CONTROLLER FOR JUMPING ANIMATIONS TO ACHIEVE TARGET POSITIONS

By

Ian Charles Ooi

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Examining Committee:

Barbara Cutler, Thesis Adviser

Charles Stewart, Member

Shawn Lawson, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2015
(For Graduation August 2015)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Algorithms

CHAPTER 1
INTRODUCTION

Animations of human characters are used heavily in video games, movies, and other fields. Especially with the increasing usage of complex environment traversal in both film and video games, many similar animations of athletic motions must be created with small changes to tune the motion to the particular situation, environment, and character. Creation of such animations is largely done by hand by artists using keyframing. In a keyframe animation, certain "key" parts of the animated sequence are specified, with the remaining frames filled in, or "tweened" using an automated interpolation method or manual frame addition. For 2D animation, this occurs as a series of images which are played back in order to produce the animation. In 3D, keyframe animations are performed on a 3D model.

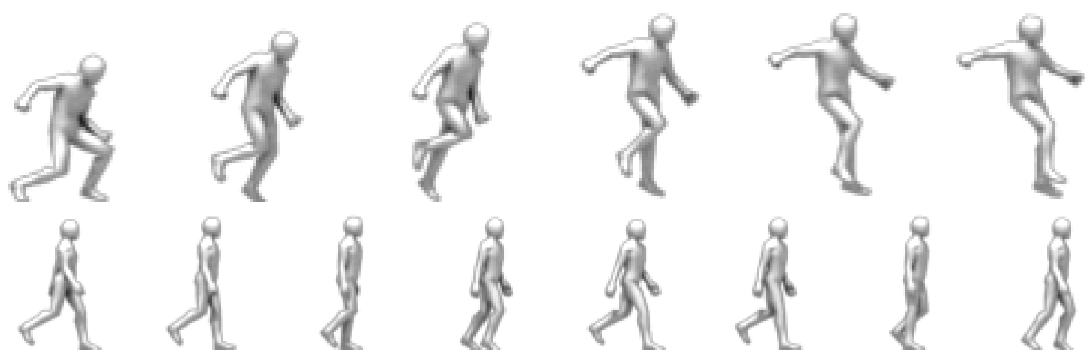

Figure 1.1: This example shows a 2D sprite sheet used to produce a jumping animation for a stick figure character. The frames in this case are laid out in a single image for demonstration purposes, progressing in order starting with frame 0, the frame farthest left in this sprite sheet.

3D models are described as a mesh, a collection of primitive polygons (i.e. quadrilaterals or triangles) which are stored as vertices. This mesh describes what is drawn, including any texture, color, and other material information. Along with the mesh, a skeleton, or rig, is stored. The rig describes a heirarchical structure of bones and accompanying joints. Each vertex is given a series of weights describing the impact each joint has on its transformation. This allows many vertices, and therefore many polygons, to be transformed at once in organized groups, simplifying the problem of animating the model to a matter of transforming the skeleton in the desired manner. To animate this 3D model, an artist specifies keyframes of the animation by positioning the skeleton at different time steps. The stored keyframes, instead of an image, are the transformations of each joint at this frame or step of the animation, which a rendering or game engine can interpolate between to produce the final result.

