UNIVERSITY OF CALGARY

Physically-based animation of plant motions

by

Alejandro Garcia

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

June, 2023

© Alejandro Garcia 2023

Abstract

The creation of realistic and lifelike plants has been a long-standing challenge in computer graphics. While significant progress has been made regarding the generation of plants using procedural methods, there is still a gap in understanding how to simulate their dynamics as efficiently and realistically as possible. One of the major challenges in this area is the incorporation of complicated non-inertial effects into plant motion. Previous works tend to either focus on quasistatic simulations - which by definition assume the absence of non-inertial effects - or ignore secondary motion in their dynamics calculations altogether. Either of these result in incomplete simulations that do not adequately capture the wide range of plant motion observed in nature. This is important because the human eye is keenly critical of inconsistencies in motion, meaning that incomplete models can easily appear off-putting and uncanny. To address these limitations, this thesis proposes a generalized and comprehensive physics model that aims to better capture the dynamics of procedurally-generated plants.

Acknowledgements

I would like to express my deepest appreciation and gratitude to my supervisor, Dr. Prusinkiewicz, for his invaluable guidance, support, and unwavering patience throughout my prolonged journey in completing this master's thesis.

Additionally, I would like to give a heartfelt thanks to my family and friends for their much needed company throughout.

To Dad

Table of Contents

Abstract **ii**

Acknowledgements **iii**

Dedication **iv**

Table of Contents **v**

List of Figures and Illustrations **viii**

1 Introduction **1**
- 1.1 Methodology outline . . . 2
- 1.2 Contributions . . . 4
- 1.3 Chapter outline . . . 4

2 Previous work **6**
- 2.1 Previous work on plant modeling . . . 8
  - 2.1.1 Reconstruction methods . . . 8
  - 2.1.2 Interactive modeling methods . . . 8
  - 2.1.3 Procedural or rule-based methods . . . 10
- 2.2 Previous work on plant animations . . . 12
  - 2.2.1 Data-driven methods . . . 14
  - 2.2.2 Physically-based methods . . . 16
  - 2.2.3 Hybrid methods . . . 22
- 2.3 Previous works that couple plant modeling and animation . . . 23

3 Physics background **28**
- 3.1 Contents of the chapter . . . 28
- 3.2 Coordinate systems . . . 28
  - 3.2.1 Transformation matrices . . . 30
- 3.3 Rigid body dynamics . . . 32
  - 3.3.1 Definition of a rigid body . . . 32
  - 3.3.2 Newton-Euler equations for a rigid body . . . 36

4 The Articulated-Body Algorithm **41**
4.1 Chapter overview . . . 41
4.2 Classification of rigid-body dynamics algorithms . . . 43
4.2.1 Maximal coordinate vs generalized coordinates . . . 44
4.2.2 Forward vs inverse dynamics . . . 48
4.2.3 Kinematic trees vs closed-loop systems . . . 50
4.2.4 Classicifation of the articulated-body algorithm . . . 50
4.3 Formal description of an articulated body . . . 51
4.3.1 Parent and children bodies . . . 52
4.3.2 Indexing and connectivity . . . 53
4.3.3 Body-fixed frames . . . 53
4.3.4 Dynamic state of an articulated body . . . 54
4.4 The articulated-body algorithm . . . 55
4.4.1 First pass: forward propagation of velocities and accelerations . . . 57
4.4.2 Spatial Algebra . . . 64
4.4.3 Second pass: the articulated-body equations of motion . . . 70
4.4.4 Third pass: solving for accelerations . . . 77
4.4.5 Final algorithm . . . 77

5 L-Systems **79**
5.1 L-systems . . . 79
5.1.1 0L-systems . . . 80
5.1.2 Parametric 0L-systems . . . 81
5.1.3 Turtle interpretation of L-system strings . . . 83
5.1.4 Bracketed 0L-systems . . . 85
5.2 Articulated-body L-systems . . . 85
5.2.1 Structure of articulated-body L-system strings . . . 87
5.2.2 Parameters of rigid-body and joint modules . . . 89
5.2.3 Note on small rigid bodies . . . 93

6 Representing continuous branching structures with articulated bodies **94**
6.1 Summary of deformation paradigm employed . . . 95
6.2 Deformation types . . . 97
6.3 Continuous pure bending . . . 99
6.3.1 Discretized pure bending . . . 102
6.4 Continuous torsion . . . 104
6.4.1 A note on the values used for the shear modulus of elasticity . . . 106
6.4.2 Discretized torsion . . . 107
6.5 Infinitesimal rotations . . . 110
6.6 Damping . . . 111

7 System Overview **115**
7.1 RBDL . . . 115
7.1.1 6-D Vector Algebra Library . . . 116
7.1.2 The `PhysicsModel` data structure . . . 116

7.1.3 The PhysicsModel::ForwardDynamics() method . . . 117
7.2 LSDL . . . 118
7.2.1 The PlantModel::PlantModel(string lsystem_string) method . . 119
7.2.2 The PlantModel::PhysicsStep(float dt) method . . . 121
7.3 Example applications using LSDL . . . 122
7.3.1 OpenGL application . . . 122
7.3.2 Unreal Engine 5 application . . . 124

8 Validations **127**
8.1 The physical pendulum . . . 127
8.2 Double physical pendulum . . . 130
8.3 Cantilever beam . . . 134

9 Results **138**
9.1 Plant simulations - monopodial structures . . . 138
9.1.1 Example: gravity . . . 141
9.1.2 Example: moving base . . . 142
9.1.3 Example: pulling . . . 143
9.1.4 Example: modifying elasticity . . . 144
9.1.5 Example : motion comparison against a real monopodial plant . . . 145
9.2 Plant simulations - structures with higher-order branches . . . 148
9.2.1 Example: twisting motion . . . 149
9.2.2 Example: the importance of shape . . . 152
9.2.3 Example : motion comparison against a real branching plant . . . 152
9.3 Model parameters . . . 155
9.4 Accuracy and efficiency . . . 156
9.4.1 Accuracy . . . 157
9.4.2 Efficiency . . . 160

10 Conclusion and future work **167**
10.1 Future work: accuracy of the model . . . 168
10.2 Future work: efficiency . . . 170
10.3 Future work: coupling of growth and dynamics . . . 171

Bibliography **174**

A Equations of motion for the double physical pendulum **186**

List of Figures and Illustrations

2.1 Spectrum of plant animation methods 13
2.2 A triangle classifying previous work in plant animations 13

4.1 A 2-D double compound pendulum 41
4.2 Classification of different kinematic constraints 47
4.3 Let's take b_1 above into consideration. Its inboard joint is j_1, and its outboard joints are j_2 and j_3. Its parent is the root body b_0 (not pictured), and its children are b_2 and b_3. 52
4.4 A rigid body experiencing simple linear motion. 55
4.5 A linear force not about the center of mass results in a non-trivial system. . 56
4.6 A linear force not about the center of mass plus a torque about a different point results in a complicated articulated-body system. 56
4.7 Necessary geometric quantities between two bodies. The diagram hints towards a revolute joint, but the same quantities apply for a prismatic joint. . 60

5.1 The local reference frames of rigid bodies and joints. 91

6.1 The local reference frames of rigid bodies and joints. 95
6.2 A spherical joint j_i and its local coordinate frame. Each spherical joint will be approximated as three perpendicular revolute joints each acting about one of the axes. Each revolute joint will have an angular spring associated with it: $s_i^{\hat{\mathbf{R}}}$, $s_i^{\hat{\mathbf{H}}}$, and $s_i^{\hat{\mathbf{U}}}$. 96
6.3 Rotations at the joints being used to represent the deformation of a continuous body 98
6.4 Types of elastic deformation 98
6.5 A bent internode experiences tension and compression. 101
6.6 I-beams are universal in construction and engineering because they maximize the second moment of area whilst minimizing cross-sectional surface area. This means that they are relative sturdy for their low material costs. 101
6.7 Calculating the discrete curvature K_i between two rigid bodies. 102

8.1 Free body diagram of a 2D physical pendulum 128
8.2 Snapshots of two single physical pendulums simulated side by side. 130
8.3 A 2-D double physical pendulum 131
8.4 Two double physical pendulums side by side. Notice the chaotic motion starting at the 4th frame. 132

8.5 A cantilever beam of length l supported on its left end. 134
8.6 A cantilever beam simulation with 25 bodies. 137

9.1 A monopodial plant structure modeled via an L-system. 138
9.2 (top): A leaf being approximated by a single cylindrical internode. (bottom): A leaf represented by several cylindrical internodes. 139
9.3 (left) The plant that is rendered versus (right) the actual appearance of its articulated-body. 140
9.4 (left): The rest pose of a broad-leaf plant experiencing normal gravity. (right): The rest poses of the same plant if it were to experience the 5x force of gravity in other directions. 141
9.5 Simulating the movement of a potted plant by only accelerating its base. (left column): rightwards and then leftwards. (middle column): forwards and backwards. (right column): counter-clockwise motion. 142
9.6 (top row): progressively pulling a small herbaceous plant near at its tip. (bottom row): the resulting motion after releasing. 144
9.7 A rose with reduced Young's modulus falling under its own weight. 145
9.8 Side and top views of a tropical plant. 146
9.9 Comparison of real and synthesized plant motion. The plant is first pulled (frames 1-2) and released (frame 3). 147
9.10 Growth time stamps of an inflorescent plant. The yellow spheres indicate flower placement and size. 148
9.11 Many flowering plants can be modeled as a vegetative part (blue square) and a flowering part (red square). 149
9.12 (top row): twisting a plant in incremental amounts. (bottom row): top-down view of the same motion, the twist amount is represented by the angle between the dashed line and solid line. 150
9.13 Snapshots showing the oscillatory motion of a twisted plant at the time of release. 151
9.14 (left): An inflorescent plant with exaggerated laterally-extending branches at rest. (middle): The plant's main stem is twisted in the direction of the arrow. (right): The plant's curved appearance shortly after the release of the external torques; the arrows denote instantaneous direction of motion. 151
9.15 (top): Three plants experiencing equal bending torques. The radii of the plants vary ascendingly from left to right, but the plants are otherwise identical. (bottom): The plants are again experiencing equal bending torques. The lengths of the plants vary ascendingly from left to right, but the plants are otherwise identical. 153
9.16 Side and top views of the orchid. 154
9.17 Comparison of real and synthesized orchid plant motion. 154
9.18 A table showing performance metrics of several simulations presented in this chapter. 166

10.1 My methodology treats branching points as independent segments (left), whereas in reality, plants exhibit unique geometry at the branching points (right). . . 169

10.2 A state diagram illustrating the methodology employed in this thesis. Growth can influence dynamics, but dynamics cannot influence growth. 172
10.3 A more elegant solution would account for bi-directional flow between the growth model and dynamics model 173
10.4 The most elegant solution would incorporate dynamics as a part of its growth model altogether. 173

A.1 A 2D double physical pendulum 186
A.2 Free body diagram of a double physical pendulum 187

Chapter 1

Introduction

This is an exploratory project whose aim is to combine known methods for procedurally generating plants with known methods for procedurally animating them. Previous works in this field have tackled the problem of finding out the growth response that a plant may exhibit due to its weight, light and/or space availability, or even exposure to wind fields. However, the physics models in these works can be expanded upon; some works are limited to quasistatic behaviour, whereas others explicitly ignore secondary motion in their dynamics calculations. To address these limitations, this thesis aims to develop a more complete model that better captures the dynamics of procedurally-generated plants, including non-inertial accelerations.

The goal of this research is to develop a more comprehensive and accurate model of plant behavior that can simulate a wide range of plant movements and responses to environmental factors. It has the potential to advance the field of computer graphics by enabling the creation of more realistic and dynamic plant simulations, which could have applications in fields such as video game design, movie production, and virtual reality environments. Additionally, this research could contribute to a better understanding of simulating plant biomechanics, which could have positive uses in fields such as agriculture and horticulture.

1.1 Methodology outline

There have been numerous advances in the field of *developmental plant modeling* in the previous decades. The field is interested in using computers for the modeling, simulation, and visualization of plants as they develop over time. Developmental models consist of procedural systems that create spatial structures and consider these structures as they develop over time. However, most works in developmental plant modeling are (understandably) only concerned with the creation of the spatial structure itself, and the research stops there. This means that we do not get to see nor analyze how the end-result plant reacts to environmental forces (human interaction, wind forces, etc.). This thesis explores this gap by exploring the dynamics problem of procedurally generated plants, that is, finding the accelerations response of the plant due to applied input forces.

The methodology employed in this thesis for simulating plant motion is rather general, meaning that it can be adapted to many existing works in developmental plant modeling. However, I will only focus on animating plants generated using *Lindenmayer systems*.

Lindenmayer systems, or L-systems for short, are *parallel re-writing systems* and a type of formal grammar, and they excel at representing developmental models of plants. Starting with an initial string called the *axiom*, an L-system evolves in accordance to a given set of *production rules* that define how each letter in the axiom evolves and produces other letters over time. These letters, called *modules*, can each correspond to different organs of the plant, such as internodes, leaves, or flowers. The structure of the plant is thus wholly represented by the modules composing its L-system string at any point in time. The question that proceeds is: *how do we solve the dynamics problem of plant models represented by this structure?*.

There are several candidate methodologies that can been used to solve the dynamics of L-system models (an overview is provided in Chapter 2), but ultimately, we chose to pursue a

method derived from rigid-body dynamics. The reason for this is because Dr. Prusinkiewicz and I deemed it logical to represent each organ of the plant as a rigid body, and to connect these rigid-bodies with three-dimensional angular springs. The job of the angular springs is to resist rotation between successive rigid bodies, and, once rotated, to introduce restoration forces that return the rigid bodies to their rest orientations, causing the illusion of elasticity. This composition of rigid-bodies and joints is called an *articualted body.* With this physics model at hand, we can use one of several articulated-body dynamics algorithms to calculate the acceleration response of the plant due to abstract input forces. It is at that point that a time integration scheme may be used to advance the system forwards in time, yielding an interactive environment of plant motion.

The rigid-body dynamics algorithm used in this thesis is Featherstone's *articulated-body algorithm* [21]. Assuming that the input forces are provided, the articulated-body algorithm computes the acceleration of each rigid body in the system in $O(n)$ time, where n is the number of degrees of freedom of the plant.

It is worthwhile to note that despite serving distinct purposes, the articulated-body algorithm and L-systems are eerily alike in how they operate; both are based on the idea of *locality*, meaning that any element of the L-system string/articulated-body is only aware of the existence of itself and those attached to it. Moreover, both employ a recursive mechanism with which to transfer information throughout the system. Indeed, a key reason we pursued to employ the articulated-body algorithm was due to its theoretical similarities with L-systems.

