at such steep angles.

A similar non-zero effect can seen in figure 4.5.2. The forward swing amplitude is generally a lot higher than the backswing (about 30% larger). Similar to before, this is because Nao's legs and arms move the centre of mass of the system forward.

One last investigation before moving onto more advanced techniques, was seeing how the mass of the swing might affect the maximum amplitude achieved by Nao. This was suspected to be the case, since the friction relation in the previous section has a mass dependency, but factors such as the moment of inertia are also expected to change. Though only 5 simulations were run, it is quite evident that that a correlation exists.

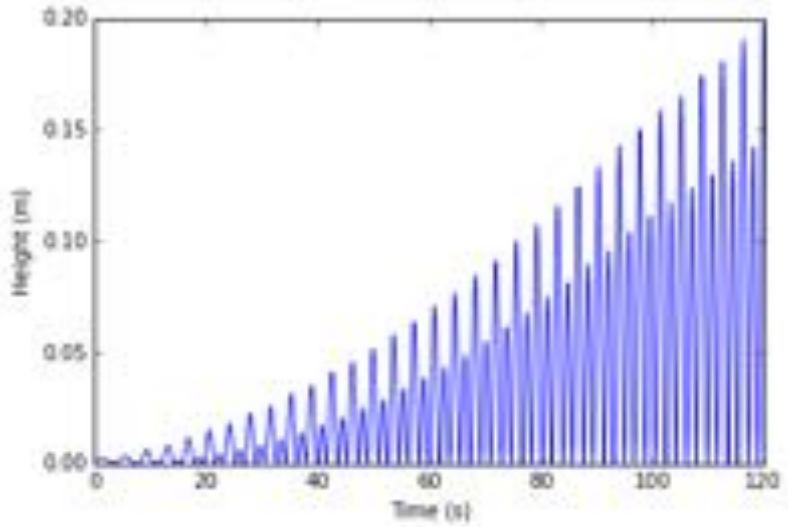


Figure 4.5.2: Virtual Sitting Swing from Rest over 120s

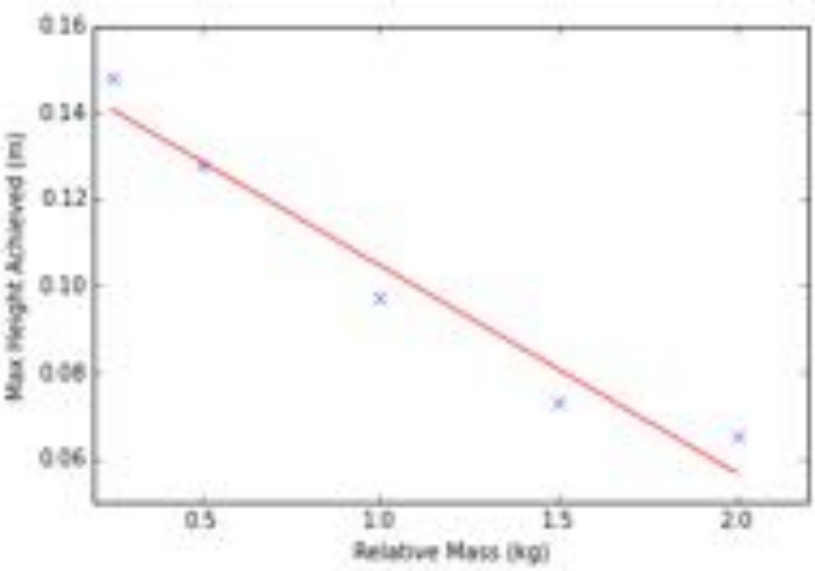


Figure 4.5.3: Adjusting the swing's relative mass (not Nao's) changed the max amplitude acheived by Nao. Linear fit is shown, but it is likely to be a different, complex relation.

Reducing the mass by a factor half, roughly increases the max amplitude reached by 0.02m. More data points would need to be taken for a more conclusive result, but since this was not directly helpful to this year's project team, collecting more data for this study was not prioritised.

——————— The following was contributed by: Fred Grover ———————

4.5.1.2 Mapping parameters

To see what the optimal setup for the robot swinging would be, three parameters would be adjusted independently and the maximal amplitude achieved recorded. Ultimately this would allow the best set of values for the parameters to be used giving the best amplitude from the robot swinging. The three parameters which were decided on were the phase in which the forward motion would begin, the phase in which the backward motion would begin and the rate at which the motion would be performed. The phase parameter dictated at what part into the period of the swing the robot would begin its motion. To make the adjustments of the parameters and run the simulation would be too inconvenient and time consuming to manually perform, therefore a script would have to be constructed which would automate it. To do this method the windows batch command scripts would be used. The script would be required to sequentially increase the input variables and initialise Webots which would then read in and then output the amplitude achieved. The script would then be required to terminate the Webots simulator, before looping around again.

To perform this three batch scripts would be required; the master script would be involved simply with incrementing the parameter to be adjusted, writing the value to the corresponding input file and the spawning of the other two scripts which would run in parallel. The second script was uniquely in charge of starting the Webots executable as a child process, the relation between the script and the sub-sequential Webots application were co-dependent, this was essential for causing the script to close when Webots was terminated. The third batch file was in charge of terminating the Webots program once the required time had elapsed. Once both scripts had come to a conclusion, the master script would then begin in the next step of the loop repeating the whole process. This method allowed the parameter optimisation to occur autonomously which was important considering the time required to run the large number of simulations expected. Two outputs were recorded by the simulations to check two different points of information about the set-up, the time taken to achieve 15 degrees and the maximum amplitude achieved. This meant it would be possible to see if certain parameters were better as getting the swing up to a certain amplitude at a faster rate or if some were more efficient at achieving a higher amplitude.

Motion Speed Parameter The initial script involved would modify the time elapsed for the second part of each motion files, this would change the speed in which the two swinging motions were performed. For this initial set up the robot would begin it's motions at the apex of each swing. The parameter was adjusted between 100 and 600 ms in increments of 50 ms. This was a compromise between run speed and accuracy of results, lower increments would give more detailed results but at the cost of the run time of the optimisations. The robots minimum motion duration was found to be just over 0.365 seconds. The range in which the data was analysed was chosen so the minimum duration of the motion was slightly lower then the real robot was able to achieve with the maximum being much larger; this was important as it allowed the determining of what motion duration would be optimum for the robot perform to achieve this highest amplitude Figure 4.5.4 shows the relation between the length of the Robot's motion and the amplitude achieved.

The data seen in figure 4.5.4 shows a linear relationship; as the rate in which it takes to perform the swinging motion is decreased the maximum amplitude achieved by the swing is increased. The amplitude achieved at 325 ms was found to be 0.139 meters and the optimum duration for the motion. Though being a simulation the accuracy of each data point is only as reliable as the power of the software and errors are difficult to ascertain the data showed a clear relation between the parameter and the amplitude. This meant that in all future simulations and real world experiments the robot would be programmed to attempt the motion at the fastest rate at which it could achieve as this was shown in the simulation to consistently achieve the largest amplitude.

Phase Parameter The next parameter which required adjusting was the phase at which the robot began it swing motion; this was done by changing at which part of the swing the robot was told to start the motion file. To do this several assumptions had to be made, the first was the fact that the current period didn't change dramatically from the previous period. The second assumption was that the only delay in beginning the motion

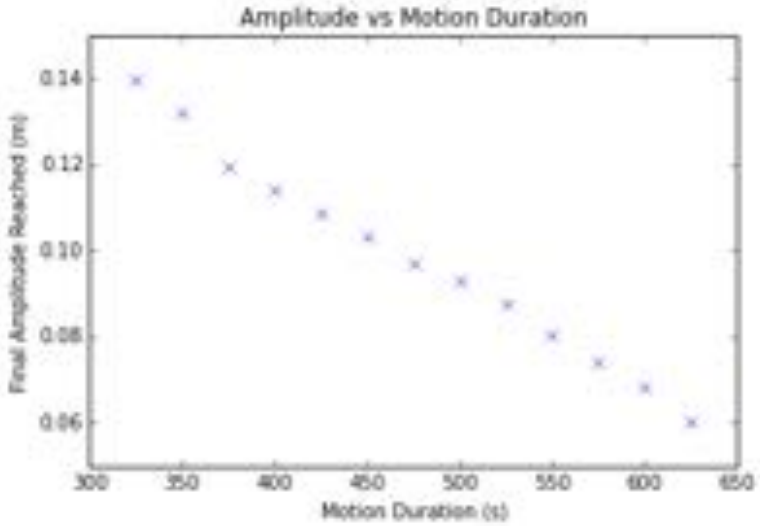


Figure 4.5.4: amplitude against Motion duration

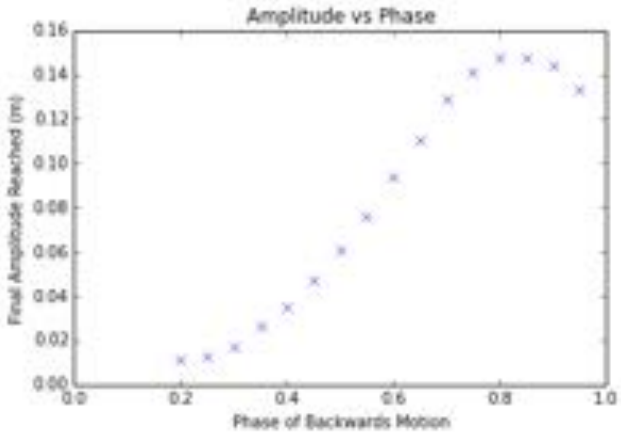
was due to the known time lapse due to the emitter and receiver communication lag. These assumptions were acceptable as previous simulated runs of the swing had shown a roughly consistent period with little lag in the beginning of the motions.

It was possible to determine how far into the current period by recording the time at which the swing was at the apex and using this as the initial time and deducting this from the current point in time. By comparing the ratio of this value to the previous period and including the delay due to emitter-receiver communication it was possible to change at what fraction of the period the robot's motion were begun. The phase at which each swinging motion begins would be modified independently by the batch script and the maximum amplitude achieved recorded. Initially the two phases were run independently, this involved running the script where one phase was changed between 0.2 and 0.95 in gaps of 0.05 while the other phase was held constant. This gave a rough estimate of which region the optimum phase for the both motions were. This set-up assumed independence of the phases which would be checked later.