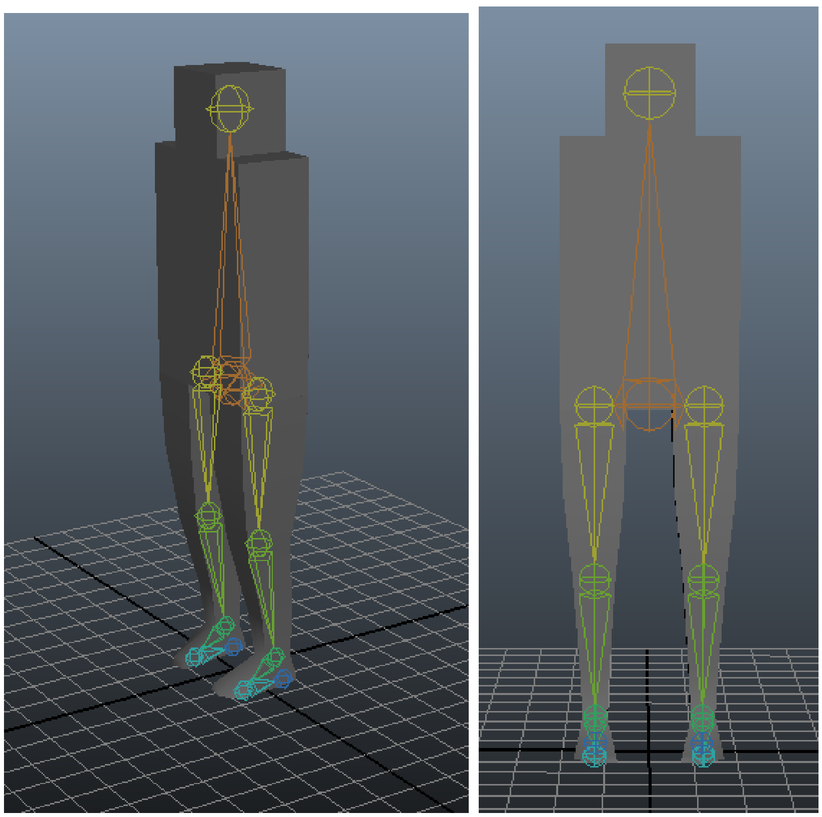

Figure 1.2: Above is an example of a character model in AutoDesk Maya. The character skin or mesh is shown in gray, with a rig shown in multicolor. While it is visualized as a series of spherical joints with connecting solids, the rig itself does not have a visual component in practice. The rig acts as a skeleton, deforming the mesh of the character model to make animation easier. Additional tools such as deformer groups and inverse kinematics handles can be used to further simplify the creation of animations for artists. While these tools simplify movement of the model, the artist still must position each joint for each frame of the animation, which is then stored for later playback.

Specifying these animation frames is work intensive, taking up significant time and resources to produce for a single character. Additionally, similar animations may need to be produced for slightly different scenarios, with only minor modifications required. These minor modifications can be to fit a different setting, such as a character jumping on Earth or on the moon, or can be for different characters, such as a large person moving in contrast with a small child. Though the movement itself may be similar, manual changes must be made, requiring artist time which could be spent generating new assets.

Recent work in animation generation seeks to automate this process, replacing the manual process with a procedural one. Physics based simulations can be used to produce controllers for the skeleton, determining joint positions and rotations for keyframes automatically. Not only does this reduce the effort involved in the creation process, but this also provides a basis for dynamic interaction between a character's animation and the environment, which is not possible with manual keyframe animations.

We present a controller that takes a skeleton as input, with additional parameters describing the character, which produces a sequence of poses for a keyframe animation. The additional character parameters describe the character's mass as well as the constraints placed on each joint to prevent unnatural rotations. Weight is specified per-joint to allow for calculation of the character's center of mass. Our controller works with the initial flex and takeoff stage of the jumping motion. The motion is calculated by modeling muscles as simple springs attached to the skeleton at 2 points. Spring constants for the muscles are determined by the user, which are applied in a linear spring calculation to determine the approximate change from rest length required to achieve a particular force. This change from rest length approximates the flexion required, which can be used to calculate a plausible amount of bend for the wind-up motion of a jump.

To control the motion and maintain plausibility, calculations are performed to determine the character's center of mass and supporting polygon. As the character should maintain balance while flexing its joints, the position of the character's joints are adjusted to keep the center of mass positioned over the supporting polygon. Flexion proceeds with the character bending progressively, maintaining balance while moving to achieve the desired spring force in its leg muscles.

The next stage of the motion, the take-off where the character releases from the ground, uses the potential energy of the spring-muscles and accelerates the character's center of mass upward. Application of force works from the joints and muscles closest to the center of mass outwards through the skeleton. While not handled by our controller, the character would then proceed through the in-air portion of the jump, where the acceleration changes due to gravity as well as other forces before they finally land. We assume a simple trajectory for the in-air phase, though more complex motions with turns, flips, or interaction with the environment could be created. Other work has handled landing with a similar approach. [2]

Our controller is made to be a module, able to be used with other controllers as part of a larger system. This allows each controller to do a smaller job well. Several such controllers can be connected to produce more complex animations or animation sequences, utilizing bounded starting and finishing conditions for the character. Additionally, certain cases during the duration cause the controller to stop early, for example a mid-air collision mid jump which would require a separate controller to handle this case. [3]

1.1 Skeleton and Joints

For the purposes of animation, a joint is an object with an associated position, associated transformation, a parent joint, and some number of child joints. In the case of the root, the parent joint is absent and in the case of the end joints such as tips of the fingers there are no child joints. Each child joint is connected to the parent by a rigid bone, which protrudes from the parent at a given resting angle. These joints are structured in a tree, as the parent and child joints imply, with the root of the tree at the pelvis. This tree serves as a hierarchy for transformations.