1.2 Contributions

This thesis does not contribute a 'new way' of doing things; it simply explores the combination a proven method for modeling plants (L-systems) with a promising method for animating their dynamics (the articulated-body algorithm). Additionally, the work in this thesis aims to to verify how useful and appropriate it is to use Featherstone's articulated-body algorithm for animating accurate plant motions in real-time.

1.3 Chapter outline

The remaining chapters are arranged as follows:

- **Chapter 2**: Provides a short survey on physically-based plant modeling and animation literature.
- **Chapter 3**: Covers preliminary physics background information, setting up the stage for chapter 4.
- **Chapter 4**: Provides a general overview of articulalted-body dynamics algorithms, points out where Featherstone's *articulated-body algorithm* lives in this classification, and derives the articulated-body algorithm from mechanical principles.
- **Chapter 5**: Quickly goes over the basics of plant modeling with L-systems, and introduces the notion of 'Articulated-body L-systems', which are L-systems modeled in such a way that they explicitly represent developmental models of simulation-ready articulated-bodies.
- **Chapter 6**: From bio-mechanical principles, discusses the methodology that was employed in this thesis to make our articulated-body L-system plants move as lifelike as possible.

- **Chapter 7**: Provides an overview of the programming components completed in this thesis.
- **Chapter 8**: Goes over several test cases that mathematically support the credibility of the animations.
- **Chapter 9**: Presents many example simulations showcasing plants reacting to different user interactions, as well as comparisons of synthesized plant motion against real plants.
- **Chapter 10**: Concludes the contributions of the thesis and provides a list of research questions that remain unsolved.

Chapter 2

Previous work

The work described in this thesis belongs to two main research topics in computer graphics: *plant modeling* and *plant animation*. Plant modeling refers the general process of creating virtual representations of plants that can be rendered to a screen. This is a vague definition at best, and there exist a plethora of approaches in which one can achieve this. Plant animation is a subtopic within plant modeling, and in the context of this thesis, will be defined to be the art of adding *mechanical motion* to said plant models. Mechanical motion is motion that emerges from the addition of energy or application of forces to the mechanical system at hand.

Mechanical motion may be *predefined*, which is manually created by an artist, or *procedural*, which is computationally generated on the spot. The previous works shown in this chapter - as well as the contents presented in this thesis - will only tackle procedural motion. Furthermore, there will be an emphasis on *physically-based* plant modeling and animations, which is the art of constructing and animating models of plants such that their shape and motion conform to the laws of nature.

This chapter explores relevant previous works in physically-based plant modeling and ani-

mations, and is organized as follows:

- **Section (2.1)**: Previous works on plant modeling
- **Section (2.2)**: Previous works on plant animations
- **Section (2.3)**: Previous works that couple plant modeling and animations

The difference between the second and third criteria points above can best be understood by using the definitions of *developmental* and *non-developmental* models of plants:

> **Definition:** *A developmental model is a procedural system that creates a spatial structure, and considers this structure as it develops over time. A non-developmental model is one that is not developmental.*

The works presented in Section (2.2) can be said to animate non-developmental plant models because they first create static geometric plant model and *then* add mechanical motion to it. This is in contrast to the works presented in Section (2.3) because their method allows the plants to experience and be affected by mechanical motion *during* their formation; they animate developmental plant models. This is an important distinction because plants are living, sensing organisms; their development is heavily influenced by environmental stimuli, and as such, a model that does not capture this behaviour cannot be fully 'complete'. Having said this, it is not at all trivial to add mechanical motion to developmental models of plants, which is likely the reason why many works fall under the first two criteria whereas only a handful belong to third.

The work in this thesis aims to take steps towards a unified physically-based developmental plant model such that one can add versatile mechanical motion (e.g. wind motion, user interaction, collisions and contacts, gravitational effects, etc.) to developmental models of plants, but ultimately, still belongs in Section (2.2) because **I create a model of a plant and animate it after the fact.**

2.1 Previous work on plant modeling

Plant modeling approaches can be categorized into three principal classes: reconstructions from existing real world data, interactive modeling methods, and procedural or rule-based systems.

2.1.1 Reconstruction methods

Reconstruction methods consist of taking data from real-world plants and using them to virtually create the geometric plant model. They are also known as data-driven modelling methods. Common approaches involve the use of individual photographs [89, 88], sets of photographs taken from multiple points of view [78, 73, 33], and recently, point clouds obtained from laser scans [97, 100, 16].

Reconstruction methods are ideal for creating models of plants whose shape is geometrically accurate, however, the method's main drawback is its inability to produce a wide variety of models. To address this limitation, Shlyakhter *et al.* [84] extract tree silhouettes from images and reconstruct the main tree skeleton, and from there, use L-systems to procedurally grow the small branches and foliage. Neubert *et al.* used particle flow to extract branch structure from the recovered tree volume. Stava *et al.* [86] combine the reconstruction and procedural approaches by taking polygonal tree models as inputs - which were themselves reconstructed from images - and use them to estimate the parameters of a procedural model such that it would produce trees similar to the input.

2.1.2 Interactive modeling methods

Interactive modeling methods aim to enhance an user's modeling experience by minimizing the effort required to produce visually plausible models *without* imposing excessive limita-

tions on the user. Ideally, a decent interactive modeling software will give a beginner the ability to create realistic models, without limiting the capabilities of an expert.

Early ideas in interactive plant modeling date back to Reeves and Blau in 1985 [75], where they let an user specify a surface of revolution that would then define the overall silhouette of a tree. Weber and Penn [94] furthered user interactability by exposing ample control over a tree's architectural parameters. Deussen and Lintermann [19, 48] introduced a method in which the user models in 'components'; each component represents a different geometric construct, which the user can then arrange in different configurations to create unique botanical structures . Motivated by this idea, Prusinkiewicz *et al.* [70] explored a visual way in which to procedurally model plants, which was done by allowing users interactive control over the local attributes of plant architecture as functions of their location along the principal stems of the structure.

Several previously cited models employ a *global-to-local* methodology, which means that users are in charge of defining the overall structure, whereas the model is in charge of generating the details. Boudon *et al.* [15] for example adapted such a paradigm by allowing users to define the silhouette of the structure, allowing for detailed constructs to be inferred procedurally thereon. Advancements in touch interface technology meant for the immediate incorporation of this technique into such devices [96, 49]. Longay *et al.'s* TreeSketch [49] is notable because it integrates procedural tree generation with a multi-touch tablet interface that provides detailed control of tree form. The procedural component is used to simulate the competition of branches for space and light as the tree develops over time, and is based on a culmination of previous works on *self-organization* [91, 3, 52, 77, 62].

2.1.3 Procedural or rule-based methods

Procedural plant modeling can be traced back to the early tree modeling techniques presented in Ulam 1962 [91] and Honda 1971 [37]. Honda modeled trees by defining a few parameters that would then be used to describe an explicitly defined recursive structure, whereas Ulam considered trees as *self-organizing* structures; the branching structure was not explicitly defined, but rather emerged through the competition for space of the individual elements of the structure.

Honda's method has led to further development in explicit recursive generative algorithms in plant modeling. Aono and Kunii [2], as well as Bloomenthal [13], explored more realistic generation of large trees. Reeves presented an early example of using these recursive methods for the generation of forests in early 3D animations [75]. Oppenheimer tackled the problem of the 'jarring similary' that oftentimes presented itself in explicitly recursive models, and did so by exploring randomness and the bending and twisting of branches [59]. For a similar purpose, Weber and Penn [94] explored ample control over parameters to give users further control of the structure. Recursive methods were also the basis to some previously mentioned interactive modeling methods [48, 71].

Likewise, Ulam's self-organizing method was also furthered by researchers in plant modeling. Měch and Prusinkiewicz introduced *Open L-systems*, which as opposed to regular *L-systems*, have the ability to communicate with its local environment [52]. Runions et al. used *space colonization* algorithms to create 2D venation patterns on leaves [76] and 3D branching structures in trees [77], the latter of which was extended by Palubicki et al. [62]. Some of the previously mentioned sketch-based interactive methods were actually primarily dependent on space-colonization methods [49, 35], with the principal idea being that the user can sketch to define the 'available space' that the plants can grow into.

Certain methods even employ a mixture of a self-simiar and self-organiszing approach. Makowski et al. [50] tackle the synthesis of plant ecosystems, and describe their plant models as “deterministic at the scale of architectural units and self-organizing at the scale of the whole plant”. Such methodology proved them successful in the creation of developmental plant models across differing ecosystems.

Developmental models of plants using L-systems

L-systems are parallel rewriting systems introduced by Lindenmayer in 1968 [46] with the goal of describing the development of simple (filamentous) multicellular organisms over time. Since then, research in L-systems has evolved to the point that they can be used to describe many facets of the development of three-dimensional branching structures.

Chapter 5 goes over the details, but for now, it suffices to know that **L-systems were chosen as the methodology for generating plant structures in this thesis** because they excel at namely two things amongst others:

- Modeling developmental models of plants.
- Providing an easy-to-parse string representation of the object at an instance in time.

Having easy to parse representation of developmental models of plants is relevant because they greatly facilitate the process of creating a physical model from the developmental one. In our case, all L-system objects are comprised of *modules* which can be arranged linearly like a string. Each module may represent an individual component - such as an individual organ - of the plant being modeled. For example, the string

`I I F`

could denote two internodes `I I` followed by a flower `F`. This readily-available discrete description of the developing plant will be a convenient first step towards their rigid-body

interpretations.

2.2 Previous work on plant animations

The real-life motion of a plant can most generally be described through the concepts of *stresses* and *strains*. Stress is a quantity that represents the internal forces that a plant perceives due to applied external forces. For the context of this thesis, we will only be modeling *elastic* strains, which means that strain is proportional to stress and can be defined as the ratio of deformation of the material due to the applied stress(es) [55]. External forces may either be *surface forces* or *body forces.* Surface forces act on the external boundary of the plant, such as those from wind and water droplets, whereas body forces are those that act on the volume of the plant, such as gravity. Once a plant has been deformed from its rest state, the internal binding forces of the plant, called *restoration forces*, will attempt to restore the plant to its original shape.

It is this push and pull between external and internal forces that causes the oscillatory motion of plants that we see every day. It is only in the past few decades, however, that computers have given computer scientists the ability to synthesize this motion.

There are two general animating techniques that are used to synthesize the motion of a plant: *data-driven* and *physically-based* techniques. Data-driven methods are those in which the synthesized plant motion predominantly originates from pre-obtained data, whereas physically-based methods are those in which the synthesized plant motion predominantly originates from physical simulations. **This thesis employs a physically-based approach**. It's true that empirical data (e.g. Young's modulus, bulk tissue densities) is needed to guide a physically-based method, but ultimately, the motion itself is dictated by numerically inte-

grating some type of equations of motion forward in time.

There are various works in plant animation literature that *strongly* follow either a data-driven or a physically-based approach, however, from observation, most employ a non-obvious mixture of both techniques. It may therefore be appropriate to say that works in plant animations belong in a spectrum between data-driven methods and physically-based methods, with those towards the centre being classified as *hybrid methods* (Figure 2.1).

Figure 2.1: Spectrum of plant animation methods

There is, however, a third criterion that we should take into account, and that is the extent to which the use of *heuristics* is involved. Heuristic approaches, in the context of plant animations, are those in which shortcuts were used - either to minimize complexity or maximize run-time speed - at a potential cost in physical accuracy. With this added criteria, a given previous work in plant animations can now be thought of as belonging somewhere inside the triangle depicted in Figure 2.2.

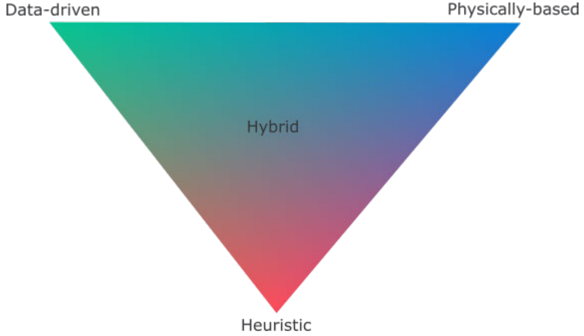
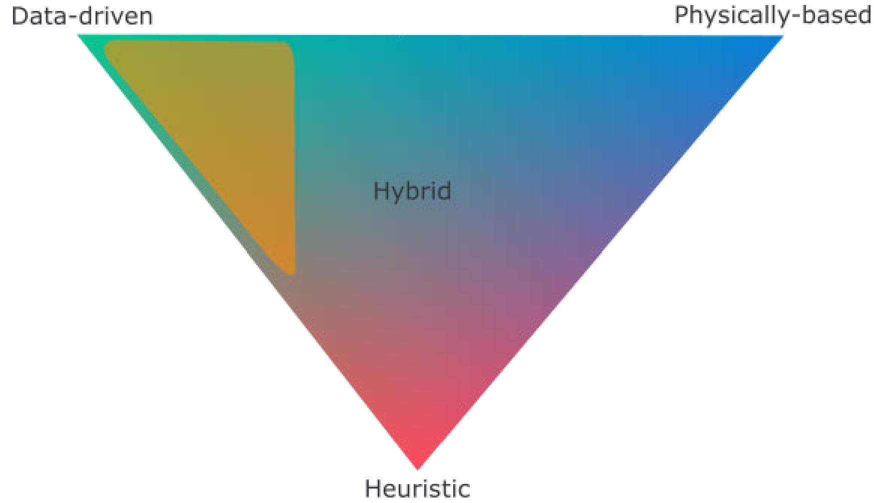


Figure 2.2: A triangle classifying previous work in plant animations

The rest of this section will be concerned with exploring data-driven, physically-based, and hybrid methods by introducing a handful of literature belonging to each classification.

2.2.1 Data-driven methods



In theory, an idealistic physically-based simulation of plants would be able to perfectly recreate real-life plant motion at real-life speeds. In reality however, such a model does not exist. Computationally efficienct physically-based methods tend to be inaccurate, and accurate methods tend to be inefficient. Additionally, different applications require different levels of physical abstraction. One wouldn't animate the motions of a forest at the cellular level. Physically-based methods are thus not ideal for all applications, especially those in which efficiency and stability play more important roles than versatility; data-driven methods offer exactly such traits.

In the context of plant animations, data-driven methods are those in which plant motion is synthesized primarily through the use of pre-obtained data. This data is typically either empirically obtained from real-world specimen (e.g. motion captured data), or it is generated using some sort of physical property of real-world specimen (e.g. modal analysis methods).