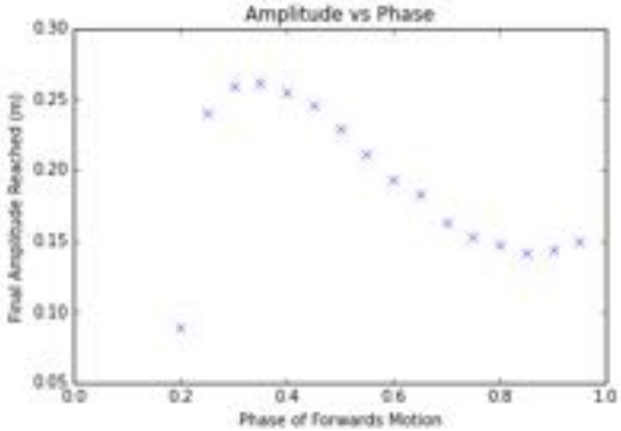
The graph 4.5.5a shows the changing of the backward phase when the forward phase was maintained at 0.9. The data shows a peak at the region of 0.85, that it is to say that beginning the motion 85% into the swing's period appeared to give the highest possible achievable amplitude of the swing. By measuring the period from one apex to the other to be consistently 1.376 seconds on the simulation at the terminal amplitudes it was possible to determine that for this motion speed it was best to perform the backward swing 1.17 seconds into the loop or 0.20 seconds before the end of the period. As the motion's duration is 0.366 seconds it was noticed that there appeared to be a rough relation between the motions length and the point at which the robot should begin its motion; the robot should begin its movement so that half of the motion occurs after the apex has been reached by the swing.

The second graph shows the changing of the forward phase when the backward phase was held constant at 0.9. The peaks appears in the region of 0.35, or 35% into the period. By taking into account the time period of the motion it states that it is best to begin swinging 0.48 seconds into the motion or 0.21 seconds before the middle point in the swings motion. This shows the same similarity to the backward phase but in relation to the centre point. It showed that the best point to begin the forward motion was so that the middle point of the motion occurred at halfway into the swing's period. To test if this was true this set up was repeated for other measurements of the motion's duration. The repeating of the data for a motion duration of 0.5 seconds showed a similar relationship.

To find a more refined value for the two phases the set-up would be repeated concentrated around the previously found values with smaller gaps between each measurement. To analyse whether the two phases may be co-dependent on each other they would be compared by the use a 3d surface plot. The forward phase was defined to the limits of 0.2 to 0.5 and the backward phase between the limits of 0.65 to 0.95 each with gaps of 0.025. The surface plot for this parameter optimisation is shown in figure 4.5.6.



(a) Amplitude Against Backward Phase



(b) Amplitude Against Forward Phase

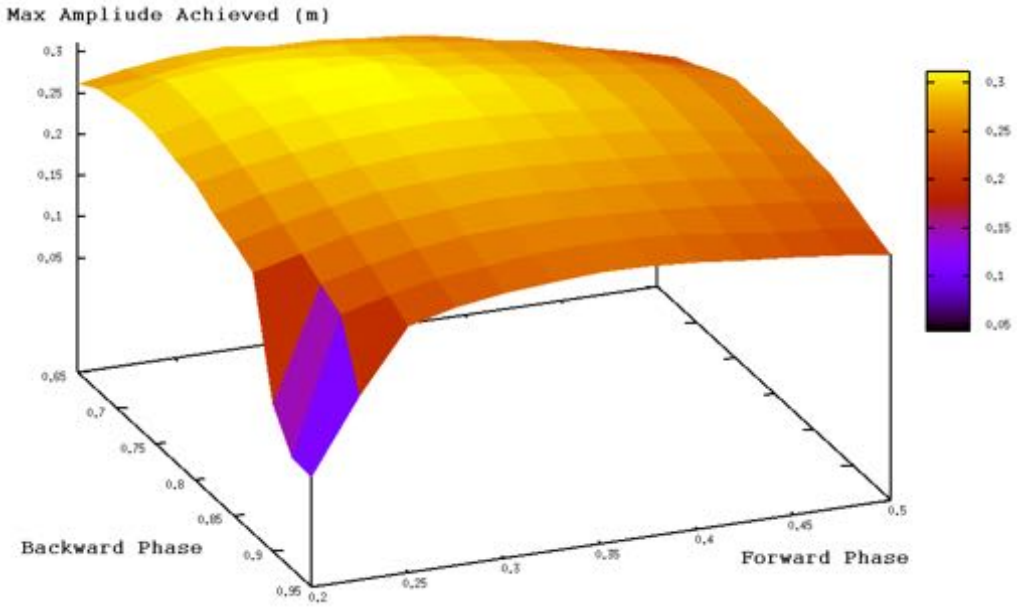


Figure 4.5.6: Amplitude Against Motion's Phases

The plot of figure 4.5.6 shows the amplitude achieved for different values of the two parameters. The figure shows there to be a generalised area where the highest amplitudes are achieved, as the parameters move away from these optimised values the height achieved is seen to dramatically drop. The plot of The two values of the parameters determined to give the highest amplitude achieved by the swing were 0.725 for the backward phase and 0.3 for the forward phase. The values differed from the previously calculated optimum parameter levels showing some level of co-dependence between the two phases, further analysis however would be needed to demonstrate this exact relation. For the forward swing the value of 0.725 meant that the robot would be required to swing 0.3784 seconds before the swing reached its apex. This value coincides with the duration of the motion. The simulation thereby states that the robot should begin its motion so that the motion is finished as the apex has been reached.

The different between this calculated value and the previous one indicated there are several factors which require balancing when determining what is the best point to begin swinging. At the lower amplitudes achieved when the forward phase was maintained at a non-optimum value it was more important to begin the swinging motion so that the main part of the motion occurred just after the swing was at its apex thereby applying the most torque to the system. However at the larger amplitudes achieved when the forward phase was at an improved value the best point to swing was more dependent on swinging so that the centre of mass was extended horizontally to the full amount at the swing's apex. For the forward phase the value which gave the highest amplitude was seen to be 0.3 the way into the period. This value was lower than the previous value determined when the backward phase was held constant also indicating that forward phase is dependent on several conflicting factors. If the robot began its motion much earlier than this then the robot would provide too much torque in the opposite direction to the motion; reducing energy gain of the system. However beginning the motion at a later point would mean that the horizontal position of the centre of mass of the robot relative to the swing would be reduced too late and therefore the motion of inertia would not be converted into angular velocity to the highest extent. These simulations indicated what the optimised values to achieve the highest swing amplitude were, though further parameters could be adjusted to see how these factor into the height achieved by the system.

——————— The following was contributed by: Michael Wright ———————

4.5.2 Conclusion

Even though the virtual environment team did not manage to feed information directly to the robot programmers, the investigations were able to produce a number of findings which will be useful for those undertaking the same project in the future.

Firstly, it was found that the Webots program is indeed capable of simulating the robot and swing system to a reasonable degree. The robot-driven standing swing exhibited a similar 2.7s long period in both cases. However it was noticed that the amplitudes achieved by the virtual swing were often higher than those seen in real life. For example, on the standing swing, similar motions performed at the apexes produced a max amplitude of 9 deg in the real environment, but 20 deg in the virtual. This could be caused by underestimated friction values or "perfect" properties of the simulation. For example, the virtual swing is constrained to move in one axis, whereas the real swing can oscillate sideways slightly. It may be possible to recreate this in the virtual swing with a different type of 'joint' node. Additionally, the real robot had difficulty gripping the swing whereas the virtual maintained a static connection due to the connector nodes.

Initial investigations found that both standing and seated swinging should be possible. Our study found that the seated swing has a lot of potential, achieving high amplitudes after just a few cycles. However a key problem was noticed with it's stability; the upper pivot does not move much, whereas the lower pivot oscillates between large values. The steepness of the swinging causes Nao to easily fall off the seat before higher amplitudes can be achieved. Moving the second pivot higher or removing it entirely should fix this problem.

Another swing modification which should improve performance is a lower swing mass. This reduces the system's moment of inertia and the robot's applied torque will have a bigger impact on the angular acceleration: $\tau = I\alpha$, resulting in higher amplitudes. Currently the swing's rods are likely made from solid steel rod which, although sturdy, contribute to most of the swing's mass. It may be possible to redesign the swing to use a lighter material such as aluminium, or change the shape to hollow tubes.

Our final investigation looked at how the max amplitude achieved depended on how fast Nao's motions were performed, and at what point they started. It was found that the quicker the motion was performed, the greater

the amplitude achieved. The real Nao is constrained to a maximum move speed, the robot programmers were informed that this is what should be used. For the standing swing, the best times at which to begin the forward and backward motions are 0.300 and 0.725 of the full swing after the apex, respectively. Using these values with the controllers in the apex, an optimised maximum amplitude of 0.311m or 33 deg was acheived.

——————— The following was contributed by: Vincent Fisher ———————

4.6 Nao Swinging from Pre-Coded Motions

4.6.1 Theory and Approach

The idea behind this approach is to use motions which are pre-programmed and called from a remote machine within an application. The goal is to create a program which can be started when Nao is securely on the swing; from this state the robot should be able to start swinging, increasing the systems amplitude until an angle is reached which can be sustained. For this to work, a set of motions have to be designed which have the desired response and there has to be a way to know when Nao should perform these motions. To do this we make use of the Naoqi Python SDK, as seen in 3.3.3.4.

To design the motions that would be used, we modelled how a human swings and tried to recreate these movements in a way which was efficient for Nao to execute but also similar to their human counterparts. We created four sets of motions to accomplish this: pulling and pushing the swing, which rely solely on the arms, and squatting down and standing up which uses the legs to perform the squat coincided with the arms to perform a push or pull movement. Once these motions were created, the next thing we needed to know was when they should be performed. This was another piece of information discovered by modelling how a human swings; it was found the movements are initiated around the peaks of the swing and are completed by the time the system goes through the centre of masses rest point. From this we made the approximation that Nao should perform the relevant motion at each peak; when he reaches the forward peak this will be pulling back on the swing/standing up and when he reaches the backwards peak this will be pushing the swing/squatting down. From this information, the next relevant thing to do was to make a program which could detect when Nao reaches these peaks.