Joints are associated with a set of vertices from the mesh to be animated. Each of these may be associated with multiple joints, and are assigned a weight for each joint which acts as a scale factor for the transformations performed on the vertex.

1.2 Contributions

This thesis describes a controller which simulates a jumping motion on a character. The generated animation is created to be plausible in appearance, though it may not be a physically accurate representation. Specifically, we contribute a model for the windup and take-off phases of a jump and created a controller using this model in Unity3D.

Unity 3D is a game engine which we used to develop this system. It provides infrastructure for rendering, scene management, skeletal animation, asset import and management, lighting, and scripting. The system developed leverages the provided features through the Unity3D scripting interface. We developed scripts for calculation of muscle forces, as well as for applying the force to an imported model. Models were created in AutoDesk Maya and imported using Unity3D's asset import as Unity3D game objects with attached Transform components which allow arbitrary transformations.

CHAPTER 2
PREVIOUS WORK

2.1 Background

Producing athletic animations for human characters is difficult. One method, motion capture is used for production of realistic animations for human athletics and other motions, however it requires the collection of information for each motion and does not adapt to the virtual environment. Muscle-based approaches produce realistic motions which adapt to the environment, using a complex model of the musculo-skeletal structure. Geijtenbeek et al. use a rough, user created muscle routing on a skeleton to produce various gaits that are learned based on the velocity and environment. The muscle routing is optimized to remain within a region while providing optimal forces on the skeleton based on freedom of motion of the skeletal joints and the calculated optimal length of the muscle. This model is then used to compute sequences of muscle activations, modeling neural signals, which produce the final animations. This method is effective, producing good results in various levels of gravity on at least 10 different bipedal skeletons [1].

Inverse kinematics approaches attempt to generate the motion based on a desired final position, determining the skeletal position by solving the system given constraints. Koga et. al use path planning, inverse kinematics, and forward simulation to generate animations of arm motions for robots and humans working cooperatively. They produce arm manipulations that avoid collisions and result in final positions and orientations for specified parts of the arm to produce motions such as a human putting on glasses and a robot arm and human cooperating to flip a chessboard [4].

Physical simulations utilize a rigid-body character with a user-defined skeleton to find optimal poses based on desired conditions. Ha et al. utilize such a scheme to generate landing motions for human characters based off linear velocity, global angular velocity, and angle of attack. The system chooses either a feet first or hands first landing strategy and moves into a roll to reduce stress on the body using principles from biomechanics and robotics. A sampling method is applied to determine successful conditions, producing bounding planes for the data. The movement is broken into stages of airborne and landing, in which the character re-positions for the designated landing strategy, and executes the landing strategy respectively. Each of these is separated into impact, roll, and get-up stages. Movement and joint positions are produced using PID servos [2]. Other work on producing such controllers was produced by Faloutsos et al. who described a method of composing such controllers by giving pre-conditions, post-conditions, and intermediate state requirements. The composed controllers are then chosen at each step based on the current pose and which controller is deemed most suitable [3]. Hodgins et al. created several controllers for running, vaulting, and bicycling, creating realistic motions and secondary motion using rigid bodies and spring-mass simulations [5]. Geijtenbeek and Pronost provide a detailed review of physics based simulations [6].

CHAPTER 3
ANIMATION

Jumping is the acceleration of a character's center of mass upward. This motion can be divided into several stages. First is the lead-up or wind-up stage in which the character flexes or gathers momentum to perform the jump. This takes the form of a slight crouch (TODO reference cat jumping paper or the background section. should this be in background?) which prepares the character to exert the necessary force against the ground.

Next comes the take-off stage. The character pushes against the floor with their feet, accelerating their center of mass to break contact with the floor.