The key mindset behind these methods is that it is generally faster to 'look-up' stored motion

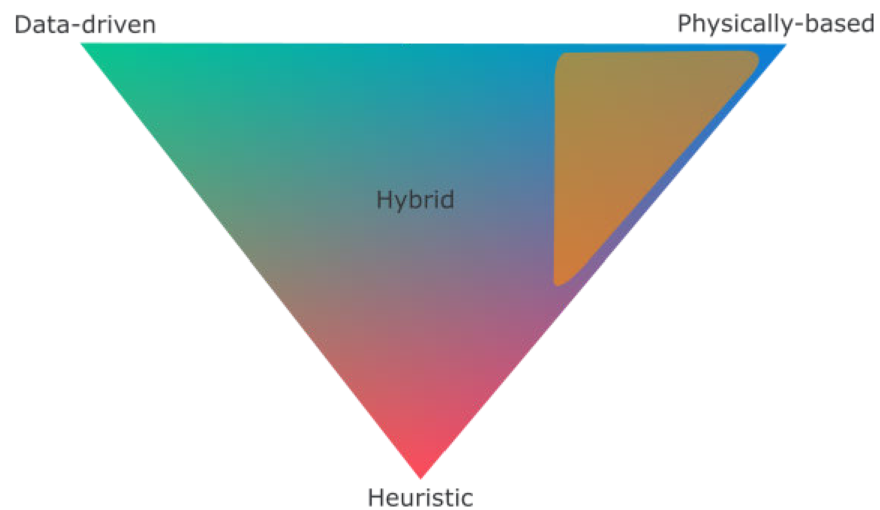
than it is to generate it on the fly. Also, since all the motion originates from this 'database', the output motion will be *predictable* (that is, there is a reassurance that simulations will not 'blow up' due to numerical instabilities, as is often the case in physically-based animations).

Concrete examples of procedural data-driven animation methods include techniques like "motion graphs" [43, 45] and "video textures" [80]. The main premise behind both of these methods is to create infinite synthesized motion from finite input motion. In the motion-graph method, connections between different sets of input videos are created by finding seamless transitions between similar frames in different sections of the input data, this map is what is called the *motion graph*. Then, at runtime, one can selectively traverse the motion-graph endlessly, yielding infinite motion from finite input data. Video textures work by randomly rearranging (and possibly blending) original frames from the source video, so that individual frames may be re-used, but the video will never loop. James et al.'s approach [39] extends motion graph and video texture techniques to work on scenes with many elements such as bundles of plants; instead of trying to find unlikely transitions between entire scenes, they find transitions between the individual elements of the scene, such as individual plants or even the individual organs of each plant. Haevre et al. [92] and Zhang et al. [99] explore data-driven animation on trees by first using offline physically-based methods to generate the database, and then using motion-graph techniques on this data to synthesise tree motion in real time. On a similar note, both Chuang et al. [17] and Habel et al. [34] have used *stochastic motion textures* - noise-generated textures capturing the stochastic nature of the wind - to drive the oscillations of branches on a look-up basis.

There are also methods that go a level of abstraction further: instead of using input motion to directly synthesize the output motion, use the input motion to extract parameters that can then be used to synthesize the output motion. For example, Sun et al. [87] extract wind parameters from a video of a tree undergoing wind motion, which can be used on other

plants, as well as different objects entirely like snow, dust, hair, etc. Wang et al. [93] extract the material parameters of plants from video, these material parameters are then used to run physically-based finite-element-method (FEM) [102] simulations of a wide variety of plants.

2.2.2 Physically-based methods



It is not at all trivial to animate plants in a physically-based manner in real-time; user interactions are unpredictable, environmental forces are chaotic, and internal binding forces are difficult to quantify. Nevertheless, the motion of plants is inherently appealing to the human eye, and as such, we are drawn as computer scientists to the problem of synthesizing such motion. This endeavour goes by the name of *physically-based animations*. In short, physically-based animation requires that a set of equations of motion be integrated forward in time using some sort of numerical method.

There exist a wide variety of physically-based approaches with which to simulate the motion of plants, with each having their own advantages and disadvantages. However, by construction of the problem, all physically-based works require two things: the creation of a discretized physical model of the real-life object they are representing - the *physics model* -

and a simulation algorithm that will operate on said model - the *simulation method*. The next few subsections explore common physically-based animation techniques within the context of plant animations.

Link-and-spring methods

The most common approach within physically-based plant animations is to represent the plant with thin rigid links connected by angular spring-like joints [79, 20, 60, 64]. The idea here is that each link shall be defined to be undeformable, with the incentive being that plant motion will instead emerge from changes in the relative orientations of the links. Each angular spring has two uses; to resist rotation and, once rotated, to introduce a *restoration torque* which will attempt to bring the spring back to its rest orientation. On the grander scale, these restoration torques act as a mechanism with which to re-orient a plant back to its rest pose.

Implementing physically-based simulations of link-and-spring methods is non-trivial for two key reasons. First, it is difficult to calculate the inertial properties of any particular interior element of the branching structure because it depends on the the inertia of everything both directly and indirectly attached to it. Secondly, since most elements are attached to other elements that are in motion themselves, the 'child' elements experience what are called *fictitious forces*. A fictitious force is a *perceived* force due to the acceleration of the object it is attached to. Both of these problems are difficult to handle in real-time.

In order to to maintain real-time stability, Sakaguchi and Ohya [79] employ a *decoupled system*; they ignore the accelerations and velocities of parent rods when computing the acceleration of child rods, and ignore the inertia of children when calculating the inertia of parents. This method greatly simplifies the dynamics calculations and has been adapted by other works as well [20, 34, 64], however, it does significantly compromise the quality

of certain animation scenarios. As an example, let's take a potted plant in static equilibrium; if we were to accelerate the pot alone, we would expect the plant to 'swing back' in the opposite direction of acceleration as it attempts to maintain zero inertia, and we would expect the plant to oscillate back and forth thereon until it uses up all its energy. However, in a decoupled system as described above, the plant would remain stationary the entire time because the links would ignore the acceleration of the pot altogether.

Another simplifying shortcut is to ignore axial twisting between segments [95]. This is cost effective because we are essentially ignoring one degree of freedom per spring. This is a relatively safe shortcut because plant motion mostly emerges from bending as opposed to twisting, but there are definitely scenarios where this is not the case (i.e. twisting the stem of a herbaceous plant).

Rigid-body methods

One step slightly beyond rod and spring methods in plant animations are *rigid-body methods*. The idea here is to represent the plant with rigid-bodies that are connected with elastic spherical joints, and to use rigid-body dynamics algorithms [4, 25] as their simulation method. This is the method employed in this thesis.

The equations and algorithms governing the dynamics of *articulated bodies* - which is a common term given to systems of rigid bodies connected by joints - have been theoretically known by roboticists for decades before their introduction into computer graphics [1]. Most stem from either the Euler-Lagrange equations of motion, which are described in the next chapter, or the Lagrangian formulation of the equations of motion for a mechanical system.

There are two key caveats in pursuing an articulated-body implementation; the first is the

[1] A wortwhile survey is presented in [26].

'mental barrier' that can arise due to the increased level of complexity in the mechanics and algebra required to write articulated-body software, and the second is a 'technological barrier' that arises due to the computational complexity and numerical instability of articulated-body simulations. This is because not only do articulated-body methods experience the same instabilities as link-and-spring methods, but they do so at a much larger extent. For example, even moreso than link-and-spring methods, articulated-body simulations tend to be immensely *stiff*. Within the context of numerical analysis, a stiff system is one in which small perturbations in the inputs yield potentially large and unpredictable responses in the system's output. In the context of articulated-bodies, this means that small forces may result in large and unpredictable accelerations (butterfly effect). This combined with the fact that every rigid-body is connected by a stiff spring means that there are a lot of large forces contributing to the system's instability. It should be noted that this instability was the key problem that led to major setbacks during the development of this thesis, and is discussed in Section 9.4.2.

As such, a stable and efficient numerical integration scheme of such systems remains a work in progress within the field. Quigley et. al [72], for example, approach their Featherstone-esque tree dynamics by evolving their springs independently of one another by solving their analytic equations locally. This may compromise the 'liveliness' of tree motion, but it makes it feasible to simulate large, stiff trees in real-time; something that can *not* yet be done in a stable manner otherwise.

Deformable rod methods (1-D continuum mechanics)

Other methods treat individual plant segments as deformable instead of rigid. Habel et al. [34] use the explicit analytic formula of the *1D Euler-Bernoulli beam model* to drive the deformations of individual, decoupled branches of varying thicknesses. Bertails [10]

represents the branches of a tree with thin deformable rods, doing so by solving Kirchoff's discrete equations for piecewise helical rods, except unlike its predecessors [61, 11], does so in a recursive manner similar to Featherstone [21] to achieve real-time results. Likewise, Bergou et al. [9] are able to model bending and twisting in a tree with techniques derived from nonlinear Kirchoff rods. Pirk et al. [63] use PBD-based Cosserat rods [44], which are a generalization of Kirchoff rods that add stretching and shearing, to model the dynamics of branches in their combustible trees models.

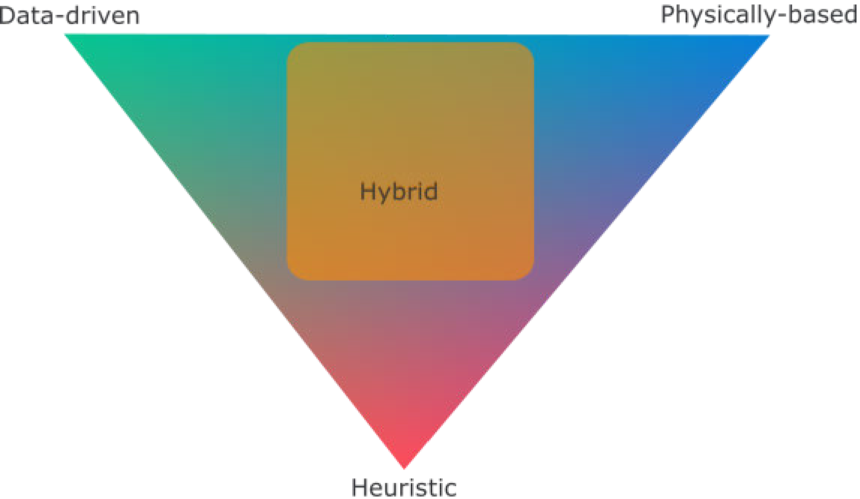
Deformable volume methods (3-D continuum mechanics)

Finally, as computational power increases, it is becoming feasible to run three-dimensional continuum mechanics algorithms in real-time. The most common approach in this area is geared towards finite-element method (FEM) simulations. The main setback with such techniques is that they struggle to model highly elastic materials, which is an ubiquitous property of many plants. A common approach is to partition the deformable body and introduce constraints that allow large deformations between the partitions [90, 6], giving an illusion of high flexibility. A handful of papers have emerged from Jernjej Barbič's lab in recent years exploring interactive large-deformations of plants in real-time, also using finite-element method simulations [5, 14, 6, 101, 93]. The efficient simulation of large deformations is in part possible due to heuristic speed-up techniques such as *domain decomposition* [6] (splitting up the deformable object into different domains) and *model reduction* [5] (reducing the complexity of each domain's dynamics). The resulting animations have high quality dynamics and can be ran at interactive speeds. An interesting extension to the methods in this thesis would involve the creation of a simulation-ready FEM-esque mesh from either an L-system, or some other procedural plant-modeling technique.

Multi-scale and level-of-detail methods

Multi-scale models are an interesting emerging concept in computer graphics. The key idea to identify here is that computer graphics as a field deals with the *subjective*; the 'appropriate' level of fidelity with which to model a virtual scene strictly depends upon the frame of reference of the observer. If an observer is very zoomed into the object of interest, for example, then what is important in that instance is to present a model of the microscopic structures and behaviours of the visible elements of the scene. On the contrary, if the user were to zoom out and observe the same exact scene from a distanced point of view, then it would only be worthwhile (and feasible) to model the structure and behaviours that are observable at that macroscopic frame of reference. Multi-scale methods are those in which we attempt to create a model of a scene that supports a *seamless* transition between the requirements of these differing frames of references.

Within the context of plant modeling and animations, most works in level of detail simulations pertain to the simulation of large forests. Giacomo et al. [20] use heuristic wind motion to drive the motion of thin delicate branches (predictable motion: inaccuracies are forgivable by the human eye), whereas physically-based methods were used to drive the motion of larger branches that were bent through user interaction (unpredictable motion: inaccuracies stick out to the human eye). Beaudoin and Keyser [7] successively group branches together (as needed) and simulate them as a single branch. Zhang et al. [98] assign adaptive priorities to joints in order to bypass the computation of lesser important joints in certain frames. They also incorporate LODs into their wind fields by generating mipmaps of their wind fields, such that further away trees can use coarser, and thus more efficient, windfield mipmaps.



2.2.3 Hybrid methods

From observation, the vast majority of works in plant animation are not purely data-driven nor purely physically-based, but rather, employ a mixture of both techniques. An argument can even be made that cited works in the previous sections are actually hybrid; heavily data-driven methods like motion-graph techniques may need physical cues to determine appropriate graph transitions, and heavily physically-based simulations require empirical data like shape and material properties to be accurate. However, these are just nuances, and in this section we will instead explore previous works in plant animations that employ a mixture of data-driven and physically-based techniques to a somewhat equal extent.

A common hybrid approach consists of using stochastic data to aide in the physically-based simulation of a plant. The main justification here being that since we often perceive natural phenomena as random, it can adequately be modeled as such. Early examples in computer graphics include stochastic fractals for mountain range generation [51, 29] and stochastic particle effects for fuzzy objects and explosions [74]. In the context of plant animation, wind motion is the main cause of stochastic motion. Shinya and Fournier [83] model the stochastic properties of wind in the fourier domain, and use this data to drive the forwards time-integration of their physics model, bypassing the need to calculate complex wind forces.

Stam [85] does the same as Shinya and Fournier, but instead of synthesising the wind field (the cause), he synthesizes the motion of the branches themselves (the effect).

Another hybrid approach consists of using data-driven approaches for some parts of the simulation, and using physically-based approaches for others. Ota et al. [60] use $1/f^{\beta}$ noise - whose frequencies have been shown to be prevalent in many natural phenomena - to drive the oscillation of the leaves, and resort to the aforementioned link-and-spring method for the physical simulations of the branches.

2.3 Previous works that couple plant modeling and animation

In real plants, the time scales in which growth and dynamics operate are much different, meaning that instantaneous growth does not have a practical influence on instantaneous dynamical motion. Therefore, by coupled growth and dynamics, we are instead referring to capturing how every-day dynamical motion affects long-term growth and how long-term growth affects every-day motion.

The growth of a plant can have a significant impact on its every-day dynamics. A thickening stem will prevent bending, for example, whereas an elongating stem augments bending, possibly resulting in a plant being unable to support itself (e.g. Euler buckling). Additionally, the spontaneous dynamical forces acting on a plant can have a significant impact on long-term growth. For example, a plant subjected to constant harsh winds and thunderstorms may result in critical deformations and even loss of branches, which not only has immediate effects, but also permanently affects the future growth of the plant (the removal of existing branches could allow other branches to grow in their stead, for example). This means that

growth and dynamics are related to each other, and so the modeling of such interactions is essential.