4.6.2 Detecting Maximum Amplitude

Peaks will correspond to when the maximum amplitude of the system is reached, so the angle from the centre of mass at rest is its most positive or most negative. When Nao is positioned on the swing, as he has to lean back to accommodate his frame, the position of the systems centre of mass at rest is offset slightly from the origin of just the swing at rest.

Nao harbours an inertial unit located in the torso with its own processor. This unit is made of two inertial sensors; a 3-axis accelerometer and a 3-axis gyroscope. Hence both of these sensors are able to measure their respective values in 3 dimensional components. The dimensions are labelled as follows:

- X-Component: facing outwards from his chest in the forward direction
- Y-Component: bisects the chest place horizontally, direction is from the right side of the body to the left
- Z-Component: the vertical direction from the feet through to the head

An accelerometer is an electrical device used to measure acceleration forces; it gives measurements with units of meters per second squared, however these are not strictly the acceleration values, at least in the traditional sense. This is because it can deduce measurements from such forces that may be static, like the continuous force of gravity. This plays an important role in how you interpret accelerometer values; it measures 'proper acceleration' instead of coordinate acceleration. Gyroscopes on the other hand are devices built to measure or maintain rotational motion. The gyro used within Nao is capable of measuring angular velocity to five percent precision. These measurements have units of radians per second. As well as these sets of values, there is also a third. Nao uses an algorithm to combine data from the accelerometer and gyroscope to compute two torso angles, x and y. Theoretically we should be able to use the right component from any of these 3 values to determine the point of maximum amplitude. For the accelerometer, this would be when maximum acceleration is

recorded. As mentioned earlier, this is proper acceleration. So even when the unit is stationary, the components value will be maximum when it is aligned with the gravitational field. So for the x-component, which is the component of interest, it will be most positive when facing up, and most negative when facing down. For the angle values, this will also be detecting its most positive or most negative values, but with the Y-component. For the gyroscope, as Nao reaches the peak of his swing, the angular velocity decreases to zero. When the swing reverses and goes back in the opposite direction the value becomes negative and decreases until it reaches a minimum (most negative point) at the origin. On the increase to the opposite peak it then increases until it reaches the peak where it will be zero. In short; every time it changes direction, the sign changes, so what we look for is when there is a change in sign of the gyroscopes Y-component values.

4.6.3 Swinging From a Pre-existing Motion

Now we have an idea how to detect maximum amplitude and the motions to use, the next logical step would be to attempt to get Nao swinging from a pre-existing motion. The goal here is to find out what angle could be sustained if released from a large angle (say more than ten degrees), and to also ensure that if released from a small angle (less than five degrees) he could then react in a way which would increase the amplitude. To do this we need to create a program that translates the principles of the last section into code that detects when a movement should take place and subsequently execute it. As mentioned, we should be able to use any of the inertial values to do this.

To check the values change as we expect them to, and see which one would be the best to use, it is a good idea to gather data. To do this Nao was set up on the swing; he was then pulled out to an angle around 15 degrees. The swing was let go, free to swing in an isolated fashion until the system came to rest. Whilst doing this, a simple program was running which collected data from the inertial sensors and wrote them to a file. The graphs below show the results of this investigation:

From these graphs it is clear to see that the gyroscope is by far the smoothest and most reliable set of data. The angles and accelerometer both follow the trend we would expect but some values become unpredictable with both seeming to experience double peaking at points or random spiking. This data would therefore not be suitable for our purpose as it could cause motions to be executed at points we do not desire. Hence we use the gyroscope to do the job at hand by writing a program that detects when two successive values change sign then execute a motion. This program is a python module labelled 'SwingInMotionGyro.py' and can be seen in the appendix, once ready, it was tested in the same way we gathered data, but with the addition of Nao speaking when he reached a peak. This worked well, so now movements were threaded into the program. Here we hit a problem. As the sensor is in his chest, when a movement is executed the gyroscope values spike, and trip the program into making successive undesired movements. This can be seen in 4.6.4, here I have plotted the gyroscope values measured over a period of successful swinging time when started from an angle of 20 degrees. There is spiking data-peaks as well as what we would expect to see in dark blue. I attempted to fix the problem by making the program wait until the swing had reversed direction and went back through its origin until it started detecting for peaks again. Whilst this improved the program, there were still some undesirable effects, such as peak detections being missed. This coupled with how the gyroscope values react when squatting, as seen in 4.6.5, I decided under the time constraints it would be better to use the encoder to control the program instead of trying to overcome this problem. referring back to 4.6.5 you can see when Nao swings using squatting motions the gyroscope values become unreliable. The data has a lot of spiking and ends up looking like noise, you can no longer see the damping we would expect, which made me feel the problem would take more time to fix then we could afford to give.

By replacing the job of the gyroscope with the encoder, a second edition of this program was made, called 'SwingInMotionEnc.py'. This program monitored the angle for every cycle and when it became its most positive or most negative, a motion would be executed. We tested the program by placing Nao on the swing, giving him a push, then starting the program. Once it had been debugged fully and deemed as functioning in the expected way, we started to test how well Nao could maintain his swing. We found that performing squats kept the swing at a slightly higher amplitude than arm motions alone, so this is what we focused on. Referring to 4.6.6 you can see that Nao was able to sustain an amplitude of 9 degrees on the positive side and -10 on the negative. If you then look at 4.6.7 you can see when starting from a smaller amplitude, he managed to increase this to 8/9 degrees. The aim from here is to achieve this height when starting from rest. Whilst testing this program, we found that it would function appropriately when started from an amplitude of 2 degrees or higher, this will be

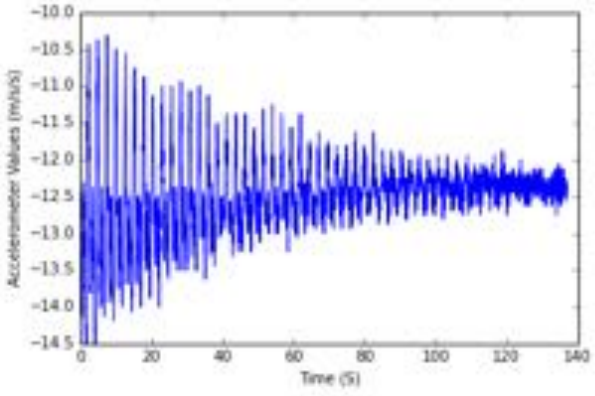


Figure 4.6.1: accelerometer values varying with time

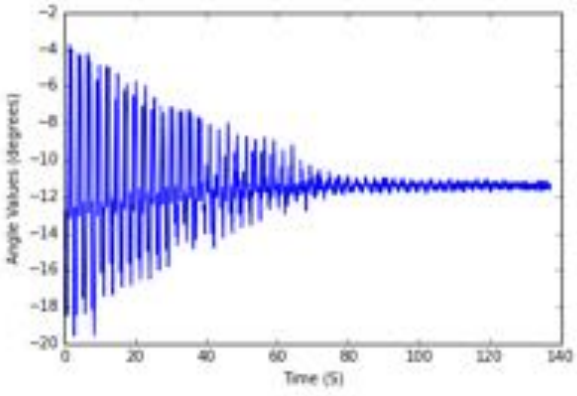


Figure 4.6.2: how the angles vary with time

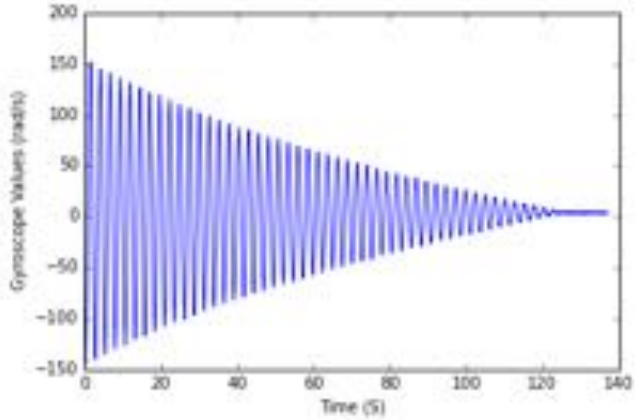


Figure 4.6.3: how the gyroscope values vary with time

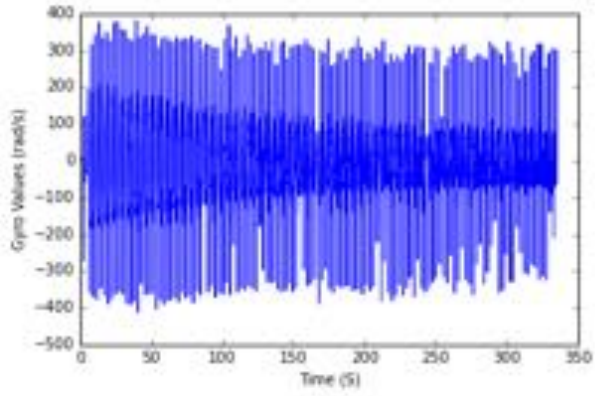


Figure 4.6.4: Shows gyroscope values when swinging with arm motions

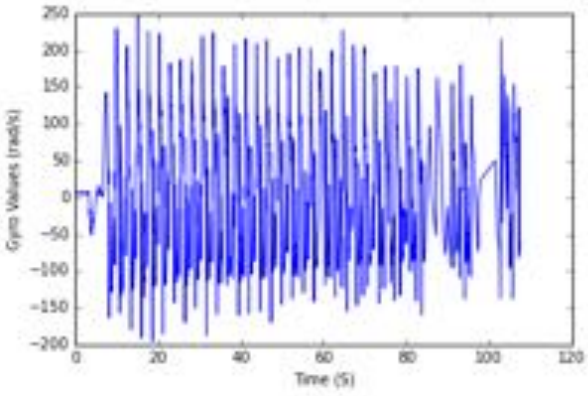


Figure 4.6.5: Shows gyroscope values when swinging with squat motions

an important consideration within the next section.