3.1 Path Estimation

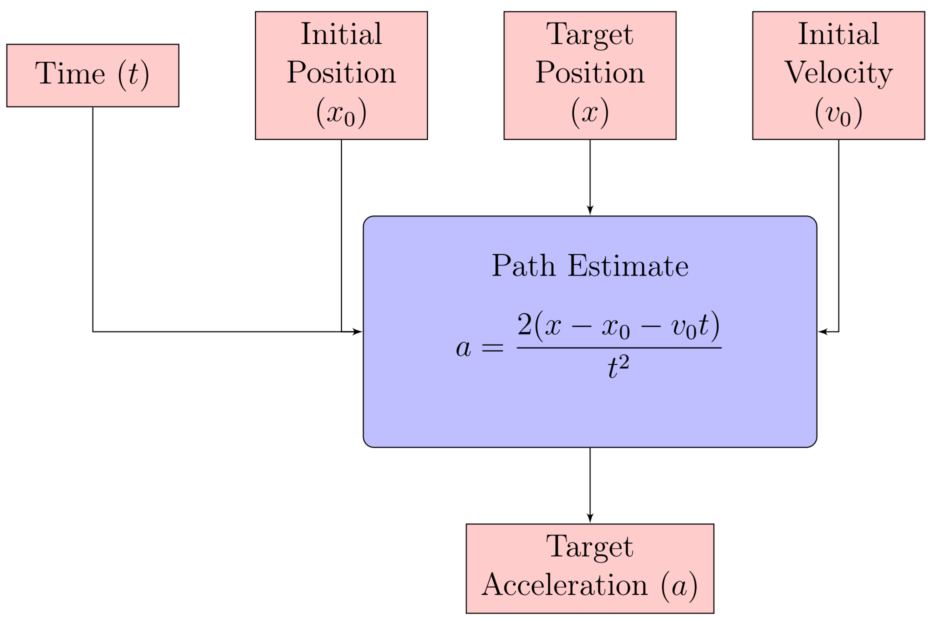

Figure 3.1: Diagram of the path estimation step.

Figure 3.2: Example of a path estimation.

Before calculations relating to the model's skeleton are performed, an initial estimate of the jump path is performed. The estimate uses a simple forward kinematic calculation to determine the force required to move an object through the air from the initial position of the model, denoted as x_0 in Figure 3.1, to a final position, denoted as x. To facilitate a character jumping while moving, the path estimate takes into account the initial velocity v_0.

The user specifies a desired time (t), which indicates the time the character will spend airborne during the animation, i.e. the time between when the character's feet break contact with the ground and when they regain contact with the ground. This is useful as the desired animation can be more easily adjusted to fit a desired time as an in-game animation or to fit a particular storyboard for an animated film sequence.

3.2 Windup

From this force, the acceleration can be determined using $F = ma$ from classical mechanics. This assumes the character is a rigid body with negligible air resistance acted upon by gravity of $10\frac{m}{s}$. The mass is calculated as the summed total of the distributed masses assigned to the character's limbs, which are summed to produce a total mass for the character.

Proportional derivative control is used to produce the windup motion once the initial path estimate is calculated. The error function E_{all} calculates error from the desired force E_{force} as well as the balance $E_{balance}$.

Once computed, error is compared to a threshold (ϵ). If the error is below the threshold, the skeleton is considered bent to the proper position for windup and the system proceeds to the thrust phase in which the character unbends. If the value is above the threshold, a new position for the hip is calculated using proportional-derivative control, where the new position for the iteration of the controller, $u(i)$, is calculated as

$$u(i) = k_p E_{all}(i-1) + k_d(E_{all}(i-1) - E_{all}(i-2))$$

where i is the iteration, and k_p and k_d are weights which determine the rate of change.

This new hip position is given to an inverse kinematics component to calculate the positions of the remaining leg joints, assuming the feet should remain in the same position. These new joint positions and angles are then passed back to the PD controller to re-calculate the center of mass as well as the new force and balance errors for the next iteration.