We are therefore concerned in previous work that couple mechanical processes and developmental models of plants, that is, the plants can be influenced by environmental factors as their geometry and topology evolve over time. There are three notable classes of models present within the topic of 'coupling plant growth and animation':

1. Models that animate plant development without considering mechanical influences, but that do consider *mechanistic* ones.
2. Models that animate plant development considering mechanical influences using *quasi-statics*.
3. Models that animate plant development considering mechanical influences using *dynamics*.

Mechanistic growth models

Mechanistic growth models address environmental effects in a 'cause and effect' manner (i.e. 'if there is enough space, then grow in that direction'). Mechanical models constitute a subcategory within mechanistic models, and focus on using physics to resolve interactions. Mechanical models are therefore also mechanistic (i.e. 'if a force is acted, the acceleration response is as such'), but not all mechanistic models are mechanical.

The majority of growth models that interact with their environment presented thus far are mechanistic. For instance, the way in which plants interact with their environment presented in [52] can be regarded as a purely mechanistic manner of doing things. The idea is that there should be bi-directional information exchange between plants and their environment, with this information influencing plant growth over time. For example, there is competition

for space, competition for light, and even competition between root tips for nutrients and water in soil.

Mechanical growth models - quasi-statics

A quasi-static process is one that is assumed to occur at an infinitesimally slow rate. In the context of animating the development of plants, a quasi-static growth model may ignore the notion of *inertia* because we are assuming that the system is evolving so slowly such that it is in the absence of non-inertial accelerations. Let's say we are trying to model the effects that some natural phenomena has on plant growth; a quasi-static approach is ideal if one is more interested in the *final result* of this interaction, rather than in the motion the plant took to get there (which is what dynamics tackles).

For example, Power et al. [65] incorporated inverse kinematics into L-system models to support the interactive arrangement of organs. Their plants are modeled by rigid links connected with spring-like joints, which act as a discretization of a continuous bending rod. Jirasek et al. [40] extended the aforementioned work and fully integrated a quasi-static model of biomechanics into L-systems, which was used to model the effect of gravity and tropisms on the plant's shape; phenomena that were until then modeled through non-physically based methods.

Mechanical growth models - dynamics

Dynamics is the branch of physics concerned with the study of forces and their effects on motion. Developmental models that incorporate dynamics allow for the simulation of growth and dynamical effects at the same time. One may think that this is an unrealistic and inapplicable scenario because long-term growth and immediate physical motion operate on

completely different time scales. However, such models are educational in the sense that they can help botanists further understand the effects that environmental forces have on plant growth. In the context of games and virtual reality, such plant models would instantly be applicable to the emerging field of *procedurally generated worlds*. A key limitation in the field is the 'dullness' that oftentimes emerges due to lackluster procedural methods. Incorporating environmentally-sensitive plant models into the 'open-world' paradigm would allow developers to populate their worlds with plants that can adapt and truly belong to their diverse environments.

The effect of wind on the immediate and long-term shape of virtual plant branches and leaves were studied in by Derzaph [18] in the context of small herbaceous plants, and Pirk et al. [64] in the context of trees. In both works, the immediate plant response to wind are based off Sakaguchi's rod-spring method [79], and regarding long-term growth response, both employ a method in which the strength of the wind field determines the extent to which the plant exhibits permanent deformation. Derzaph uses L-systems to model their herbaceous plants, whereas Pirk et al. employ a competition for space approach which has been shown to be better suited for trees [62]. Pirk et al. also presented a particle-based model of model of *climbing plants* [63]; the plants, whom are physically represented by shape-matching clouds of particles [54], are able to latch onto neighbouring surfaces as they grow, whilst simultaneously avoiding penetration and conforming to gravitational forces.

The methodology presented in this thesis does *not* couple plant modeling and animation. This is because a snapshot of a developmental model is first taken, which is then used to create dynamics simulations. Growth and mechanics do not operate in parallel. However, the key motivator behind this work is indeed to further the work towards an unified physically-based model coupling plant growth and dynamics, and the topic is elaborated further in the last section of the thesis (Section 10.3).

Chapter 3

Physics background

3.1 Contents of the chapter

The dynamics algorithm to be presented in the next chapter is not necessarily straightforward, and as such, this chapter will aim to cover relevant background material that will ease in its understanding. Section 3.2 discusses the important notion of reference frames, and Section 3.3 summarizes the rudimentary components of rigid-body dynamics needed for the derivation of Featherstone's articulated-body algorithm.

A reader familiar with elementary rigid-body dynamics may find that they can skip this chapter.

3.2 Coordinate systems

A *coordinate system* or *coordinate frame*, also called a *reference frame* in physics, is an abstraction that lets us quantify the motion of the objects around us. An object has no 'intrinsic' position nor orientation, for example, but instead, these quantities are *relative* and always refer to an agreed upon coordinate system of our choosing. For the time being, the physics and mathematics employed will remain in Euclidean space, and not only this, but we

will also be sticking to the Cartesian representation of this space; these simplifications bring forth a simple description for the reference frames we will be using, which is that of a position vector $\mathbf{p}$ along with three right-handed orthonormal basis vectors, $\{\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}\}$, that will span our three dimensional Cartesian space. Technically, we also need to agree on a *length scale* on which each reference frame operates, but we will assume that unless otherwise specified, an unit length along an axis will correspond to a real-world metre for all reference frames presented hereafter.

The vectors $\mathbf{p}$ and $\{\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}\}$ are still abstract, meaning that we haven't agreed what they refer to (or equivalently, which reference frame they are in), therefore, we will postulate that unless otherwise specified, the positions and orientations of objects and reference frames will refer to the *world reference frame*, $\mathcal{F}_{\mathcal{O}}$, by default. The world reference frame $\mathcal{F}_{\mathcal{O}}$ looks as follows:

$$\mathbf{p}_{\mathcal{O}} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \text{ and } \{\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}\}_{\mathcal{O}} = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}. \tag{3.1}$$

The three basis vectors - or axes - in Equation 3.1 can be combined to make up what is called the *rotation matrix* of $\mathcal{F}_{\mathcal{O}}$, $\mathbf{R}_{\mathcal{O}}$, with each axis making up a column in the matrix. This is compactly notated as $\mathbf{R}_{\mathcal{O}} = [\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}]$ with matrix form

$$\mathbf{R}_{\mathcal{O}} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.2}$$

We can now formally define a general reference frame, $\mathcal{F}$, to be the tuplet $(\mathbf{p}, \mathbf{R})$ with $\mathbf{p}$

being its position and $\mathbf{R}$ its rotation matrix. Again, unless otherwise specified, these refer to the global frame $\mathcal{F}_{\mathcal{O}}$.

3.2.1 Transformation matrices

Now that we have defined a frame $\mathcal{F}$ to be the tuplet $(\mathbf{p}, \mathbf{R})$, we can define the *transformation matrix* of $\mathcal{F}$, $\mathbf{T}$, to be the following affine transformation matrix:

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{p} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_1 \\ r_{21} & r_{22} & r_{23} & p_2 \\ r_{31} & r_{32} & r_{33} & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.3}$$

The group of matrices with the above form can be referred to be the matrix group of *rigid-body motions.* They encapsulate the possible motions of a rigid body, which is that of a translation and rotation, but no deformation (e.g. no scaling nor skewing).

A transformation matrix $\mathbf{T}$ has three major uses:

1. to represent the configuration (position and orientation) of a reference frame.
2. to change the reference frame in which a vector or frame is represented.
3. to displace a vector or frame.

In the first case, $\mathbf{T}$ merely represents the configuration of a frame with respect to another frame ($\mathcal{F}_{\mathcal{O}}$ if no frame is specified). In the second case, $\mathbf{T}$ is an operator that changes the

reference frame in which a vector or frame is represented. This is called a *passive transformation* because the physical quantity in question is *not* being physically disturbed; we are merely changing the frame in which we wish to define it. Finally, in the third case, **T** is an operator that moves the vector or frame in question within the same reference frame. This is called an *active transformation* because this time the physical quantity *is* being physically disturbed.

Foreword on reference frames

The concept of reference frames and their relative transformations is an essential one in this thesis. This is because Featherstone's algorithm necessitates that each and every rigid-body (and joint) have a reference frame *embedded* within it, with the intention being that the transformation matrices will act as information-transfer protocols between the connected elements[1]. The location and orientation of the body-fixed frame is not critical because the only thing of importance is that the reference frame move and rotate with the body (one-to-one correspondence). If this is so, the rigid body's position and orientation can be wholly described by the position and orientation of its reference frame, meaning any calculations needing the relative configurations of bodies can resort to the relative configurations (and thus transformations) between their respective reference frames. This also means, however, that there will be an abundance of reference-frame related algebra in the next chapter, so for ease of following, this algebra will be introduced as needed.

[1] A reader familiar with L-systems might instantly see their theoretical correlation with Featherstone's algorithm.

3.3 Rigid body dynamics

3.3.1 Definition of a rigid body

A rigid body may be regarded as a system of particles whose relative positions are fixed. This means that a rigid body by itself cannot deform, but rather, it will be the relative displacements between interconnected rigid bodies that will ultimately yield intricate articulated-body motion. An independent rigid body may be uniquely described by three properties:

- Its mass
- Its center of mass (CoM)
- Its moment of inertia tensor

each of which will be explained in the next three subsections.

Mass of a rigid body

The mass of a rigid body represents its resistance to linear acceleration and is obtained through the volume integral of its density equation ρ

$$m = \int_v \rho dv. \tag{3.4}$$

However, for simplicity's sake, we will only be considering rigid bodies with uniform density, meaning ρ is a constant and may be taken out of the integral

$$m = \rho \int_v dv = \rho V \tag{3.5}$$

where V is the volume of the rigid body.

Center of mass of a rigid body

The center of mass of a rigid body is the weighted average position of the body. A special and useful property of the center of mass is that a linear force acting at the center of mass of a rigid body will create a linear acceleration and no angular acceleration. Its components are given by integrating over the volume of the body:

$$x_{cm} = \frac{\int_v \rho\, x dv}{\int_v \rho\, dv} \qquad y_{cm} = \frac{\int_v \rho\, y dv}{\int_v \rho\, dv} \qquad z_{cm} = \frac{\int_v \rho\, z dv}{\int_v \rho\, dv}. \tag{3.6}$$

Since we are only considering rigid bodies of uniform density, ρ is again constant and may be taken out of the integrals, yielding simpler formulas for the center of mass:

$$x_{cm} = \frac{\int_v x dv}{\int_v dv} \qquad y_{cm} = \frac{\int_v y dv}{\int_v dv} \qquad z_{cm} = \frac{\int_v z dv}{\int_v dv} \tag{3.7}$$

These equations are are also those of the *centroid* of the rigid body, which is always equal to the center of mass when density is uniform. We can take advantage of symmetries in locating the center of mass, which usually makes the process much quicker.

Moment of Inertia tensor of a rigid body

Just like mass represents a body's resistance to a change in its linear motion, the *moment of inertia* represents resistance to a change in its rotational motion. Let us imagine that a rigid-body has been impaled by an undeformable and infinitely thin metal rod. If we rotate the thin rod alongn its axis, then the rigid body rotates with it. This thin rod is what would be called the rigid body's current *axis of rotation*, and the amount of force needed to rotate the body would depend on how the rigid-body's mass is distributed relative to the thin rod. To generalize this scenario, we can say that the moment of inertia, I_{axis}, of a body with respect to an axis of rotation is

$$I_{axis} = \int r^2 dm \tag{3.8}$$

where r is the perpendicular distance of the mass differential dm with respect to the chosen axis of rotation. It is important to keep in mind that the magnitude of a rigid-body's mass is *inherent*, but the same can not be said for the moment of inertia; the magnitude of the moment of inertia strictly depends on the chosen axis of rotation in which we want to compute it.

Take into example a thin wire of length L, then $dm = \rho dl$ and we would integrate these squared distances over the length of the wire L with respect to our chosen axis of rotation. For a surface, $dm = \rho da$ and we integrate over the surface area A. Lastly, for a rigid body or volume, $dm = \rho dv$ and we integrate over the volume V. Intuitively, I_{axis} measures the average squared distance of the objects's mass from the axis of rotation.

Again, it is important to keep in mind that I_{axis} is always defined with respect to an axis of rotation, and the value of the moment of inertia will likely change depending on the location and direction of the rotation axis with respect to the rigid body. If we assume the thin rod is 'glued' in space, and all it can do is rotate itself and its rigid body along its axis, then it is an example of one-dimensional angular motion, however, things get significantly more complicated when the axis is free to translate and rotate in three dimensions.

When dealing with three dimensional rotational motion, instead of having to recalculate I_{axis} as the axis of rotation changes, it is simpler and more efficient to calculate a *moment of inertia tensor* **I** which is defined at a fixed location and coordinate system with respect to the rigid body. The moment of inertia tensor is a 3x3 symmetric, positive-definite matrix and has form

$$\mathbf{I} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix} \tag{3.9}$$

where the I_{ii} diagonal components are the moments of inertia about axis $\hat{\mathbf{i}}$, meaning if the axis of rotation were to coincide with $\hat{\mathbf{i}}$, then only the I_{ii} term would be needed, producing the aforementioned one-dimensional motion. The I_{ij} terms are called the *products of inertia*, and in general, appear because the axis of rotation points along an arbitrary direction. For the moments and products of inertia to be constant quantities, we would need to compute them with respect to a coordinate system that is fixed to the body and rotates with it; hence the usefulness of the aforementioned paradigm in which each rigid body has a coordinate frame attached to it.

The origin of the body-fixed coordinate frame can be located at any point in space, however, it is oftentimes convenient to pick the center of mass of the rigid body. On a greater note, it is extremely convenient to pick the *principal axes* of the rigid body as the body-attached reference frame axes. The principal axes of a rigid body are its frame axes such that $\mathbf{I}$ is diagonal, significantly simplifying all calculations involving $\mathbf{I}$.

The principal axes of a rigid-body need not be unique, but they always exist. We can mathematically see this is the case because $\mathbf{I}$ is symmetric and positive-definite, meaning it can always be diagonalized, regardless of the chosen point of origin. $\mathbf{I}$ can therefore take the form

$$\mathbf{I} = \begin{bmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{bmatrix} \tag{3.10}$$

where I_i are the principal axes of the rigid body. It is intuitive to assume that three mutually orthogonal axes of symmetry of a rigid-body make up a set of principal axes, and this is indeed correct.