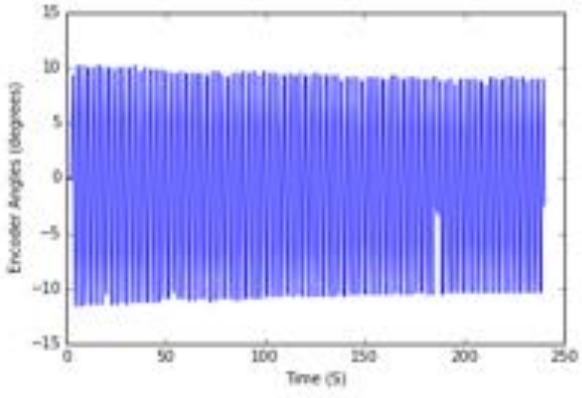


Figure 4.6.6: shows Nao sustaining an amplitude around 10 degrees

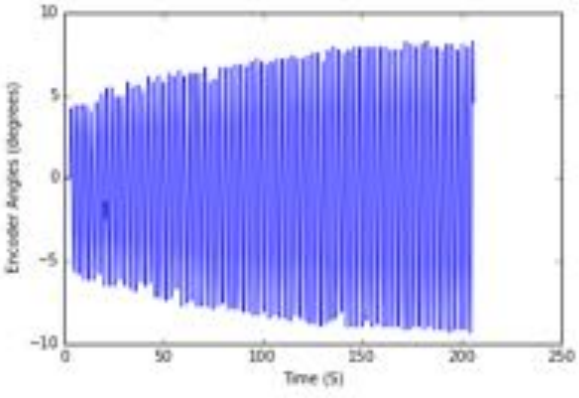


Figure 4.6.7: Shows Nao starting with a small amplitude which he increased over time

4.6.4 Starting From Rest

The final piece needed to demonstrate Nao has the potential of recreating a full swinging motion is for him to get the system moving coherently from its rest point. This was First tested using a simple program which allowed me to manually control his motions. This can be seen below:

```
importNao.Py
robot = remoteNao(''IP",PORT)
    while(True):
        choice=int(raw_input())
    ifchoice=1:
        robot.pushSwing(armSpeed,ElbowSpeed,neckSpeed)
        ifchoice=2:
        robot.pullSwing(armSpeed,ElbowSpeed,neckSpeed)
    else:
        break
```

When this program is running, pressing the 1 key followed by enter will execute the push motion, similarly pressing 2 will execute pull. Via timing these motions by eye it was possible to get Nao up to an amplitude of three degrees. We stopped here as from this angle I was satisfied the swing in motion program would be able to take over. This manual experiment was conducted as a proof of concept, so that we knew it was possible and gave us a clue into the timing needed to translate the process into code. As the motions are executed at the peaks of each cycle, it was decided to start from rest we needed to find out a rough period and tune the motions to be called at half the time interval of this period. When starting from rest we only use arm motions instead of squatting as the system responds better to the forward then back motion the arm movements create rather than the squats which involve up and down motion as well.

In section 4.5.2, you can see the period for the swing with Nao on it was found to be 2.57s. So following this logic, the sleep interval to test is 1.28 seconds. A small program was created which calls the appropriate motions with a sleep delay between them. Testing with 1.28 seconds did not yield successful results; the motions were too far apart and counteracted each other, causing the systems amplitude to vary with small magnitude around the origin but not actually increase. From here some trial and error took place; the sleep delay was decreased by small values and tested again, the process repeated until a value was found which had the desired outcome. This value ended up being 1.15 seconds; using this Nao successfully reached an amplitude of more than 2 degrees which is all we need from this process.

4.6.5 The Final Program

Now that we have a way for the robot to swing when in motion, using arm movements or squats, and we have a means to start the robot from rest, the final step is to piece all of this information together into one program which can be started once Nao is secured to the swing. To do this a piece of coding logic had to be designed that changes what Nao does dependent on the state of the system. The idea is to start from rest using the delay timer previously talked about, then, once the system has reached an amplitude of two degrees; we change over to using the encoder the same way we do in the SwingInMotion module; to dictate movements at peaks. Initially however we do this only using arm motions, as the amplitude is still too small for the squatting motions to be their most effective. Once the system has reached an amplitude of four degrees, we transition to the final part of the program which now executes squat movements instead of just arms. The final program to do this is a Python module named 'SwingFromRest.py' and can be seen in the appendix. Below is a sample of the logic used to make the appropriate decisions:

```
maxAngle = 0
    while (True):
        angleHolder = get encoder value

        if (abs(angleHolder) > maxAngle):
            maxAngle = abs(angleHolder)

        if(maxAngle >= 4):
            squatSwitch = True

        if (maxAngle < 2):
            #Execute push or pull motion
            #Sleep for requested time

        elif(maxAngle>=2):
            #use comparisons between successive angles
            #Execute push/pull if squatSwitch is false
            #Execute squat/stand if squatSwitch is True

            ang1 = ang2
```

This is just to show how the logic from the program works, much of it has been stripped out and replace with psuedocode. The important thing to realise is its the maximum angle that the system has recorded which dictates how Nao reacts as previously mentioned. in the full program, we also use multiprocessing so the program can execute concurrently instead of sequentially. This is as at less than 2 degrees we use sleep commands which block all parts of a sequential program. This becomes a problem when you want to do something like write data continuously to a file. Referring to the appendix you can see I use two different functions concurrently to do just this, one method gets the encoder data and continuously writes it to a file, whilst the second is responsible for controlling Nao. The result for running this program can be seen in 4.6.8. Nao managed to reach a height of just under eight degrees when on the swing for a time of fifteen minutes. With some more optimising of the motions used and the parameters within the program I feel this height could be pushed to around ten degrees and the run time decreased. Also, shortly after finishing this program, we realised there is a more effecient way to drive the swing when squatting; this is to squat at either peak then stand up when going through the origin. Unfortunately under the time constraints it was not possible to test this theory and make possible changes to the program.

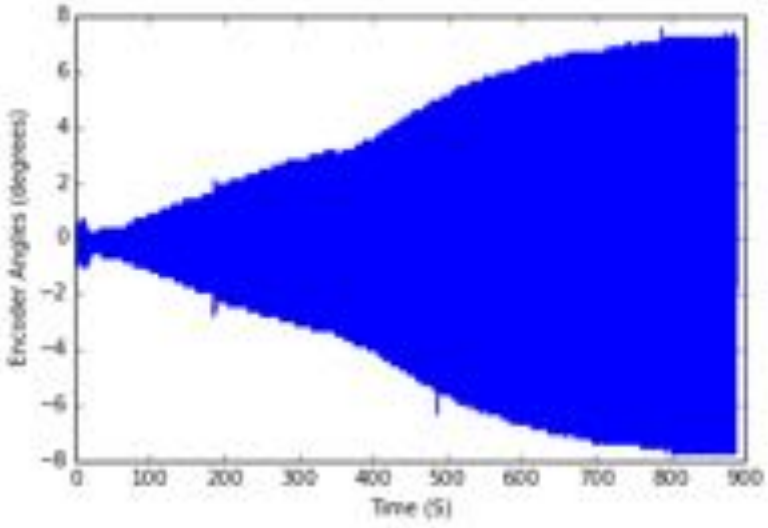


Figure 4.6.8: Encoder Values for Nao swinging from rest

Chapter 5

Critique

What follows is a critical analysis of how the group study was executed as a whole. A questionnaire was distributed to the group asking for their opinion on certain aspects of implementation (see Appendix). The objective of this critique is to inform the subsequent year of the "hazards" group orientated research can present; especially to those who have not studied in a group related environment before, and educate them on what aspects of physics, robotics and computer science may be the most beneficial.

A recurring theme throughout the submissions is that perhaps the **Machine Learning** (ML) sub-group should have established itself earlier in the study. This is easy to say in hindsight; when the study began we did not all realise how pivotal ML *could* be to the success of it. This does not undermine the work done by the programmers who "hard-coded" NAO; utilising feedback from the encoder. Fundamentally, this was successful and allowed NAO to swing. But through the work undergone by the programmers a solid foundation concerning ML has been supplied to the next year (see *Legacy*). In regards to ML, subsequent years are strongly encouraged to make good use of both the concept and the documents left for them.

A few sub-groups (Mathematical Simulations and Robot Programmers in particular) expressed that valuable time was taken up by simply installing software. This will not necessarily be a problem for the subsequent years, but if it is the advice given would be to *plan* for this ahead of time and engage in other research or work relevant to the sub-group while the computer is installing said software. It may not be immediately obvious, but there is always something that needs doing or benefits the study. Although not a self-critical point to make per se, the consensus is that there were too many people in the group. A number of people felt that they were sometimes left with nothing to do; but, again, in hindsight this could have been overcome with better time management in the group as a whole.

Regarding time management again, it is strongly recommended that each sub-group create a set of **specific** goals early on; listing those that are essential, unessential and those to do if time permits (i.e. research concerning further study). This is probably conspicuous. A matter raised numerous times was that students did not always feel they were working on anything "tangible", or that what they were doing was not going to be useful in the "grand scale". If a set of detailed goals listed in importance is developed early on this is less likey to happen, and students will always see "the bigger picture".

Linking into another issue raised; communication is *vital*. Communication between the sub-groups is key to the success in this project - where it lacked earlier on in this study, it was made up for in the last couple of weeks when real goals where achieved by sub-groups working in parallel. Furthermore, is it absolutley essential that subsequent years use the "group meetings" time effectively, and not underestimate the potential they have. When a large group is together, problem solving is much more fluent; everybody can give their opinion on a certain task and the broad spectrum of knowledge present is really exploited. If sub-group communication is lacking, the group meetings are an ideal way to convey your team's needs to the group as a whole. This group made good use of the meetings, and it is suggested that subsequent years do the same.

Chapter 6

Conclusion

——————— The following was contributed by: Joe Preece ———————

Over the course of this project, the resources available were utilised effectively in order to meet the desired outcomes that were set at the start. Various groups helped towards meeting the overall project goals.

Initially, a specific group existed to ensure that Nao was set up correctly, and to become accustomed to the mechanisms of the robot. Another responsibility was to test Nao's feasibility for the project.

The group studying the mathematics and physics of swinging made simplified models by researching existing literature and analysing the fundamental physics of swinging. By creating appropriate equation solvers, and working from the mathematics of starting swinging from rest, the group was able to provide the rest of the group with useful information regarding swinging.