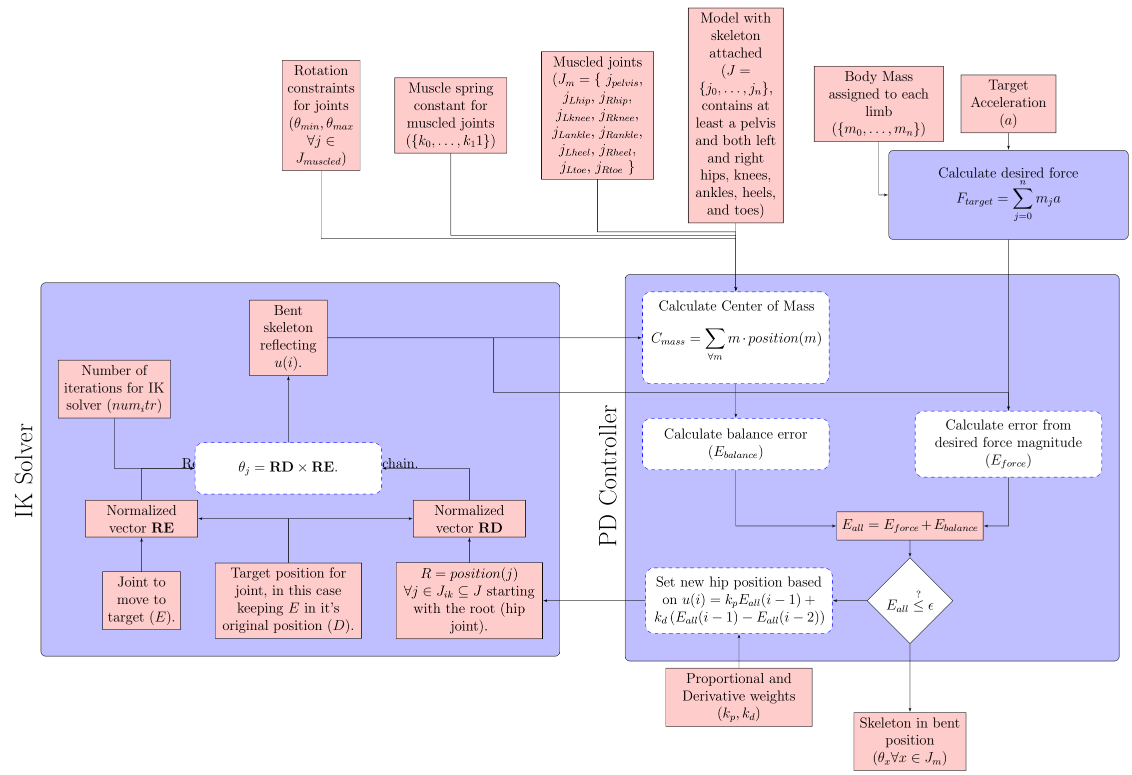

Figure 3.3: Algorithm diagram of the windup phase.

Table 3.1: Example values for the PD controller constants, showing the sweet spot that we use as well as the effect of going higher or lower (how do we show this effect, show that the steps are too large or too small? time or iterations to finish?).

3.2.1 Force calculation

Current force is computed by approximating muscles as linear springs attached to two bones and crossing a joint. The change in spring length is produced by the bend of the joint which opens a space between the rigid bones that the spring must

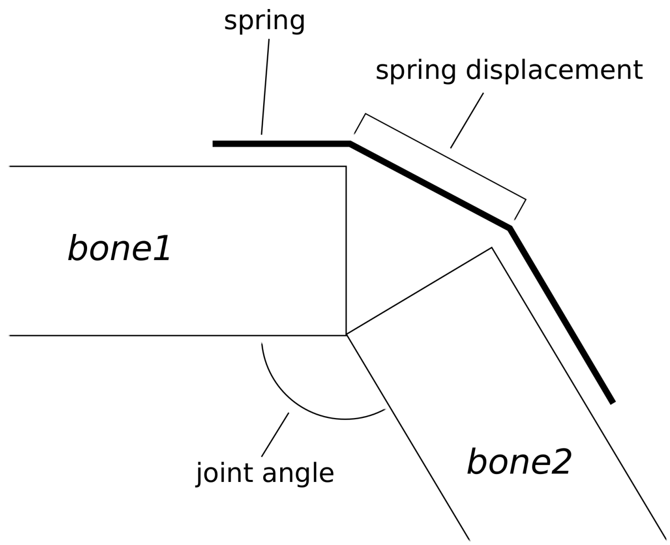

Figure 3.4: Force calculation for a joint.

stretch across. This approximation uses spring constants that simulate a flexed muscle that has been stretched by an amount equal to

$$\theta = cos^{-1}\left(\frac{2k^2r^2}{F^2} - 1\right)$$

which requires a more rigid spring constant. In this equation, r, represents the width of the bone, which is a constant that changes the amount the character must bend to achieve a force. This acts similarly to the k value, which represents the stiffness of the spring, or a constant for the linear restoring force of the muscle.