The moment of inertia, angular momentum, and rotational kinetic energy of a rigid body about any arbitrary rotational axis all take on fairly simple forms in a coordinate system whose axes are aligned with the principal axes of the rigid body. For example, let's assume $\mathbf{n}$ is an unit vector denoting the axis of rotation with respect to the body-fixed coordinate frame. We can use the direction cosines of $\mathbf{n}$ $(cos\alpha, cos\beta, cos\gamma)$ with respect to $\mathbf{I}$ to compute the moment of inertia I_{axis} along this arbitrary axis:

$$
\begin{aligned}
I_{axis} = n^T I n &= \begin{bmatrix} cos\alpha & cos\beta & cos\gamma \end{bmatrix} \begin{bmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{bmatrix} \begin{bmatrix} cos\alpha \\ cos\beta \\ cos\gamma \end{bmatrix} \\
&= I_1 cos^2\alpha + I_2 cos^2\beta + I_2 cos^2\gamma
\end{aligned}
$$

However, it should be noted that the simulation method presented in this thesis never rotates a rigid-body about an abstract axis of rotation; 3-D rotations are always represented by three successive miniscule rotations about each of the body's principal axes, which is called the *Euler angle* approach. An informed reader may know that Euler angles are generally non-commutative, and can thus struggle in representing continuous rotations. This discussion is elaborated upon in Section 6.5.

3.3.2 Newton-Euler equations for a rigid body

The change of motion, or acceleration, of any rigid body may be fully described by the external forces and torques acting on it. These forces and torques may be summed up into a single linear force $\mathbf{f}$ acting at the center of mass of the body, and a single moment $\boldsymbol{\tau}$ acting

about the body's center of mass.

First, we note that the center of mass of a rigid body has the special property that a linear force acting through it induces no angular acceleration. Therefore, we can use Newton's second law to determine the resultant linear acceleration of the body due to the linear force $\mathbf{f}$

$$\mathbf{f} = \frac{d\mathbf{p}}{dt} = \frac{d(m\mathbf{v})}{dt} = m\frac{d\mathbf{v}}{dt} = m\mathbf{a}$$

where $\mathbf{p}$ is the body's linear momentum, m its mass, $\mathbf{v}$ the instantaneous velocity of the center of mass, and $\mathbf{a}$ the resulting linear acceleration of its center of mass.

For the rotational component, we can use the rotational equivalent of Newton's second law to determine the resultant rotational acceleration of the body due to $\boldsymbol{\tau}$,

$$\boldsymbol{\tau} = \frac{d\mathbf{L}}{dt} = \frac{d(\mathbf{I}\boldsymbol{\omega})}{dt}$$

where $\mathbf{L}$ is the angular momentum of the body, $\mathbf{I}$ its moment of inertia, and $\boldsymbol{\omega}$ its instantaneous rotational velocity. However, the quantities in the equation above have to be defined with respect to an inertial, non-accelerating reference frame, otherwise the equation is void. Any non-moving frame can be used, so let's just refer those quantities to our world frame $\mathcal{F}_{\mathcal{O}}$.

$$\boldsymbol{\tau} = \frac{d\mathbf{L}_{\mathcal{O}}}{dt} = \frac{d(\mathbf{I}_{\mathcal{O}}\boldsymbol{\omega})}{dt}. \tag{3.11}$$

However, this means that in order to use this equation of motion, we have to re-compute the body's world-frame inertia tensor, $\mathbf{I}_{\mathcal{O}}$, every time we step the simulation forwards in time. The problem here is that in the general case, $\mathbf{I}_{\mathcal{O}}$ is *not* aligned with the principal axes of the body, and thus Equation 3.11 needs to be in its non-diagonalized form:

$$\mathbf{L}_{\mathcal{O}} = \mathbf{I}_{\mathcal{O}} \cdot \boldsymbol{\omega} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}.$$

The issue here is that due to the existence of off-diagonal terms in this representation of the inertia tensor, the direction of the angular momentum does not necessarily align along the axis of rotation, which will greatly complicate future calculations. Therefore, we re-state the equation, but this time using the predefined diagonalized inertia tensor that will be rotating with our rigid-body, $\mathbf{I}_{\mathcal{B}}$:

$$\mathbf{L}_{\mathcal{B}} = \mathbf{I}_{\mathcal{B}} \cdot \boldsymbol{\omega} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}.$$

This simplification doesn't come for free, however. Since we are now defining our equations in a reference frame that is attached to a possibly accelerating rigid-body, then it is no longer an inertial frame of reference, and the equation of motion is incomplete. We need to add a velocity-dependent term to our equation to account for the fact that this reference frame could be accelerating:

$$\begin{aligned} \left(\frac{d\mathbf{L}}{dt}\right)_{\mathcal{O}} &= \left(\frac{d\mathbf{L}}{dt}\right)_{\mathcal{B}} + \boldsymbol{\omega} \times \mathbf{L}_{\mathcal{B}} \\ &= \frac{d(\mathbf{I}_{\mathcal{B}} \cdot \boldsymbol{\omega})}{dt} + \boldsymbol{\omega} \times \mathbf{L}_{\mathcal{B}} \\ &= \frac{d\mathbf{I}_{\mathcal{B}}}{dt}\boldsymbol{\omega} + \mathbf{I}_{\mathcal{B}}\frac{d\boldsymbol{\omega}}{dt} + \boldsymbol{\omega} \times \mathbf{L}_{\mathcal{B}} \end{aligned}$$

The time derivative of the body-fixed inertia tensor, $\mathbf{I}_{\mathcal{B}}$, within the viewpoint of the body-fixed frame is null, therefore that term cancels to zero:

$$\left(\frac{d\mathbf{L}}{dt}\right)_{\mathcal{O}} = \mathbf{I}_{\mathcal{B}}\frac{d\boldsymbol{\omega}}{dt} + \boldsymbol{\omega} \times \mathbf{L}_{\mathcal{B}}$$

$$= \mathbf{I}_{\mathcal{B}}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \mathbf{L}_{\mathcal{B}}.$$

Finally, all the calculations in the next chapter will use the body-fixed inertia tensors and angular momentum, $\mathbf{I}_{\mathcal{B}}$ and $\mathbf{L}_{\mathcal{B}}$, as opposed to their global counterparts, $\mathbf{I}_{\mathcal{O}}$ and $\mathbf{L}_{\mathcal{O}}$. Therefore, for legibility preferences, we will assume that $\mathbf{I} \equiv \mathbf{I}_{\mathcal{B}}$ and $\mathbf{L} \equiv \mathbf{L}_{\mathcal{B}}$ unless otherwise noted. The Newton-Euler equations for a rigid-body are thus

$$\begin{aligned} \mathbf{f} &= m\mathbf{a} \\ \boldsymbol{\tau} &= \mathbf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega} \end{aligned} \tag{3.12}$$

Equations 3.12 are called the Newton-Euler equations of rigid-body motion, and give the translational and angular accelerations of the body as a function of the input force.

.

Chapter 4

The Articulated-Body Algorithm

4.1 Chapter overview

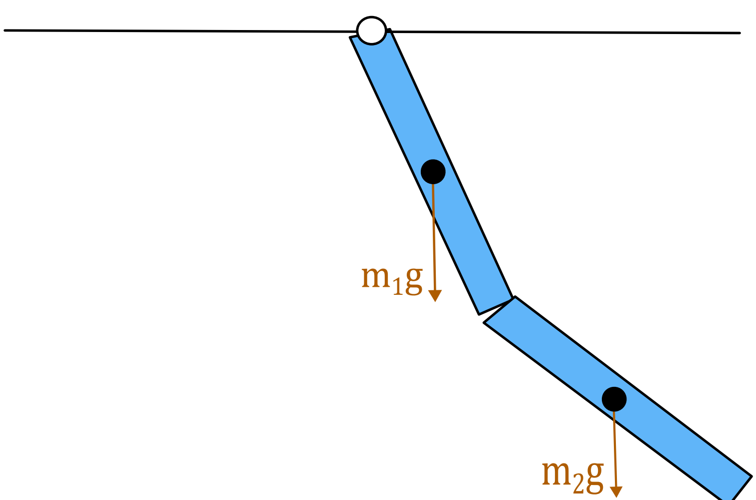


Figure 4.1: A 2-D double compound pendulum

The system illustrated in Figure 4.1 is called the double compound pendulum. We assume that the first body is connected to the ceiling via a revolute joint, and that a second body is connected to the first one via another revolute joint. Additionally, both bodies can accelerate under gravity. This system, albeit rather simple, is famous because it exhibits non-linear behaviour and does not have an analytical solution ([82] provides an analysis of its chaotic nature). This means that we must employ some sort of numerical algorithm to simulate it.

Appendix A provides a manual derivation of the equations of motion for the system, and, despite the system having only two degrees of freedom, finding its equations of motion still proved to be quite the tedious task. One might then see that it is unfeasible to manually derive the equations of motion for a larger rigid-body system; the algebra required would be overburdening. Therefore, we seek a way in which to automate the process of finding these equations of motion, which is part of the job of a dynamics algorithm.

The job of a dynamics algorithm is to construct and numerically evaluate the equations of motion of a given physical system [25, Chapter 3]. There are two fundamental classes of dynamics algorithms: *forward dynamics algorithms* and *inverse dynamics algorithms*. Forward dynamics algorithms tackle the problem of finding the acceleration of a system given its input forces, whereas inverse dynamics algorithms tackle the problem of finding the forces required to produce a desired acceleration of the system. The 'construction' portion of a dynamics algorithm refers to the process of finding the system of equations that yield the desired outputs of the system as a function of its inputs, whereas the 'evaluation' portion refers to the act of using these equations to 'march' the system to some point further in time using some sort of numerical algorithm.

In this chapter, no assumptions will be made regarding the composition of the physical system at hand, other than it be composed of rigid bodies connected by joints such that there are no loops in the system's topology. Such systems are prevalent in fields such as robotics, mechanical engineering, and computer animations to name a few, and have various names across literature. In this thesis, however, I will be sticking with the term *articulated-body* as it is the one employed by Roy Featherstone in *'The Calculation of Robot Dynamics Using Articulated-Body Inertias'* [21], which is the original paper on which the simulation methodology employed in this thesis is built upon.

The purpose of this chapter is to introduce some background information on rigid-body dynamics algorithms, and to manually derive the algorithm used in this thesis, which goes by the name the *articulated-body algorithm*. Section 4.2 provides a broad classification of rigid-body dynamics algorithms, as well as where the articulated-body algorithm lives within this classification. Section 4.3 provides a formal description of an articulated body, which will be used section 4.4, which provides a full derivation of the articulated-body algorithm from mechanical principles.

The contents of this chapter all fall under previous work, and were primarily adapted from the following sources:

1. BV Mirtich, *Impulse-based dynamic simulation of rigid body systems*, 1996 [53]
2. D House and JC Keyser, *Foundations of Physically Based Modeling and Animation*, 2016 [38]
3. R Featherstone, *Rigid body dynamics algorithms*, 2014 [25]
4. R Featherstone, *The calculation of robot dynamics using articulated-body inertias*, 1983 [21]
5. E Kokkevis, *Practical physics for articulated characters*, 2004 [42]

4.2 Classification of rigid-body dynamics algorithms

The decision to use the articulated-body algorithm to simulate the dynamics of plants was made early into the development of this thesis, and it stems from the observation that both the articulated-body algorithm and L-systems store and propagate information in a similar recursive manner, despite the fact that they achieve totally different goals. However, there

are several other viable algorithms that could have been used to simulate the motion of plants represented by rigid bodies. It is therefore important to introduce a general classification of rigid-body dynamics algorithms, and to point out where the articulated-body algorithm lives in this classification.

4.2.1 Maximal coordinate vs generalized coordinates

Let us begin with a general definition of an articulated body:

> *"An articulated body is a collection of rigid bodies that may be connected together by joints, and that may be acted upon by various forces. The effect of a joint is to impose a motion constraint on the two bodies it connects: relative motions are allowed in some directions but not in others"* [25].

This description already suffices to divide rigid-body dynamics algorithms into two classes, depending on how one wants to handle the joint motion constraints:

- **Generalized coordinate formulation**: Constraints are handled by removing the number of coordinates of the system such that only the coordinates can only represent desirable configurations.

- **Maximal coordinate formulation**: The number of coordinates is unchanged; constraints are instead handled by adding forces, called *constraint forces*, to the system. The purpose of these forces is to avoid, or restrict, the relative motion of bodies in impossible directions.

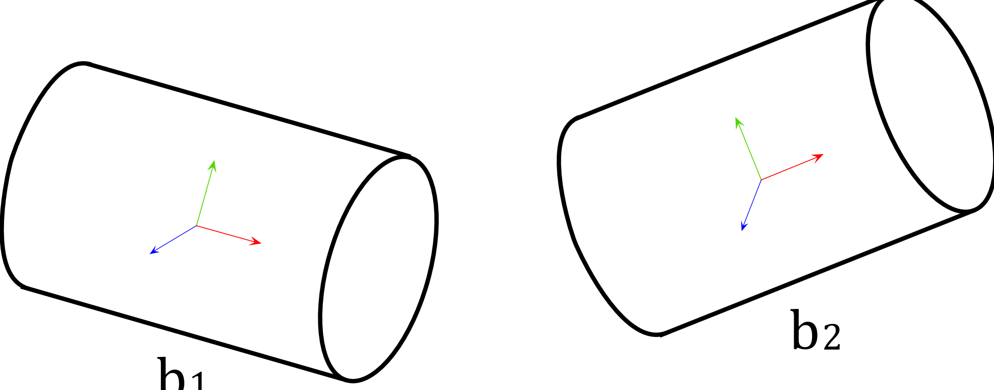
Example 1 offers a practical scenario of both approaches.

Given a system of m degrees of freedom and a set of constraints that remove c of those, then, in the generalized coordinate[1] approach, the objective is to parametrize the remaining

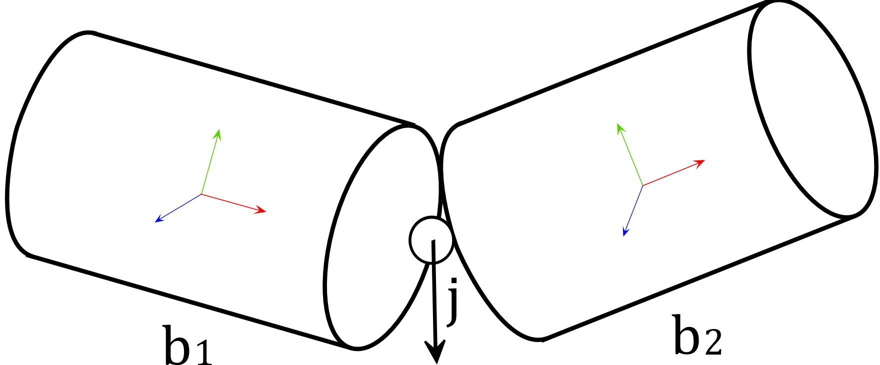
[1] Also called *reduced coordinates* or *minimal coordinates*

Example 1: Maximal vs generalized coordinates

Imagine two rigid bodies, b_1 and b_2, floating independently in space. The positional state of each rigid body may be denoted by 6 parameters; 3 translational and 3 rotational. Therefore, the system has $6{\cdot}2 = 12$ degrees of freedom.