Meanwhile, the hardware group looked towards designing a swing of maximum efficiency. This included designs for modifications to the original design, as well as installing an encoder in order to detect the angle at which the swing was at at a certain time. This date could be used by the programmers if necessary, or for running friction tests. This group was also responsible for researching and analysing how humans swing from video data.

Once the designs for the swing were in place, they were obtained by the virtual environment team, who replicated the schematics virtually within the programme Webots. After the swings were designed, virtual tests with modifiable physics could be run, to test what Nao might experience on the real swing.

The programmers were then responsible for designing scripts that could be uploaded to the robot. The final method used was pre-coded motions. However, machine learning was touched upon.

To conclude, the amalgamated efforts from all subgroups provided an overall success, in that we met our goals of having the robot swinging from standing. Unfortunately, time constraints prevented code for the robot swinging from sitting being developed. However, the swinging from sitting was investigated thoroughly in a mathematical sense, to leave a legacy for future years.

Appendix A

Guides

The following was contributed by: Ashe Morgan

A.1 Using Tracker 4.87 (Douglas Brown)

Using Tracker 4.87 (Douglas Brown) [[cite key: tracker2 here]] to track the movement of the NAO Robot on a swing using paper markers.

Tracker is an open source software package, which is able to analyse and track points from an imported video. Once tracked, the position, acceleration, velocity and angle relative to the origin of each point is calculated and can be easily exported for further analysis in other programs such as Matlab.

A.1.1 Step One: Taking the Video

Tracker is able to track points that are uniquely coloured/shaped with respect to the other objects in the field of view. The Tracker documentation suggests creating markers to place on the object to be tracked. Tracker suggests creating a white circular marker with a coloured circular center. The markers used in the first (2015) version of this Robotics Group Study can be found underneath the polystyrene in the NAO robot box, alongside some coloured paper to create further markers. The coloured center is of radius 1cm and the outer white circle of radius 2cm.

Choose colours for the marker and take the video in such a way that the markers are unique within your field of view. People with similar colour clothing in the field of view could confuse the tracking program.

It is very important that the camera used to take the video is fixed using some sort of tripod. A clamp stand and a phone can be used to take the video, which was the approach that was used in this Group Study.

Lastly, Tracker can calibrate pixel distances in the video and convert them to lengths using a calibration tape. For this it is necessary to create a calibration length within your field of view (preferable close to the object to be tracked), which is of known length. Permanent marker points separated by a known distance are an example.

A.1.2 Step Two: Importing and Tracking the Video

Make sure that the objects to be tracked are visible throughout the entire duration of the video. It should be noted that the tracking of points slows dramatically the further along a video Tracker gets. Therefore video length should be kept to an absolute minimum whilst still capturing the necessary data and some movie editing will more than likely be required to compress the video.

Open tracker. Select ‘Video’, then ‘Import’. Select the video you wish to import. Once the video is imported, create the axes by clicking on the ‘show or hide the coordinate axes’ button as shown in Fig A.1.1. The axes can be dragged so that the origin is where desired and can also be angled if needed.

Permanent marker points can be seen on the rod of the swing, this was for use as a calibration tape as the distance between the bottom and top marker was known as 15cm.

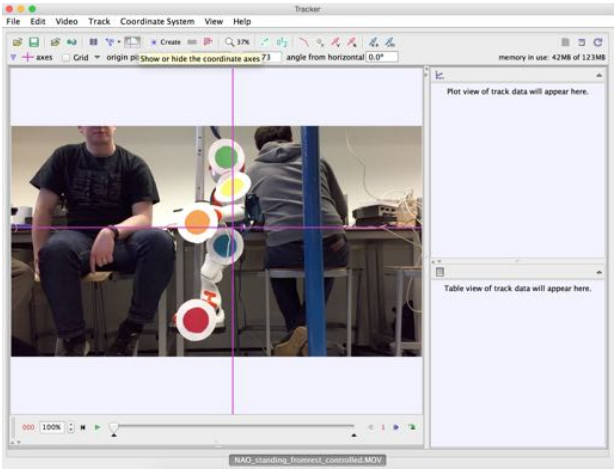


Figure A.1.1: Importing the video and set-up of axes

Create the calibration tape as shown in Figure A.1.2. This must be done before any tracking takes place. The calibration tape must be put over the length of the marker used and the distance inputted in the unit of your choice.

Select 'Track', 'New', 'Point Mass' from the program menu. Click on the box named 'mass A' and edit its properties as desired. The footprint can be changed and if a circular marker were used then a circular footprint would be best. Input the radius of the footprint when requested. This process can be seen in Fig A.1.3.

Hold *shift-ctrl* and left click on the marker to be tracked. On a Mac it is *shift-ctrl*, not *shift-cmd*. The windows shown in Fig A.1.4 will appear. The small, solid inner circle is the area that a template match image is taken from; this can be changed by dragging the handle on the low right hand corner of the box. The dotted square is the search area for the next frame and can also be altered in the same manner.

The window on the right hand side of Fig A.1.4 is the 'Autotracker' window. The 'evolution rate' is a measure of how much the template image evolves over time, with a value of 100% meaning the template image is changed after every frame. The 'Automark' setting is used to set the acceptance level of the image match. A high 'Automark' setting only allows the best match to successfully track the point.

Click on the 'Search' button and the video is automatically tracked for the selected marker on each frame. The trail of the tracked object can also be seen as a useful check that the point is being tracked correctly. This process can be repeated for more data points as required.

A.1.3 Step Three: Exporting Data

After tracking, the table of output data can be altered to include more variables by clicking on the table icon in the red circle shown in Fig A.1.5. This data can then be exported to a file by clicking on 'File', 'Export', 'Data File'.

A.1.4 Comments

1. The memory allocated to Tracker can be set by clicking on the text in the top right-hand corner and selecting 'Set memory size'. This can be useful as the default allocated size is small and as a result the program can run slowly, especially when analysing long videos.

2. The refresh button in the top right-hand corner when clicked gives the option of turning auto-refresh on or off. For long videos this should be turned off according to the documentation.

——— The following was contributed by: Christopher Hounsom ———

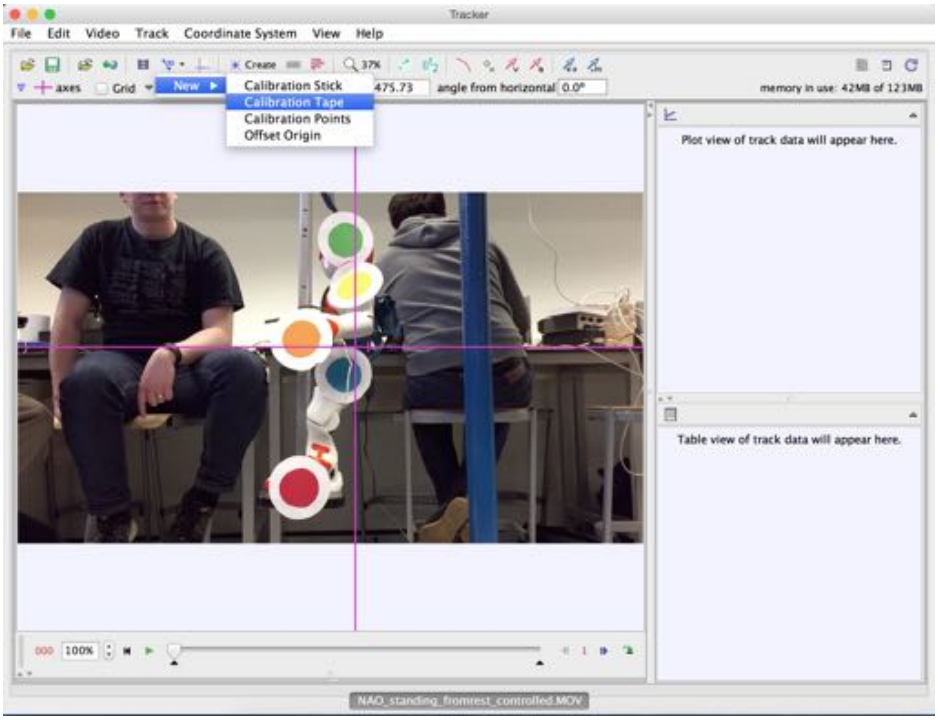


Figure A.1.2: Using and setting up a calibration tape

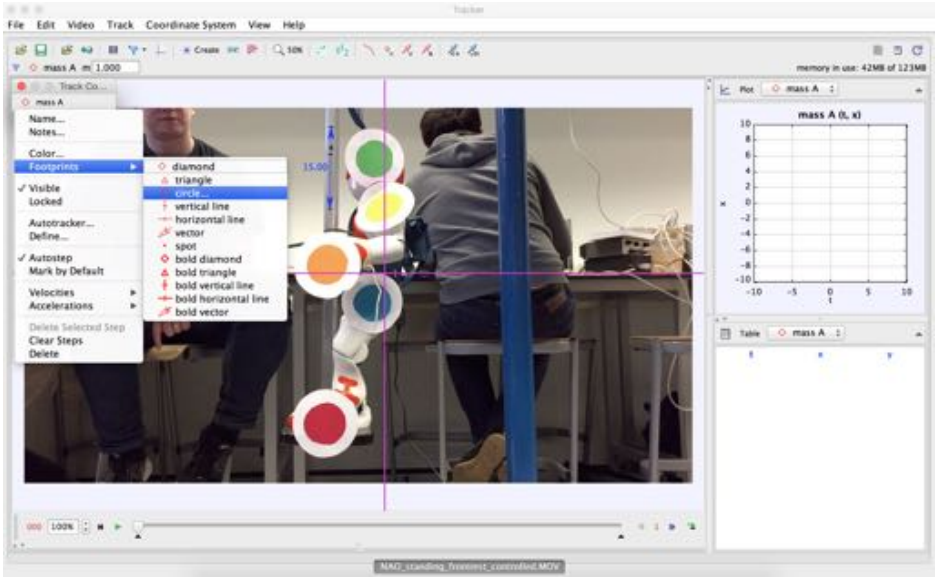


Figure A.1.3: Adding a tracking dot ('point mass')