Table 3.2: A table of various values for k and r, demonstrating effect on the model's bend. TODO Columns of k and r (varied individually), and image of fully bent character resulting from values.

At each iteration of the PD controller, current force output of the legs is computed. An accurate calculation of the force must take into account the torques (τ) produced by the muscles on each joint. We use a simplified version, calculating the magnitudes of the torques which avoids the complexity of implementation and computation. This is found for a joint j as

$$\tau_j = ||\mathbf{r_j}||F_j$$

where r is the moment arm, the cross product between the vector between the attachment point of muscle and the center of the joint with the direction of the muscle crossing the joint. Direction of a muscle is in the direction of restoration of the spring (TODO double check this, is this always the value we want, point placed along the spring based on the masses on each end) as used in [1].

Figure 3.5: Force error calculation

To calculate the error used for PD control, the vector between the two forces is measured, indicating the change required in both direction and magnitude. As we require the force to act in the direction of acceleration, the main adjustment is the force magnitude. When instead considering the torque magnitude, the magnitude alone can be considered. (TODO I think I need to be using the torque magnitude instead of a force here).

3.2.2 Center of Mass and Balance

The center of mass (CoM) is calculated as the centroid of the character. More specifically, joint positions are averaged, with a weight assigned to each joint based on the weight of the limb associated. The CoM must be recalculated with each update to the character's pose as the shift in weight changes the position.

Using the calculated CoM, the balance of the character can be determined by the position of the CoM relative to the supporting polygon of the character. The supporting polygon is a polygon determined by the points of contact of the character with the ground or other supports which provide a normal force to counteract gravity and other external forces. During the windup and thrust phases, the character maintains contact with the ground through their feet, with the outer edges of the feet forming two sides of a quad, a line between the two feet at the toes forming a third, and a line between the heels of the character forming the fourth side. This polygon should be parallel to the ground plane, and is positioned at the bottom of the feet. If the character's center of mass is over this supporting polygon, the character is balanced.

Figure 3.6: Balance error calculation.

To quantify balance, the vector between the center of the supporting polygon and the position of the CoM is measured. This vector is then projected into the same plane as the supporting polygon, giving a 2 dimensional error between where the CoM is currently and where it would need to be to be perfectly centered. The PD controller then attempts to minimize the magnitude of this vector by moving in the proscribed direction while bending to achieve the desired force, constraining the number of solutions possible to provide the desired force.

3.2.3 Inverse Kinematic Solving

As the skeleton is a hierarchy assumed to be rooted at the hip, a problem arises with applying rotations to joints. To keep a character's feet rooted to the floor as is expected, positions must be solved for using inverse kinematics. Given the desired position for the hip, and the desired position of the foot, the joint angles and positions of the knee and ankle are solved for. Constraints are placed on each joint, limiting the range of motion to an expected range as well as limiting the axes about which each joint can rotate, preventing unnatural directions of motion. These values are specified per joint and can be edited by the user to simulate varied levels of flexibility or alternate body shapes.

Table 3.3: Joint angle constraint values used for each joint, with accompanying images of expected motion range.

A solution to the joint positions is found using these constraints, and gradient descent method which works on single-chains of joints. A single chain of joints is a sequence of joints in which each joint has a single child and a single parent, with one root joint and one leaf joint which lack a parent and child respectively. Given the hierarchy of joints and a desired position for one of the non-root nodes of the chain, cyclic-coordinate descent (CCD) is used to determine rotations of the joints between the joint in question and the root that will minimize the distance between the joint in question and the desired position. This algorithm is shown in 3.1.