If we were to connect the two bodies by a revolute joint j, which allows for one rotational degree of freedom between the two bodies, then the positional information of b_2 can now be wholly described by the positional information of b_1, plus the joint angle. This means the degrees of freedom of the system is down to 7.



By introducing this joint, we have added 5 *constraints* to the system. To handle them, we can either introduce forces into the system to maintain the constraints, or, we can reduce the number of coordinates of the system down to 7. The former is the maximal coordinate approach, and the latter is the generalized coordinate approach.

$n = m - c$ degrees of freedoms of the system. The n parameters are called the *generalized coordinates* of the system, and are historically denoted with the vector $\boldsymbol{q}$. This method of formulating mechanics problems has roots in Lagrangian mechanics, in which the number of generalized coordinates is exactly the number of degrees of freedoms in the system.

In contrast, given a system of m degrees of freedom and a set of constraints that remove c of those, then, in the maximal coordinate approach[2], the system would still be represented by the original set of m *maximal coordinates*, and the constraints would instead be enforced by introducing c *constraint forces* into the system. At each instant, a basis for the constraint forces is known beforehand; the Lagrange multipliers (which must be computed) are a vector of c scalar coordinates that describe the constraint force in terms of the basis [4].

Kinematic constraints

It should be noted at this point that, as far as literature goes, there appears to be no clear-cut 'superior' methodology to employ. If n is the number of degrees of freedom in the system, then $O(n)$ algorithms exist for both forward and inverse dynamics problems in both the maximal and generalized formulations. Therefore, in terms of efficiency, there is no clear advantage of one over the other. An in-depth comparison of both methods can be found in David Baraff's work on linear-time dynamics using Lagrange multipliers [4]. However, for our purposes, it suffices to consider how each method handles kinematic constraints, and the implications that this has on the suitability of either method over the other in a particular mechanics problem.

A kinematic constraint is a relative motion constraint between two individual rigid bodies. Figure 4.2 shows a classification of the possible kinematic constraints present in rigid-body

[2] Also called the Lagrange multiplier approach

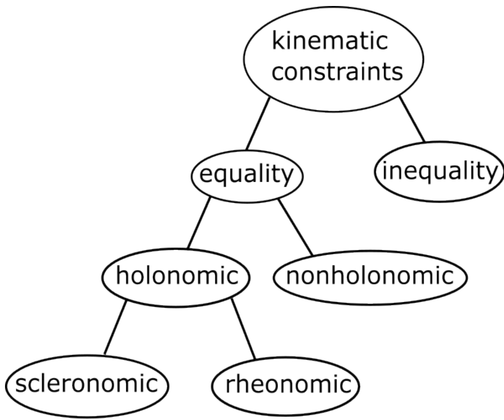


Figure 4.2: Classification of different kinematic constraints

systems. From a computer animations standpoint, the different types of constraints and their utilization can be summarized as follows:

- *Equality* constraints are suitable for enforcing permanent physical contact between two bodies, whereas *inequality* constraints allow for the separation of the bodies. Equality constraints are thus ideal for handling the hard constraints of joints, such as those imposed by prismatic and revolute joints, whereas inequality constraints can be used to describe softer phenomena such as collisions, bouncing, and loss of contact between bodies.

- *Holonomic* and *nonholonomic* constraints are both equality constraints. Holonomic constraints are functions of the positions of the system, whereas nonholonomic constraints are functions of the positions and also the velocities. Holonomic constraints only allow for the relative sliding between two objects, whereas nonholonomic constraints can also handle relative rolling between the objects.

- *Scleronomic* constraints and *rheonomic* constraints are both holonomic constraints. Scleronomic constraints do not depend on time, whereas rheonomic constraints do. If we take a simple pendulum as an example, which is scleronomous in nature, we could turn it into a rheonomous system by moving the pivot back and forth as a prescribed function of time. Scleronomic constraints, such as those imposed by revolute and pris-

matic joints, are widely used in rigid-body systems; all the constraints employed in this thesis are scleronomic.

With these kinematical constraints defined, we can deduce the following (as described in [4]): generalized coordinate approach excels at representing systems that have a relatively large number of scleronomic constraints, but no other types of kinematic constraints, whereas the maximal coordinate approach excels at representing systems with a mixture of kinematic constraints. The reasoning for the first point is that it is generally only worthwhile to re-parametrize the system if an abundant number of degrees of freedom have been strictly removed, which is the case when the system is tightly packed with scleronomic constraints. The reasoning for the second point is that since constraint forces are versatile and can enforce soft *and* hard constraints, then methods using constraint forces can better handle environments where different types of constraints are prevalent.

The plants constructed in this thesis will be connected together solely with scleronomic constraints (and *many* of them, for that matter), therefore, from a purely kinematical standpoint, it makes sense to explore a generalized coordinate approach.

4.2.2 Forward vs inverse dynamics

Forward dynamics algorithms tackle the problem of finding the acceleration of a system given its input forces, whereas inverse dynamics algorithms tackle the problem of finding the forces required to produce desired accelerations of the system. To understand this formally, we can look at the equation of motion for a general rigid-body system written in its canonical form[3]:

$$\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{\tau}. \tag{4.1}$$

[3] From hereon, assume a generalized coordinate formulation as that is the one used in this thesis

In the equation above, $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, and $\ddot{\boldsymbol{q}}$ are the generalized position, velocity, and accelerations vectors of the system, respectively. Likewise, $\boldsymbol{\tau}$ is the vector of generalized forces. Since from now on we are assuming a generalized coordinate formulation, then all of these vectors have size equal to the number of degrees of freedoms of the system, n. $\boldsymbol{H}$ is a $n \times n$ matrix called the generalized inertia matrix, and it is a mapping between forces and accelerations. It is written $\boldsymbol{H}(\boldsymbol{q})$ to indicate that it is a function of the generalized positions only. Likewise, $\boldsymbol{C}$ is the vector of generalized bias forces, and is written $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ to indicate that it is a function of both the generalized positions and velocities of the system. The vector of generalized bias forces can most simply be understood to be the value that $\boldsymbol{\tau}$ must take to produce a zero acceleration of the system as a whole. It accounts for the Coriolis and centrifugal forces, gravity, and any other forces not included in $\boldsymbol{\tau}$.[4]

Using Equation (4.1), we can now see that the forward dynamics problem consists of calculating $\ddot{\boldsymbol{q}}$ given $\boldsymbol{\tau}$, whereas the inverse dynamics problem consists of calculating $\boldsymbol{\tau}$ given $\ddot{\boldsymbol{q}}$. One can encapsulate these problems with the following two functions:

$$\ddot{\boldsymbol{q}} = \mathrm{FD}(model, \boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{\tau}) \tag{4.2}$$

and

$$\boldsymbol{\tau} = \mathrm{ID}(model, \boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}). \tag{4.3}$$

In both equations, *model* refers to the data structure containing all the information of the rigid-body system at hand not included in $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, and $\ddot{\boldsymbol{q}}$. The formal description of the *model* data structure is provided later in this chapter, and includes information such as the number of bodies and joints, the manner in which they are connected (the 'topology' of the system), the inertial properties of each bodies, and so on.

[4]It should be noted that the notation used for these equations varies between literature, but the overall meaning is consistent. The notation used in this thesis is the one used by Featherstone in [25].

4.2.3 Kinematic trees vs closed-loop systems

Rigid-body systems can be classified into two further classes; *kinematic trees* and *closed-loop systems*. A kinematic tree is any rigid-body system that does not contain loops in its connectivity, whereas a closed-loop system is any rigid-body system that does. It is considerably more difficult to devise an algorithm that works on closed-loop systems, and, interestingly enough, a standard procedure for devising such algorithms consists of adding extra constraints to a kinematic-tree algorithm in order to account for the closed loops [25, Chapter 8].

The source of this added difficulty can be understood by first noting that any rigid body in a kinematic tree can be seen as having a discrete set of *predecessors* and *successors*. The predecessors are any rigid bodies that come before it - all the way down to the base - and the successors are any rigid bodies that come after it, all the way up to the tips. When a loop is introduced, however, this trait is no longer true because a rigid body that's part of a loop is technically its own predecessor and successor, greatly complicating matters.

There are known algorithms that work on closed-loop systems (e.g. [25, Chapter 8]), but the algorithm employed in this thesis only works on kinematic trees. It should be noted that this limitation is not significant within the context of plant simulations because real plants seldomly show signs of cycles in their topology.

4.2.4 Classicifation of the articulated-body algorithm

We can now formally define the articulated-body algorithm with all these classifications at hand; the articulated body algorithm solves the forward-dynamics problem (i.e. finds the

accelerations of the system given input forces) in O(n) time, where n is the number of degrees of freedom in the system. Additionally, the articulated-body algorithm employs the generalized coordinate formalism, and operates strictly on kinematic trees.

Finally, there are two further types of forward-dynamics algorithms operating on kinematic trees:

1. **Inertia Matrix Methods**: These methods operate on a *global* scale. A system of linear equations is constructed, representing the equation of motion of the system as a whole.

2. **Propagation Methods**: These methods operate on a *local* scale. The idea is to recursively propagate dynamical information from one body to another in such a way that the accelerations can eventually be calculated one body at a time.

Inertia matrix methods are algebraically simpler than propagation methods, but can only achieve a time complexity of $O(n^3)$, whereas propagation methods can reach time complexities of $O(n)$.

The articulated-body algorithm is a propagation method and can thus reach a time complexity of $O(n)$ because it never strictly solves for $\boldsymbol{H}$ nor $\boldsymbol{C}$, which are time consuming to solve. Studies have been done showing that propagation methods start exceeding inertia matrix methods in terms of efficiency at around a dozen bodies [22, 25].

4.3 Formal description of an articulated body

Before delving into any dynamics equations, it is necessary to formally describe the elements that make up an articulated body. The following description assumes that the articulated-

body in question is an open kinematic-tree that is rooted somewhere in the world.

4.3.1 Parent and children bodies

An articulated body is a collection of rigid bodies and joints, where each joint is a connection between exactly two bodies. We will assume that the articulated-body is rooted somewhere in the world, and that the rooted rigid-body be called the *root body*, b_0, which is connected to the world via an immovable joint j_0.

Every other body b_i in the system must be connected to the system via an *inboard* joint. The body to which b_i's inboard joint connects to is referred to as b_i's *parent* body; all bodies have a parent body except for the root body.

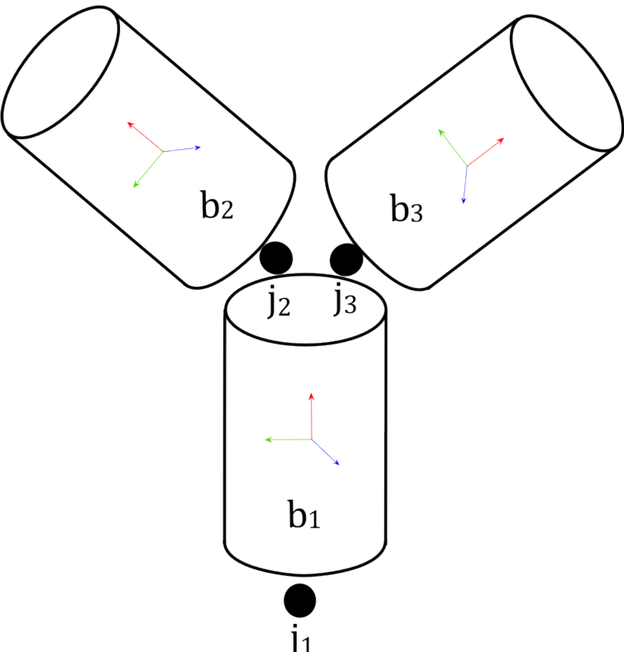


Figure 4.3: Let's take b_1 above into consideration. Its inboard joint is j_1, and its outboard joints are j_2 and j_3. Its parent is the root body b_0 (not pictured), and its children are b_2 and b_3.

Additionally, the set of all the bodies that have body b_i as its parent are called the *children*

of b_i. The joint(s) corresponding to the children of a particular body are called its *outboard* joints. See Figure 4.3.

4.3.2 Indexing and connectivity

Every rigid body and joint in the system are indexed. Additionally, the immovable root body and its immovable proximal joint are denoted b_0 and j_0, respectively. The remaining rigid bodies are indexed 1 to N_B, where N_B is the number of *movable* rigid bodies in the system. The (movable) joints are also indexed 1 to N_B. Every joint j_i is the corresponding inboard joint of body b_i for $1 \leq i \leq N_B$, such that a body always shares an index with its inboard joint. If i is the index of a body, then $\lambda(i)$ is an integer function that returns the parent of i. Likewise, if i is the index of a body, then $\mu(i)$ is a vector function that returns all the children of i. Together, the functions $\lambda()$ and $\mu()$ define the *connectivity* of the system. If we use the system in figure 4.3 as an example, then:

$$\lambda(1) = 0 \quad \text{and} \quad \mu(1) = \{2, 3\}$$

$$\lambda(2) = 1 \quad \text{and} \quad \mu(2) = \{\}$$

$$\lambda(3) = 1 \quad \text{and} \quad \mu(3) = \{\}$$

4.3.3 Body-fixed frames

The frame denoting the orientation and position of body i is denoted as $\mathcal{F}_i$. We define the *local* orientation and position of body i to be relative orientation and position of $\mathcal{F}_i$ with respect to its parent's frame, $\mathcal{F}_{\lambda(i)}$. We define the *global* orientation and position of body i to be the relative position and orientation of $\mathcal{F}_i$ with respect to the world reference frame, $\mathcal{F}_{\mathcal{O}}$. A frame $\mathcal{F}$ only has meaning if it is given with respect to another frame. This other frame is usually explicitly given, or we are referring to $\mathcal{F}_{\mathcal{O}}$.