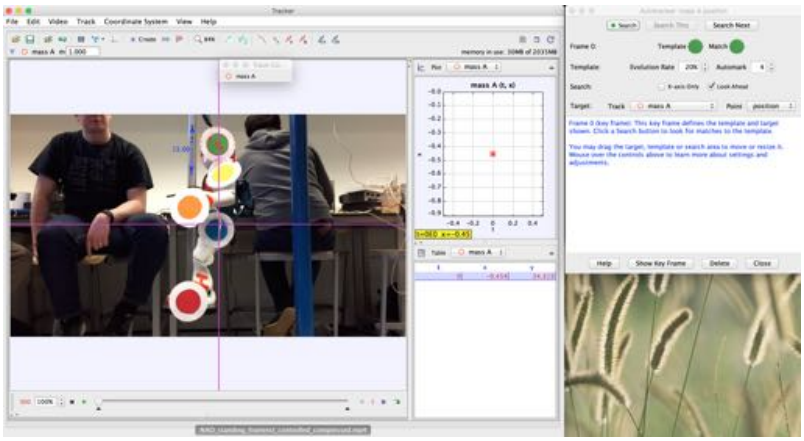


Figure A.1.4: Using 'Autotracker'

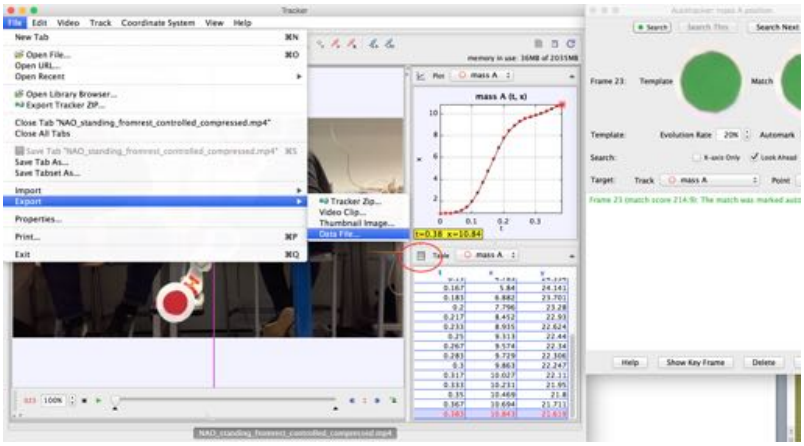


Figure A.1.5: Exporting data to a file

A.2 How to simulate NAOqi code in Webots

Firstly, the controller of the virtual NAO in Webots must be set to `naoqisim` and the simulation must be running to allow for connectivity with Choregraphe. The NAO documentation describes how to successfully connect Choregraphe to this simulated robot [14].

Create a new Python box in Choregraphe, changing the name and description of the box as required. The `RemoteNao` class (or any class) can simply be included above the `MyClass` class that is a default of the Python box, as in `naoChoregraphe.py` (note that this file does not contain the latest version of `RemoteNao`).

A few changes to the `RemoteNao` class written for the real NAO must be made before the code will work in Choregraphe:

- As the virtual NAO does not connect to Choregraphe in the same way as the real NAO, any arguments of `ip` or `port` need to be removed when in Choregraphe. For example, in `RemoteNao`:

  ```
  self.motion=nq.ALProxy(''ALMotion",ip,port)
  ````

  must be replaced by

  ```
  self.motion=nq.ALProxy(''ALMotion")
  ````

- Console input and output is not possible in Choregraphe so must be removed

- There are no text to speech capabilities for the virtual NAO so it is recommended to remove any code involving text to speech.

To instantiate the NAOqi class, a few changes must be made to the default class of the Python box:

- Create a new instance of the class in the `onLoad` method
- Execute methods of the instance in the `onInput_onStart` method (as if it were your main function)

——————— The following was contributed by: Christopher Hounsom ———————

A.3 Building an ODE Simulation

This Appendix details the process of creating a model system in ODE for use in a PyBrain reinforcement learning project. Examples of this are provided in the files accompanying this report: `pendMakeXODE.py` creates the simple flywheel pendulum and `xodenao.py` contains the code used to make both the standing swing and NAO simulation models.

ODE models are made up of combinations of cylinders, boxes and spheres and are written in XML. Prior knowledge of XML is not required, however, as PyBrain comes included with a Python script that allows for models to be created using Python.

In order to use the script, a new class must be created that extends `XODEfile`. If using the Linux virtual machine submitted with this report, this is imported using

```
from pybrain.rl.environments.ode.tools.xodetools import XODEfile
```

Opening `xodetools.py` shows the methods available to create ODE models, which will be detailed in this Appendix. In addition, the methods used to create several PyBrain examples, including `johnnie`, can also be found in this file.

The key methods used in creating a model will be detailed and their arguments explained. To see them in use, examine the examples mentioned previously.

`insertBody(bname, shape, size, density, pos=`$[0,0,0]$ `passSet=None, euler=None, mass=None, color=None)`

- `bname` - User defined identifier for the body e.g. `'head'`
- `shape` - Can be any one of: `'box'`, `'cylinder'`, `'cappedCylinder'`, `'sphere'`
- `size` - This argument is dependent on the shape used:
  - if `'box'` - `[xsize, ysize, zsize]`
  - if a `'cylinder'` - `[radius, length]`
  - if `'sphere'` - `[radius]`
- `density` - If `mass` is given, this value is ignored
- `pos` - `[x,y,z]`
- `passSet` - Custom string identifier. All bodies with the same `passSet` identifier can penetrate each other. This behaviour is required to insert joints between two bodies. Each body can have multiple identifiers, separated by a `,`. Note a `sphere` cannot be used with `passSet`.
- `euler` - `[xrot, yrot, zrot]` where the rotation is in degrees
- `mass` - If specified, `density` is calculated from this value
- `color` - `[r, g, b,` α`]` where α is the opacity of the colour: $\alpha = 0$ is transparent, $\alpha = 1$ is opaque.

insertJoint(body1, body2, type, axis=None, anchor=$(0,0,0)$, rel=False, name=None) Joints are used to connect bodies to each other

- `body1, body2` - Identifiers for the bodies to be connected
- `type` - Can be any one of: `'fixed'`, `'hinge'`, `'slider'`, `'ball'`
- `axis` - e.g. {`'x':1,'y':0,'z':0`} which must be provided as a dictionary. This describes the axis that the joint can move through.
- `anchor` - `(x,y,z)`, the co-ordinates of the attachment point. Only necessary for `hinge` and `ball` joints.
- `rel` - Boolean that, if set to `True`, defines the `anchor` co-ordinates realtive to the body's origin.
- `name` - Set a user defined identifier for the joint. By default the name is `'body1_body2'`

insertFloor(y=-0.5) Inserts a solid floor into the model in the defined y-plane.

affixToEnvironment(name) - Specify a joint to be pinned to the world and not fall under gravity.

centerOn(name) - Specify a joint/body to focus the camera of the viewer on.

writeXODE(filename=None) - Write the resulting model to an `.xode` file that can be imported by the PyBrain project. This should be called outside the class in `main` following an instantiation of the users `xode` class:

```
if __name__=='__main__':

    inst=modelMakeXODE('filename')
    inst.writeXODE()
```

The above code creates a new instance of the users `modelMakeXODE` class and writes the corresponding XML code to `filename.xode`.

In order to write the `.xode` file, navigate to the directory with the file containing the users `modelMakeXODE` equivalent file and run the file with

```
python modelMakeXODE.py
```

The console should print

```
wrote filename.xode
```

to indicate success.

With the `.xode` file created, ensure the PyBrain environment has the correct filepath to import this file. Then run the PyBrain main class with Python. In order to observe the learning process, start ODE Viewer by navigating to

```
pybrain/pybrain/rl/environments/ode
```

and running `viewer.py` in Python.

Note: To see the PyBrain examples running, navigate to

```
pybrain/examples/rl/environments/ode
```

and run any of the `.py` files in the directory in conjunction with the viewer.

——————— The following was contributed by: Stuart Bradley ———————

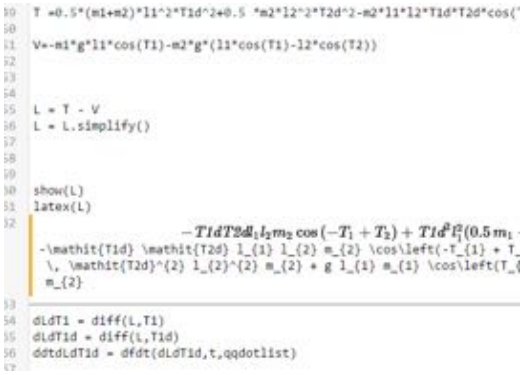


Figure A.4.1: An example of a sage worksheet

A.4 SageMathCloud

SageMathCloud is an online cloud-based computer program which can make some of the more difficult algebraic tasks you have to perform a lot easier. It can be found at https://cloud.sagemath.com. It supports Sage worksheets, Latex and Python, which is very useful for a project of this type, as it allowed us to easier keep track of complicated algebra and then give us an output of a latex expressions of anything we use or work out, for copying straight into report at the end. It has many inbuilt methods already, some of which can be used for solving equations, handling derivative, simplifying expressions, substitutions, plotting python graphs and many other useful applications, more on these later.