Algorithm 3.1: Given chain of joints C, move joint E to position D using cyclic coordinate descent (TODO cite CCD references). This process iteratively moves joint E closer to the location D, concentrating on each joint R in the chain one at a time and solving the geometric problem of minimizing distance between E and D by rotating R.

```
function SingleChainIK(C, D, E)
    repeat
        for all Joints R between E and the root, starting with the end E do
            θ_R = RD × RE
        end for
    until Desired number of iterations performed or E is close enough to D
end function
```

This approach, while for the specific case of single chains of joints, is simpler to implement than other approaches such as the pseudo-inverse of the Jacobian. In addition, this approach allows some flexibility, specifically in constraints of the joints. As each joint is addressed individually instead of the system as a whole, any constraints placed on the joint can easily be accounted for by simply preventing the joint from rotating out of the desired range while the rest of the system continues to move as close as possible to the solution. One downside is that a halting condition must be determined, through a minimum acceptable distance. To handle the case where the joint cannot be moved within this minimum distance, a maximum number of iterations must be designated. In practice, 100 iterations is enough to converge, with as few as 30 working well for our simulation.

3.3 Thrust and Takeoff

Upward acceleration is animated by calculating the angular accelerations of the joints given the forces acting upon them and the resulting torques. Torque is the change in angular momentum over time, allowing for the acceleration to be calculated as the mass can be assumed to be constant:

$$\tau = \frac{dL}{dt} = m\frac{dv_\theta}{dt} = ma_\theta$$

where τ is the torque of a joint, L is the angular momentum, m is the mass of the limb, which must take into account the mass of the rest of the body which is also moved by the limb, v_θ is the angular velocity and a_θ is the angular acceleration. This results in the below equation for determining angular acceleration.

$$a_\theta = \frac{\tau}{m}$$

3.4 Summary

CHAPTER 4
VISUALIZATION

4.1 Motion Visualization

For visualizing motion of a character or figure, there are a limited selection of different techniques. Most common is a sequence of frames in which a character is posed, either in a still sequence or as a video. As this is a final goal of our system, this is a valid visualization, but fails to provide a simple comparison between one animated sequence and another. This is desirable for qualifying or quantifying performance of the system. A sequence of still images is also space-consuming, which can be undesirable for print formats or even digital formats where length or size of document is an issue.

Specific markers can be used to highlight motion of particular parts of the body, such as the pelvis or center of mass. Other indicators placed on or around the figure can indicate other values, such as arrows to represent vectors of force. This however can result in clutter within the images, scene, or frame of video, occluding or distraction from the primary animation.

4.2 Summary

CHAPTER 5
RESULTS

5.1 Images

5.2 Limitations

Specifying data can be work intensive, but gives freedom to make wide changes to the animation by tuning parameters

Limited secondary motion.

5.3 Summary

CHAPTER 6
FUTURE WORK AND CONCLUSION

- learning model - slight adjustments to different situations can be more easily generated through the use of a learning model such as in XYZ [1, 2].
- upper body affects, non-negligible effect of upper body movements on acceleration of CoM
- expand to non-bipedal
- more complex muscle and motion model, trials to determine level of effect on simulation (is it worthwhile)
- in-air phase: add complexity, can we jump to create vaulting motions, mid-jump push off from objects (jump and push off a wall), rotations, or flips.

6.1 Conclusion

References

[1] T. Geijtenbeek, M. van de Panne, and A. F. van der Stappen, "Flexible muscle-based locomotion for bipedal creatures," *ACM Transactions on Graphics*, vol. 32, no. 6, 2013.

[2] S. Ha, Y. Ye, and C. K. Liu, "Falling and landing motion control for character animation," *ACM Trans. Graph.*, vol. 31, no. 6, pp. 155:1–155:9, Nov. 2012. [Online]. Available: http://doi.acm.org/10.1145/2366145.2366174

[3] P. Faloutsos, M. van de Panne, and D. Terzopoulos, "Composable controllers for physics-based character animation," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 251–260. [Online]. Available: http://doi.acm.org/10.1145/383259.383287

[4] Y. Koga, K. Kondo, J. Kuffner, and J.-C. Latombe, "Planning motions with intentions," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '94. New York, NY, USA: ACM, 1994, pp. 395–408. [Online]. Available: http://doi.acm.org/10.1145/192161.192266

[5] J. K. Hodgins, W. L. Wooten, D. C. Brogan, and J. F. O'Brien, "Animating human athletics," in *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '95. New York, NY, USA: ACM, 1995, pp. 71–78. [Online]. Available: http://doi.acm.org/10.1145/218380.218414

[6] T. Geijtenbeek and N. Pronost, "Interactive character animation using simulated physics: A state-of-the-art review," *Computer Graphics Forum*, vol. 31, no. 8, pp. 2492–2515, 2012. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2012.03189.x