4.3.4 Dynamic state of an articulated body

We define the *dynamic state* of an articulated body to be the instantaneous values of the system's state variables required to perform a given dynamics calculation. For the problem of forward dynamics, the dynamic state consists of the generalized joint positions,

$$\mathbf{q} = [q_1 \quad q_2 \quad \cdots \quad q_n]^T,$$

their first time derivatives,

$$\dot{\mathbf{q}} = [\dot{q}_1 \quad \dot{q}_2 \quad \cdots \quad \dot{q}_n]^T,$$

the generalized forces acting on the joints,

$$\boldsymbol{\tau} = [\tau_1 \quad \tau_2 \quad \cdots \quad \tau_n]^T,$$

and the global forces acting on the system's rigid bodies,

$$\mathbf{f}_{ext} = [\check{\mathbf{f}}^T_{ext_1} \quad \check{\mathbf{f}}^T_{ext_2} \quad \cdots \quad \check{\mathbf{f}}^T_{ext_{N_B}}]^T.$$

The values above define the dynamic state of the articulated-body, and are required in order to compute the instantaneous generalized accelerations of the system,

$$\ddot{\mathbf{q}} = [\ddot{q}_1 \quad \ddot{q}_2 \quad \cdots \quad \ddot{q}_n]^T.$$

Note that except for $\mathbf{f}_{ext}$, all of the vectors above have size n where n is the number of degrees of freedom of the system. The vector $\mathbf{f}_{ext}$ has size $6N_B$, where N_B is the number of movable rigid-bodies in the system. This is because six scalars (three translational and three rotational) are required to indicate the net external force acting on any rigid-body at an instant in time; these six scalars are encoded in a single six-dimensional vector, called a *spatial vector*, which is denoted with an upside-down hat. Spatial vectors and spatial vector

algebra will be introduced later in this chapter.

At this point, we may note that the joint coordinates *are* the generalized coordinates of the articulated-body, so from hereon, the terms generalized coordinates and joint coordinates are interchangeable. Additionally, it is helpful to visualize a joint as either being a *revolute* or *prismatic* joint; this is because any higher-order joint can be modeled as composition of revolute/prismatic joints with volume-less bodies in between. Lastly, just like an Euclidean vector belongs to Euclidean space, the generalized coordinate vectors of the articulated body belong to what is called the *configuration space* of the system.

It is worth mentioning that whilst vectors generally start at index 0, the elements q_0, $\dot{q}_0$, and $\ddot{q}_0$ correspond to the 0-DoF joint connecting the immovable root body to the earth (one can even say that the immovable root body *is* the earth), thus they always have a value of zero and for convenience, are generally omitted from their vectorial representation.

4.4 The articulated-body algorithm

A forward dynamics problem requires finding the accelerations of a mechanical system given its input forces. As an example, let's take a single rigid body m_1 and assume it is freely floating in a vacuum but currently stationary. If we act upon it with a linear force at the center of mass, we can use Newton's equations to find that the resulting acceleration only has a linear component which is equal to $\mathbf{a} = \mathbf{f}/m_1$ (Figure 4.4).

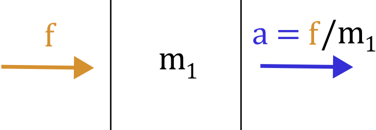


Figure 4.4: A rigid body experiencing simple linear motion.

However, if another body were to be glued on top of it, then the previous equation no longer works because the effective mass of m_1 has been changed. Additionally, the center of mass will have likely changed as well, so we wouldn't even be able to say that the resulting acceleration would only have a linear component (Figure 4.5).

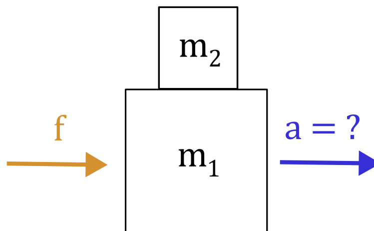


Figure 4.5: A linear force not about the center of mass results in a non-trivial system.

Furthermore, what if we applied some sort of external torque at some point on m_2? How would this affect the motion of m_1? (Figure 4.6).

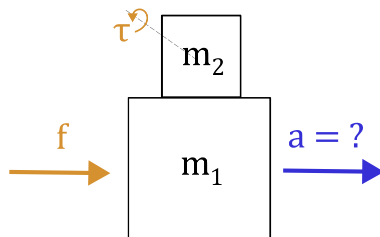


Figure 4.6: A linear force not about the center of mass plus a torque about a different point results in a complicated articulated-body system.

The articulated-body algorithm tackles this problem by traversing the open kinematic tree three times and recursively calculating different mechanical properties in each traversal. The input to the algorithm is the current dynamic state of the system, and the output of the system is the instantaneous acceleration of the system, $\ddot{\mathbf{q}}$:

1. The first traversal starts at the root and ends at the leaves. It is in charge of computing the *absolute* position and velocities of the system, as well as how a specific body's acceleration is influenced by the motion of the bodies that come before it in the

kinematic tree. An *absolute* quantity is one that is defined with respect to the global coordinate frame, $\mathcal{F}_{\mathcal{O}}$.

2. The second traversal starts at the leaves and ends at the root. It is in charge of letting each rigid body know how its children affect the body's *apparent* inertia and acceleration in that instantaneous configuration of the system.

3. The third and last traversal starts at the root and ends at the leaves. It is in charge of computing the acceleration of the system by making use of the properties derived in the previous two steps.

The remainder of this chapter deals with the derivation of these three algorithms, and attempts to do so from mechanical principles.

4.4.1 First pass: forward propagation of velocities and accelerations

The instantaneous absolute motion of the system has to be known in order to carry out any dynamics calculations. Therefore, the first set of derivations we will be looking at consists of computing the absolute motion of each rigid body given the generalized motion of the system. This problem is summarized as follows:

Problem: Forward propagation of velocities and accelerations

Given the instantaneous joint positions $\mathbf{q}$, velocities $\dot{\mathbf{q}}$, and accelerations $\ddot{\mathbf{q}}$ of the articulated body, compute the absolute linear velocity $\mathbf{v_i}$, angular velocity $\boldsymbol{\omega}_i$, linear acceleration $\mathbf{a}_i$, and angular acceleration $\boldsymbol{\alpha}_i$ of each rigid-body.

This algorithm will essentially translate the positions, velocities, and accelerations of the

system from configuration space into Euclidean space. The starting point for this algorithm is to note that the absolute motion of the base body is zero, therefore it is known and we can inductively start from there.

The mathematical procedure for this problem involves finding the equations that make up a particular *recurrence relation*. Recurrence relations consist of an equation, or a set of equations, that define a sequence. As long as as an initial state is provided, these equations give us the next term of the sequence as a function of the previous term(s). For the problem at hand, we need to prove the vailidity of the following recurrence:

> **Base case**: the absolute motion of the root body is initially known.
>
> **Recursive step**: the absolute motion of any child body[5] can be stated in terms of the absolute motion of its parent, plus the motion of the joint connecting them.

The way to prove this recurrence relation is by finding its corresponding equations. The recurrence relation above is special in that only the immediately previous term is needed to define the next one. Our goal is thus to prove that this recurrence relation holds, and we shall do so by finding the equations that give the motion of any child body in terms of the motion of its parent body, plus the motion of the joint connecting them. Once we have these equations, we can propagate the velocities and accelerations from the stationary root body outwards, yielding the absolute motion of each body with respect to the global reference frame, $\mathcal{F}_{\mathcal{O}}$. **This step will also compute any non-inertial accelerations that a body may be experiencing due to the velocities of its ancestors**.

Base case

The base case is trivial here. The root body with index 0 is immovable by definition, so its absolute motion with respect to $\mathcal{F}_{\mathcal{O}}$ is zero. Therefore,

[5] Recall that all bodies except the root body are child bodies

$$\mathbf{v_0} = \mathbf{0}$$

$$\boldsymbol{\omega}_0 = \mathbf{0}$$

$$\mathbf{a}_0 = \mathbf{0}$$

$$\boldsymbol{\alpha}_0 = \mathbf{0}$$

Inductive case

For the inductive case, we will need to find the absolute motion of body i. Our inductive hypothesis will be that we already know the motion of i's parent, $\lambda(i)$.

To proceed, we first define a few geometric quantities between the parent and child that are required in order to derive the propagation equations. They are illustrated in Figure 4.7. The vector $\mathbf{r}_i$ denotes the displacement between the body-fixed frames of the parent and child: $\mathcal{F}_{\lambda(i)}$ and $\mathcal{F}_i$. The direction vector $\hat{\mathbf{u}}$ is the joint axis of joint i; if i is revolute, $\hat{\mathbf{u}}$ is the axis of rotation, else, i is prismatic and $\hat{\mathbf{u}}$ is the direction of translation[6]. Finally, $\mathbf{d}_i$ is the vector pointing from the joint's location to the location of $\mathcal{F}_i$.

Now that we have these geometric quantities, we can start by defining the *relative angular velocity* of a body to be the angular velocity it would have if its parent were fixed in space. Equivalently, it is the angular velocity of the body due solely to the motion of its connecting joint. The absolute angular velocity of body i can therefore be stated as the absolute angular velocity of its parent, plus its relative angular velocity with respect to the parent;

$$\boldsymbol{\omega}_i = \boldsymbol{\omega}_{\lambda(i)} + \boldsymbol{\omega}_{rel}. \tag{4.4}$$

The equation for the linear relative velocity is is nearly identical, except there's the additional

[6]Prismatic joints were not used in this thesis, but the derivation will still include them for completeness.

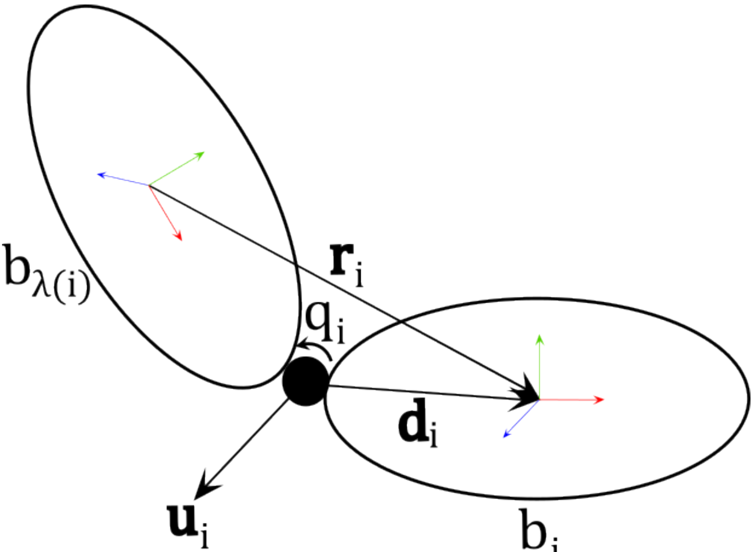


Figure 4.7: Necessary geometric quantities between two bodies. The diagram hints towards a revolute joint, but the same quantities apply for a prismatic joint.

cross-product term of $\boldsymbol{\omega}_{\lambda(i)} \times \mathbf{r}_i$;

$$\mathbf{v}_i = \mathbf{v}_{\lambda(i)} + \boldsymbol{\omega}_{\lambda(i)} \times \mathbf{r}_i + \mathbf{v}_{rel}. \tag{4.5}$$

This term appears because unless $\mathbf{r}_i = [0, 0, 0]^\intercal$, the angular velocity of the parent will induce a linear component on the child's velocity. The angular acceleration may now be obtained by taking the time derivative of Equation (4.4):

$$\boldsymbol{\alpha}_i = \boldsymbol{\alpha}_{\lambda(i)} + \boldsymbol{\alpha}_{rel}. \tag{4.6}$$

Similarly, the linear acceleration is obtained by taking the time derivative of (4.5);

$$\mathbf{a}_i = \mathbf{a}_{\lambda(i)} + \boldsymbol{\alpha}_{\lambda(i)} \times \mathbf{r}_i + \boldsymbol{\omega}_{\lambda(i)} \times \dot{\mathbf{r}}_i + \mathbf{a}_{rel}.$$

Since $\mathbf{r}_i$ defines the relative position of body i with respect to its parent, its time rate of change $\dot{\mathbf{r}}_i$ is the sum of two quantities: an induced linear velocity due to the rotation of the

parent, $\boldsymbol{\omega}_{\lambda(i)} \times \mathbf{r}_i$, plus the linear relative velocity of body i with respect to its parent, $\mathbf{v}_{rel}$,

$$\dot{\mathbf{r}}_i = \boldsymbol{\omega}_{\lambda(i)} \times \mathbf{r}_i + \mathbf{v}_{rel}.$$

This can now be substituted back into the equation of the linear acceleration to obtain

$$\mathbf{a}_i = \mathbf{a}_{\lambda(i)} + \boldsymbol{\alpha}_{\lambda(i)} \times \mathbf{r}_i + \boldsymbol{\omega}_{\lambda(i)} \times (\boldsymbol{\omega}_{\lambda(i)} \times \mathbf{r}_i) + \boldsymbol{\omega}_{\lambda(i)} \times \mathbf{v}_{rel} + \mathbf{a}_{rel}. \tag{4.7}$$

Equations (4.4) through (4.7) state $\boldsymbol{\omega}_i$, $\mathbf{v}_i$, $\boldsymbol{\alpha}_i$, and $\mathbf{a}_i$ in terms of the state of the body's joint and parent. What's left is to figure out the remaining unknowns in these equations, which are the relative velocities and accelerations of the body: $\boldsymbol{\omega}_{rel}$, $\mathbf{v}_{rel}$, $\boldsymbol{\alpha}_{rel}$, and $\mathbf{a}_{rel}$. First, we recall that $\boldsymbol{\omega}_{rel}$ and $\mathbf{v}_{rel}$ are the angular and linear velocities of a body due solely to the motion of its joint, therefore they should be functions that only depend on the joint's velocity variable, $\dot{q}_i$. If the joint is prismatic, then then two bodies it connects can only slide forwards and backwards with respect to each other, meaning the joint axis vector $\hat{\mathbf{u}}_i$ points from the joint to the child, thereby creating no angular veclotiy;

$$\begin{aligned} \boldsymbol{\omega}_{rel} &= \mathbf{0}, \\ \mathbf{v}_{rel} &= \dot{q}_i \hat{\mathbf{u}}_i. \end{aligned} \tag{4.8}$$

For a revolute joint, $\hat{\mathbf{u}}_i$ is the axis of rotation, giving its equations a slightly different look:

$$\begin{aligned} \boldsymbol{\omega}_{rel} &= \dot{q}_i \hat{\mathbf{u}}_i, \\ \mathbf{v}_{rel} &= \dot{q}_i \hat{\mathbf{u}}_i \times \mathbf{d}_i. \end{aligned} \tag{4.9}$$

In the equations above, $\mathbf{d}_i$ is the relative displacement between the joint and child body, and the cross product term in the equation for $\mathbf{v}_{rel}$ arises because unless $\mathbf{d}_i = [0, 0, 0]^\intercal$, the angular velocity of the joint will induce a linear component on the child's velocity. Note that $\dot{q}_i$ is not bold because it is the single joint velocity between the two bodies and is thus

a scalar. Given these equations, we can now solve for $\boldsymbol{\alpha}_{rel}$, and $\mathbf{a}_{rel}$ for both prismatic and revolute joints by taking the time derivatives of (4.8) and (4.9), respectively. The main trick in these derivations is to note that since $\hat{\mathbf{u}}_i$ rotates with body $\lambda(i)$, then the change in $\hat{\mathbf{u}}_i$ is $\boldsymbol{\omega}_{\lambda(i)} \times \hat{\mathbf{u}}_i$, leading to the following derivation:

$$\frac{d}{dt}(\dot{q}_i\hat{\mathbf{u}}_i) = \frac{d}{dt}(\dot{q}_i)\hat{\mathbf{u}}_i + \dot{q}_i\frac{d}{dt}(\hat{\mathbf{u}}_i) \quad = \ddot{q}_i\hat{\mathbf{u}}_i + \boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i\hat{\mathbf{u}}_i.$$