A.4.1 Basics

Stuart Bradley

A.4.1.1 Assignment and Basic Operators

Since the coding in sage is python, assignment might be slightly different to what you are expecting or used to if not familiar with python. There are no static types so
a=5,
a=1/5,
and a='tree' are all just as viable as each other. You can do it with more complicated expressions involving variables too.
x1 = l1*sin(T1)

Other than that most operators are the same as you'll be used to.
+ (addition)
- (subtraction)
* (multiplication)
/ (division)
% (remainder)
** or ^ (exponential)
== (equals)
!= (not equal to)
<(less than)
<= (less than or equal to)

A.4.1.2 Creating variables

Variable creation in sage is quite simple, first you give the letter or phrase you want to use in the code, followed by what you want it to be represented as in the output. As shown in figure A.4.2

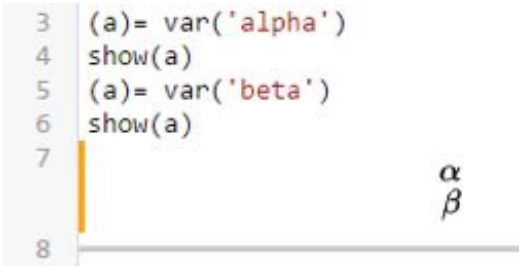


Figure A.4.2: An example of a creating a variable

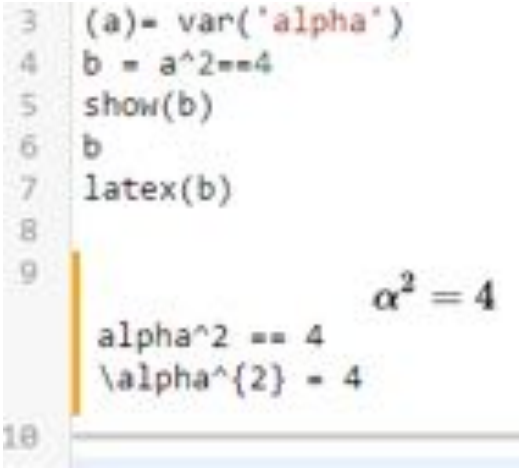


Figure A.4.3: An example of a creating a variable

It is also possible to declare multiple variables in one line by using the syntax shown below,

(m1,m2,M,g,l1,l2) = var('m1 m2 M g l1 l2')

A.4.1.3 Creating Equations

In sage you can also create equations, this assignment but in this case you are assigning two sides of an equality instead of just on expression. For example,

Eqn1 = T == 0.5*(m1+m2)

mgmg = T+F-2*a == 0.5*(m1+m2)

A.4.1.4 Output

In figure A.4.3 you can see a simple equation and then three different ways of displaying this equation as the output of the cell.
- Show() displays whatever object is as it would appear in properly formatted latex.
- Just typing in the object causes it to appear in the code you could use in sage.
- Latex() returns the latex code needed to produce show if it was ran in a latex editor.

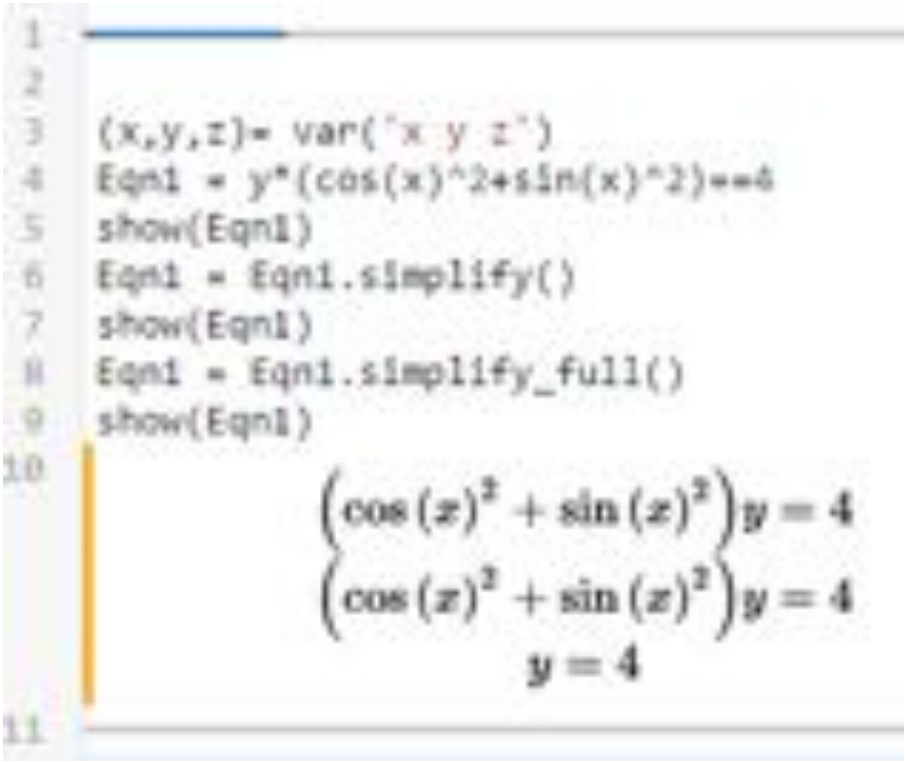


Figure A.4.4: Uses of .simplify() and .simplify_full()

A.4.2 solve() and Algebra Manipulation

A.4.2.1 .subs()

.subs() is a function that can be used for substituting variables and expressions in an equation, the syntax for using it is as follows,

Eqn1 = Eqn1.subs(To be replaced == To replace with)

A.4.2.2 .simplify() and .simplify_full()

Sage has some great facilities to assist with handling large messy bulks of algebra, though it doesn't always process it in the way we might expect as a human. Just because it looks right to us doesn't mean that it is in its simplest form by the strictest letter of the law. .simplify() as you might imagine is used for simplifying long expressions into (hopefully) shorter more manageable ones, as hinted earlier this is not always the case, it seems to be by no means fool proof. For the most part, the automatic simplification that SageMathCloud does seems to cover most of the simplification that .simplify() would do, however there is a more "powerful" version of it .simplify_fulll().

A.4.2.3 .solve()

The solve() function has many slightly different iterations that allow you to solve different types of equations, some of which are shown below in figure A.4.5. There are many combinations of single or multiple variables in single or multiple equations, that it's not possible to give examples of them all, the examples below are just to give you an idea of how these might look. In general the syntax is

solve([$Equation_i$,$Equation_{ii}$,.....], Variable to be solved for, Variable to be solved for,.....)

You also don't have to have defined an equation either to use solve you can just right it straight into the solve() function. As a default if you are using solve() the output will be symbolic so if you wish for a numerical

```
(x,y,z)= var('x y z')
Eqn1 = x^2==4
Eqn2 = x^2 + y^2 == 8

solve([Eqn1], x)
[x == -2, x == 2]

solve([Eqn2], x)
[x == -sqrt(-y^2 + 8), x == sqrt(-y^2 + 8)]

solve([Eqn2,Eqn1],y,x)
[[y == 2, x == -2], [y == -2, x == -2], [y == 2, x == 2], [y == -2, x == 2]]
```

Figure A.4.5: Some uses of solve()

```
(x,y,z)= var('x y z')
f = y*cos(x)
#Derivative with respect to z
diff(3*sin(1/3*z),z)
#Partial derivative of function f with respect to x
f.diff(x)
#Partial derivative of function f with respect to y
f.diff(y)
cos(1/3*z)
-y*sin(x)
cos(x)
```

Figure A.4.6: Examples of differentiation

output and haven't got one you can use the find_root() function instead, it takes slightly different arguments though so I recommend looking in the API before using it.

A.4.3 Differentiation and Integration

There are already inbuilt function already for handling many of your differentiation and integration needs, however in this section I am also going to include a code section, that was supplied to us courtesy of Dr Wolfgang Thesis that assists in handling derivatives, when you have many variables. For differentiation we have the diff() function which can be used to calculate both partial and full derivatives.

It is much the same for integration, except using integral() instead of diff(),

integral(cos(x))

would give us an output of sin(x).

In figure A.4.7 There is code that allows you to differentiate variables that are a function of time with respect to time, for instance sage won't know that $\dot{\theta}$ is the time derivative of θ unless tell it that in a way it understands.

A.4.4 Plots and Graphs

Sage has many many tools for drawing graph and plots, both two dimensional and 3 dimensional, you'll find a complete list of them at *http* : *//www.sagemath.org/doc/tutorial/tour_plotting.html*. Because of how many

```
# differentiate with regard to t, where qqdotlist gives [q_i, qdot_i] pairs to consider
def dfdt(f,t,qqdotlist):
    g = diff(f,t)
    for qqdot in qqdotlist:
        g = g + diff(f,qqdot[0]) * qqdot[1]
    return g
#The code above defined a new function dfdt, that checks if each variable to be differentated
has a pairing in the qqdotlist, if it is the first one then it can be differentated with respect
to t, to give the second element.

#example
(t,a,b, ad, bd, c) = var('t a b alpha beta c')
#qqdotlist defines the variables that are functions of t and their derivatives with respect to t
qqdotlist = [[a,ad],[b,bd]]
#L is an expression to be differnentiated with respect to t.
L = a^2 + b + c + t + 7
y=L.diff(t)
show(y)
#Just using diff() doesn't take into account that any of the other variables may be functions of
t, as shown by result of 1
x = dfdt(L,t, qqdotlist)
show(x)
#Using the newly defined function does take this into account of functions of t and those that
aren't, hence why a and b are both differentiated but c goes to 0.
```

$$2a\alpha + \beta + 1$$

Figure A.4.7: Very useful custom function for handling multiple variables that are functions of t

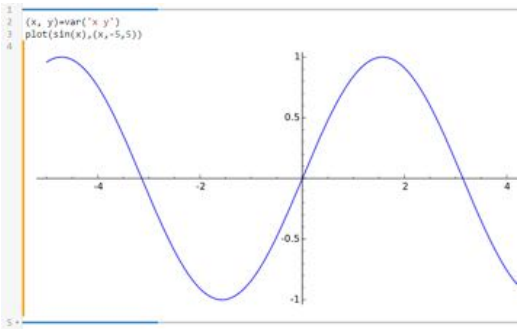


Figure A.4.8: Simple 2D plot

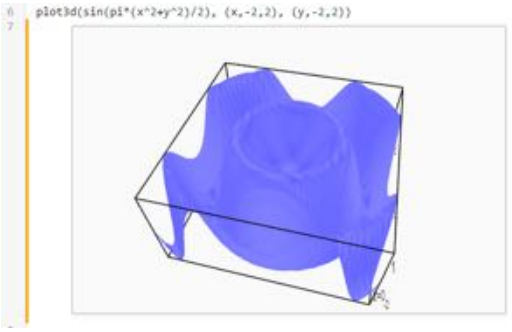


Figure A.4.9: 3D plot example

there are, I'll just cover the most basic two, to give you an idea of how to use the plot tools.

The first argument in the function is what you want to be plotted, and the second is the variable and the range that you want it in between.

The 3 dimensional plot is just the same as the two dimensional plot, the first argument in the function is what you want to be plotted, and the next two are the variable and the range that you want it in between.

This concludes explaining the most basic features of SageMathCloud, there are plenty of other things you can do and inbuilt functions to take advantage of that can be found on the SageMathCloud website.

Appendix B

Code

The following was contributed by: Vincent Fisher

B.1 Code for Pre-Codes Motions

SwingFromRest.py

This module is used to get Nao swinging from rest. Place Nao on the swing ensuring he is suitably attatched then run this program. This program makes use of concurrent programming using pythons multiprocessing module. This allows data to be taken at the same time as the program running. This is as parts of this program use time.sleep() which blocks the entire program if its running sequentially.

@author: Vincent Fisher

```
import Nao
import sys
import time
from multiprocessing import Process, Value
IP = '192.168.1.6'
PORT = 9559
robot = Nao.RemoteNao(IP, PORT)
pathToData = '/home/vince/pyCode/Data/SwingFromRestData'


def writeData(acceptValue):
    import encPy
    Data = open(pathToData, 'w+')
    timeInit = time.time()
    encPy.calibrate_zero()
    while True:
        shareAngle=encPy.get_angle()
        sys.stdout.write("\r" + str(shareAngle))
        sys.stdout.flush()
        Data.write(str(time.time() - timeInit) + " " + str(shareAngle) + "\n")
        acceptValue.value=shareAngle
        time.sleep(0.001)

def swingNao():
    ValueMemory = Value("f", 0.0)
    writeDataProcess = Process(target=writeData, args=(ValueMemory,))
    writeDataProcess.start()
    pushSpeed = 0.7
```

```
    pullSpeed = 0.7
    elbowSpeed = 0.7
    neckSpeed = 0.7
    squatSpeed = 0.5
    waitVariable = 1
    squatSwitch = False
    sleepTime=1.15
    forward = True
    ang1 = 0
    maxAngle = 0
    while (True):
        angleHolder = ValueMemory.value

        if (abs(angleHolder) > maxAngle):
            maxAngle = abs(angleHolder)

        if(maxAngle >= 4):
            squatSwitch = True

        if (maxAngle < 2):
            ang2 = angleHolder
            if ang2 >= 0 and forward is True:
                robot.pushSwing(pushSpeed,elbowSpeed,neckSpeed)
                forward = False
                time.sleep(sleepTime)

            elif ang2 < 0 and forward is False:
                robot.pullSwing(pullSpeed,elbowSpeed,neckSpeed)
                forward = True
                time.sleep(sleepTime)

        elif(maxAngle>=2):
            ang2 = angleHolder
            angleBigger = robot.angleGreater(ang1,ang2)

            if angleBigger is True:
                waitVariable = 0

            if angleBigger is False and waitVariable == 0:
                if ang2 > 0:
                    if squatSwitch:
                        robot.squat(squatSpeed)
                    else:
                        robot.pushSwing(pushSpeed,elbowSpeed,neckSpeed)
                    waitVariable = 1
                elif ang2 < 0:
                    if squatSwitch:
                        robot.squatUp(squatSpeed)
                    else:
                        robot.pullSwing(pullSpeed,elbowSpeed,neckSpeed)
                    waitVariable = 1
            ang1 = ang2

swingNao()
```

SwingInMotionEnc.py

Program which allows Nao To swing from a pre-existing motion using the encoder to dictate when movements should occur. Movements programmed to be executed at the peaks of a cycle. Choose which motions to use by editing the movement variable to be either "Arms" or "Squat". You can adjust the speeds of these motions by editing any of the *Speed parameters. A value of 1 corresponds to the fastest motion available. Press Ctrl+C to exit the program

@Authors: Vincent Fisher & Jack Ford

```
import encPy
import Nao

IP = "192.168.1.6"
PORT = 9559
robot = Nao.RemoteNao(IP, PORT)
encPy.calibrate_zero()
movement = "Arms"
pushSpeed = 0.7
pullSpeed = 0.7
elbowSpeed = 0.7
neckSpeed = 0.7
squatSpeed = 0.5
waitVariable = 1

ang1 = encPy.get_angle()
while (True):
    try:
        ang2 = encPy.get_angle()
        angleBigger = robot.angleGreater(ang1,ang2)

        if angleBigger is True:
            waitVariable = 0

        if angleBigger is False and waitVariable is 0:
            if ang2 > 0:
                if movement is "Squat":
                    robot.squat(squatSpeed)
                elif movement is "Arms":
                    robot.pushSwing(pushSpeed,elbowSpeed,neckSpeed)
                else:
                    print("No movement chosen please use 'Squat' or 'Arms'")
                    break
                waitVariable = 1
            elif ang2 < 0:
                if movement is "Squat":
                    robot.squatUp(squatSpeed)
                elif movement is "Arms":
                    robot.pullSwing(pullSpeed,elbowSpeed,neckSpeed)
                else:
                    print("No movement chosen please use 'Squat' or 'Arms'")
                    break
                waitVariable = 1

        ang1 = ang2
```

```
    except KeyboardInterrupt:
        del(encPy)
```

SwingInMotionGyro.py

This program can detect when Nao is at maximum amplitude using his gyroscope. It is designed so he then executes a motion at these points. Problem: When movement is executed gyroscope value spikes and breaks programs intentions. Something to look into for Robotics Group Studies 2016
@author: Vincent Fisher

```
import Nao

robot = Nao.RemoteNao("192.168.1.6", 9559)
gyro1 = 10
gyro2 = 10
ran = False
ran2 = False
movementExecuted = False

while (True):
    gyro1 = robot.getGyroscopeValues()[1]
    var2 = robot.gyroChangedSign(gyro1, gyro2)

    if not ran:
        if var2 == -1:
            ran2 = True
            robot.pushSwing(0.4, 0.4, 0.4)
            movementExecuted = True

        elif var2 == 1:
            ran2 = True
            robot.pullSwing(0.4, 0.4, 0.4)
            movementExecuted = True

        else:
             movementExecuted = False
             ran2 = False

    gyro2 = robot.getGyroscopeValues()[1]
    var = robot.gyroChangedSign(gyro1, gyro2)

    if not ran2:
        if var == -1:
            ran = True
            robot.pullSwing(0.4, 0.4, 0.4)
            movementExecuted = True

        elif var == 1:
            ran = True
            robot.pushSwing(0.4, 0.4, 0.4)
            movementExecuted = True

        else:
```

```
        movementExecuted = False
        ran = False
```

NaoTester.py

A sample program which shows how you can test methods present inside the Nao.py module. Simply run the program and type in the number of the corresponding method you wish to test then press enter. press 0 then enter to exit the program
@author: Vincent Fisher

```
import Nao
IP = "192.168.1.6"
PORT = 9559
robot = Nao.RemoteNao(IP, PORT)

while (True):
    choice=int(raw_input())
    if choice is 1:
        robot.pushSwing(0.7,0.7,0.7)
    elif choice is 2:
        robot.pullSwing(0.7,0.7,0.7)
    elif choice is 3:
        robot.swingStandingInit()
    elif choice is 4:
        robot.openbothHands()
    elif choice is 5:
        robot.unstiffenAllJoints()
    elif choice is 6:
        robot.closebothHands()
    elif choice is 7:
        robot.fallDetectionOff()
    elif choice is 8:
        robot.swingNeckUp(0.5)
    elif choice is 9:
        robot.swingNeckDown(0.5)
    elif choice is 10:
        robot.squat(0.5)
    elif choice is 11:
        robot.squatUp(0.5)
    else:
        break
```

Bibliography

[1] Advantages of Object oriented programming. Last accessed 18/03/2015 19:02.

[2] ALValue documentation. Last accessed 20/03/2015 22:23.

[3] Introduction to inheritance.

[4] Nao Hardware Documentation - Inertial Unit.

[5] NAO Software 1.14.5 Documentation - NAOqi. Last accessed 20/03/2015 16:30.

[6] NAO Software 1.14.5 Documentation - NAOqi Framework. Last accessed 20/03/2015 21:05.

[7] NAO Software 1.14.5 Documentation - SDKs.

[8] NAO Software 1.14.5 Documentation - Using Python in Choregraphe. Last accessed 20/03/2015.

[9] NAO Techincal Manual - Motor Information.

[10] PyBrain - Reinforcement Learning. Last accessed 19/03/2015.

[11] Python Essays. Last accessed 20/03/2015 16:30.

[12] SciPy Curve Fit.

[13] Single-Threading: Back to the Future? Last accessed 20/03/2015 22:12.

[14] Webots for Nao - Connecting Choregraphe to the simulated robot. Last accessed 20/03/2015.

[15] Webots Reference Manual.

[16] Webots User Guide.

[17] Welcome to PyBrain. Last accessed 19/03/2015.

[18] Vangie Beal. Details on polymorphism. Last accessed 18/03/2015 21:43.

[19] Douglas Brown, March 2015.

[20] W. B Case and M. A. Swanson. The Pumping of a Swing from the Seated Position. *American Journal of Physics*, pages 58, 463–467, 1990.

[21] Tesca Fitzgerald and Andrea Thomaz. Skill Demonstration Transfer for Learning from Demonstration.

[22] Javier A. Kypuros. Inverted Pendulum Control System.

[23] J. D. Lambert. *Numerical Methods for Ordinary Differential Systems*. Wiley, 1992.

[24] Kenneth C. Louden and Kenneth A. Lambert. *Programming Languages: Principles and Practice*. Course Technology, 3rd edition edition, 2012.

[25] A Post et al. Pumping a Playground Swing. *Motor Control*, pages 11, 136–150, 2007.

[26] Jette Randløv and Preben Alstrøm. Learning to Drive a Bicycle using Reinforcement Learning and Shaping.

[27] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, 3rd edition edition, 2009.

[28] Peter Sestoft. *Programming Language Concepts*. Springer, 1st edition edition, 2012.

[29] David Silver. Lecture 7: Policy Gradient.

[30] Richard S. Sutton et al. *Policy Gradient Methods for Reinforcement Learning with Function Approximation*. MIT Press, 2000.

[31] Stephen Wirkus, Richard Rand, and Andy Ruina. How to pump a swing. *College Mathematics Journal*, 29:266–275, 1998.

[32] Paul Zandbergen. Object oriented programming. Last accessed 18/03/2015 18:36.

Index of Authors