Therefore, for a prismatic joint,

$$\begin{aligned} \boldsymbol{\alpha}_{rel} &= \mathbf{0}, \\ \mathbf{a}_{rel} &= \ddot{q}_i\hat{\mathbf{u}}_i + \boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i\hat{\mathbf{u}}_i. \end{aligned} \tag{4.10}$$

And for a revolute joint,

$$\begin{aligned} \boldsymbol{\alpha}_{rel} &= \ddot{q}_i\hat{\mathbf{u}}_i + \boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i\hat{\mathbf{u}}_i, \\ \mathbf{a}_{rel} &= \ddot{q}_i\hat{\mathbf{u}}_i \times \mathbf{d}_i + \boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i\hat{\mathbf{u}}_i \times \mathbf{d}_i + \dot{q}_i\hat{\mathbf{u}}_i + (\dot{q}_i\hat{\mathbf{u}}_i \times \mathbf{d}_i). \end{aligned} \tag{4.11}$$

Equations (4.8) - (4.11) give us $\boldsymbol{\omega}_{rel}$, $\mathbf{v}_{rel}$, $\boldsymbol{\alpha}_{rel}$, and $\mathbf{a}_{rel}$ which may now be plugged back into equations (4.4) - (4.7). For a prismatic joint:

$$\begin{aligned} \boldsymbol{\omega}_i &= \boldsymbol{\omega}_{\lambda(i)} \\ \mathbf{v}_i &= \mathbf{v}_{\lambda(i)} + \boldsymbol{\omega}_{\lambda(i)} \times \boldsymbol{r}_i + \dot{q}_i\hat{\mathbf{u}}_i \\ \boldsymbol{\alpha}_i &= \boldsymbol{\alpha}_{\lambda(i)} \\ \boldsymbol{a}_i &= \boldsymbol{a}_{\lambda(i)} + \boldsymbol{\alpha}_{\lambda(i)} \times \boldsymbol{r}_i + \ddot{q}_i\hat{\mathbf{u}}_i + \boldsymbol{\omega}_{\lambda(i)} \times (\boldsymbol{\omega}_{\lambda(i)} \times \boldsymbol{r}_i) + 2\boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i\hat{\mathbf{u}}_i. \end{aligned} \tag{4.12}$$

And for a revolute joint:

$$
\begin{aligned}
\boldsymbol{\omega}_i &= \boldsymbol{\omega}_{\lambda(i)} + \dot{q}_i \hat{\mathbf{u}}_i \\
\mathbf{v}_i &= \mathbf{v}_{\lambda(i)} + \boldsymbol{\omega}_{\lambda(i)} \times \boldsymbol{r}_i + \dot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i \\
\boldsymbol{\alpha}_i &= \boldsymbol{\alpha}_{\lambda(i)} + \ddot{q}_i \hat{\mathbf{u}}_i + \boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i \hat{\mathbf{u}}_i \\
\boldsymbol{a}_i &= \boldsymbol{a}_{\lambda(i)} + \boldsymbol{\alpha}_{\lambda(i)} \times \boldsymbol{r}_i + \ddot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i + \boldsymbol{\omega}_{\lambda(i)} \times (\boldsymbol{\omega}_{\lambda(i)} \times \boldsymbol{r}_i) \\
&\quad + 2\boldsymbol{\omega}_{\lambda(i)} \times (\dot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i) + \dot{q}_i \hat{\mathbf{u}}_i \times (\dot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i).
\end{aligned}
\tag{4.13}
$$

The set of equations in (4.12) and (4.13) state the absolute velocities and accelerations of a body with respect to those of its joint and parent, proving that the recurrence relation holds □

Upon further inspection, one may see that some of the equations in (4.12) and (4.13) depend on the joint acceleration variables $\ddot{q}_i$. These joint accelerations are a product of the net forces acting on the system, and are not initially known. It is the job of the articulated-body algorithm to find their values. However, before we get into the derivation of the articulated-body algorithm, there is a problem that the set of equations in (4.12) and (4.13) are already getting messy. The reason for this is two-fold:

- We have separate equations for revolute and prismatic joints.
- Even after having picked either a revolute or prismatic context, we still have separate equations for the angular and linear components.

If we want cleaner algebra, which is a must if we are to derive the articulated-body algorithm, then we must use *spatial algebra*. Let us therefore introduce the basic elements of spatial algebra that will be utilized in the derivation of the articulated-body algorithm.

4.4.2 Spatial Algebra

Spatial vector algebra refers to the utilization of 6-D vectors, which in our context are called *spatial vectors*, for algebra involving rigid-body arithmetic. They combine the linear and angular components of rigid-body dynamics, greatly reducing the amount of algebra involved when carrying out rigid-body mechanics calculations. A thorough tutorial on 6-D vectors is offered in [24], and a more in-depth theoretical analysis on 6-D vectors and their underlying basis can be found in [23].

Spatial velocity and acceleration

The spatial velocity is the concatenation of the angular velocity and the linear velocity of a rigid body:

$$\check{\mathbf{v}} = \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v} \end{bmatrix}.$$

Note that since we need to differentiate between 6-D and 3-D vectors, we will depict 6-D spatial vector and matrices with an 'upside down' hat on top of it as above.

Similarly to the definition above, the spatial acceleration is the union of the angular acceleration and the linear acceleration of a rigid body,

$$\check{\mathbf{a}} = \begin{bmatrix} \boldsymbol{\alpha} \\ \mathbf{a} \end{bmatrix},$$

where $\boldsymbol{\alpha} = \frac{d\boldsymbol{\omega}}{dt}$ and $\mathbf{a} = \frac{d\mathbf{v}}{dt}$. Example 2 shows the spatial version of the Euler-Lagrange equations of motion for a rigid body.

Example 2: Euler-Lagrange equations in spatial notation

We saw in the previous chapter that the Euler-Lagrange equations of motion for a rigid body were as follows,

$$\begin{aligned}\mathbf{f} &= m\mathbf{a} \\ \boldsymbol{\tau} &= \boldsymbol{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \boldsymbol{I}\boldsymbol{\omega},\end{aligned}$$

where $\mathbf{f}$ is a linear force incident at the center of mass, and $\boldsymbol{\tau}$ is a torque about the center of mass. Also recall that the inertia tensor $\boldsymbol{I}$ is assumed to have been defined with respect to the center of mass of the rigid body, so that it is constant and does not need to be re-computed every simulation step.

We can write the same equation as follows:

$$\begin{bmatrix}\mathbf{f} \\ \boldsymbol{\tau}\end{bmatrix} = \begin{bmatrix}\mathbf{0} & \boldsymbol{M} \\ \boldsymbol{I} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\boldsymbol{\alpha} \\ \boldsymbol{a}\end{bmatrix} + \begin{bmatrix}\mathbf{0} \\ \boldsymbol{\omega} \times \boldsymbol{I}\boldsymbol{\omega}\end{bmatrix},$$

where $\boldsymbol{M}$ is the diagonal mass-matrix of the rigid body,

$$\boldsymbol{M} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix}.$$

We can then define the two new terms as follows:

$$\check{\boldsymbol{I}} = \begin{bmatrix}\mathbf{0} & \boldsymbol{M} \\ \boldsymbol{I} & \mathbf{0}\end{bmatrix}, \text{ and } \quad \check{\boldsymbol{p}} = \begin{bmatrix}\mathbf{0} \\ \boldsymbol{\omega} \times \boldsymbol{I}\boldsymbol{\omega}\end{bmatrix}.$$

Together, $\check{\boldsymbol{I}}$ and $\check{\boldsymbol{p}}$ define the *spatial isolated* quantities of the rigid body in question, and they incorporate the mechanical properties of a rigid body in isolation (i.e. not attached to a child or parent). $\check{\boldsymbol{I}}$ is the *spatial isolated inertia tensor* of the rigid body, expressed in body coordinates, and $\check{\boldsymbol{p}}$ is its *spatial isolated bias force*. The bias force is the external force that would be needed to produce no acceleration on the body (i.e. to counteract any velocity-dependent accelerations the body might be experiencing). With these quantities defined, the Euler-Lagrange equations can now assume their final spatial form,

$$\check{\boldsymbol{f}} = \check{\boldsymbol{I}}\check{\boldsymbol{a}} + \check{\boldsymbol{p}},$$

where $\check{\boldsymbol{f}}$ is the force acting on the body, $\check{\boldsymbol{I}}$ is the isolated inertia tensor of the rigid body, $\check{\boldsymbol{a}}$ is the acceleration of the body, and $\check{\boldsymbol{p}}$ is the isolated bias force of the body, all in spatial form.

Spatial Transforms

Just like there are 6-D vectors, there are also 6x6 matrices called *spatial matrices*. We can use these matrices to represent *spatial transformations*, which are super useful because they are used to translate physical quantities between coordinate frames. For example, the transformation of a dynamical quantity from frame $\mathcal{F}$ to frame $\mathcal{G}$ is denoted as ${}^{\mathcal{G}}\check{\mathbf{X}}_{\mathcal{F}}$.

A spatial transform between frames $\mathcal{F}$ and $\mathcal{G}$ can be thought of as having two components: one due to the position of frame $\mathcal{G}$ relative to $\mathcal{F}$, and the other due to the orientation of frame $\mathcal{G}$ relative to $\mathcal{F}$. The general form is provided below, with a detailed derivation presented in [25, Chapter 2]:

$$
{}^{\mathcal{G}}\check{\mathbf{X}}_{\mathcal{F}} = \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ -\tilde{\mathbf{r}} & \mathbf{1} \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ -\tilde{\mathbf{r}}\mathbf{R} & \mathbf{R} \end{bmatrix}, \tag{4.14}
$$

where $\mathbf{R}$ is the 3x3 rotation matrix corresponding to the rotation between $\mathcal{F}$ and $\mathcal{G}$, and $\tilde{\mathbf{r}}$ is the *cross operator* of $\mathbf{r}$, the offset vector from $\mathcal{F}$ to $\mathcal{G}$:

$$
\tilde{\mathbf{r}} = \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}.
$$

The two matrices in the middle of Equation (4.14) correspond to the translation and rotation matrices of the spatial transformation, respectively. The spatial transformation ${}^{\mathcal{G}}\check{\mathbf{X}}_{\mathcal{F}}$ can now be used to transform physical quantities from $\mathcal{F}$ to $\mathcal{G}$. For example, if we have a spatial velocity defined in frame $\mathcal{F}$, $\check{\mathbf{v}}_{\mathcal{F}}$, but we wanted to describe it with respect to frame $\mathcal{G}$, then we would carry out the following expression:

$$
\check{\mathbf{v}}_{\mathcal{G}} = {}^{\mathcal{G}}\check{\mathbf{X}}_{\mathcal{F}}\check{\mathbf{v}}_{\mathcal{F}}
$$

Note that Featherstone's algorithm is all about recursively propagating physical quantities from parents to children and vice versa. Therefore - albeit conceptually complex - spatial algebra and its notation will prove to be crucial in the rest of the chapter.

Spatial Force

Similar to velocities and accelerations, a spatial force is the union of a linear force and a torque, and has the form

$$\check{\mathbf{f}} = \begin{bmatrix} \mathbf{f} \\ \boldsymbol{\tau} \end{bmatrix}$$

Forward propagation of accelerations in spatial formulation

It turns out that one can compactly represent the previously derived equations for the forward propagation of accelerations in spatial notation. To reiterate, this is what they look like for prismatic joints (Equation 4.12),

$$\begin{aligned} \boldsymbol{\alpha}_i &= \boldsymbol{\alpha}_{\lambda(i)} \\ \boldsymbol{a}_i &= \boldsymbol{a}_{\lambda(i)} + \boldsymbol{\alpha}_{\lambda(i)} \times \boldsymbol{r}_i + \ddot{q}_i \hat{\mathbf{u}}_i + \boldsymbol{\omega}_{\lambda(i)} \times (\boldsymbol{\omega}_{\lambda(i)} \times \boldsymbol{r}_i) + 2\boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i \hat{\mathbf{u}}_i, \end{aligned}$$

and for revolute joints (Equation 4.13),

$$\begin{aligned} \boldsymbol{\alpha}_i &= \boldsymbol{\alpha}_{\lambda(i)} + \ddot{q}_i \hat{\mathbf{u}}_i + \boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i \hat{\mathbf{u}}_i \\ \boldsymbol{a}_i &= \boldsymbol{a}_{\lambda(i)} + \boldsymbol{\alpha}_{\lambda(i)} \times \boldsymbol{r}_i + \ddot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i + \boldsymbol{\omega}_{\lambda(i)} \times (\boldsymbol{\omega}_{\lambda(i)} \times \boldsymbol{r}_i) \\ &\quad + 2\boldsymbol{\omega}_{\lambda(i)} \times (\dot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i) + \dot{q}_i \hat{\mathbf{u}}_i \times (\dot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i). \end{aligned}$$

By gathering up common terms, we can re-express these equations in spatial notation. For prismatic joints, they look as follows,

$$\begin{bmatrix}\boldsymbol{\alpha}_i \\ \boldsymbol{a}_i\end{bmatrix} = \begin{bmatrix}\boldsymbol{\alpha}_{\lambda(i)} \\ -\boldsymbol{r}_i \times \boldsymbol{\alpha}_{\lambda(i)} + \boldsymbol{a}_{\lambda(i)}\end{bmatrix} + \ddot{q}_i \begin{bmatrix}\mathbf{0} \\ \mathbf{u}_i\end{bmatrix} + \begin{bmatrix}\mathbf{0} \\ \boldsymbol{\omega}_{\lambda(i)} \times (\boldsymbol{\omega}_{\lambda(i)} \times \boldsymbol{r}_i) + 2\boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i \hat{\mathbf{u}}_i\end{bmatrix}, \tag{4.15}$$

and for revolute joints,

$$\begin{bmatrix}\boldsymbol{\alpha}_i \\ \boldsymbol{a}_i\end{bmatrix} = \begin{bmatrix}\boldsymbol{\alpha}_{\lambda(i)} \\ -\boldsymbol{r}_i \times \boldsymbol{\alpha}_{\lambda(i)} + \boldsymbol{a}_{\lambda(i)}\end{bmatrix} + \ddot{q}_i \begin{bmatrix}\mathbf{u}_i \\ \mathbf{u}_i \times \mathbf{d}_i\end{bmatrix} + \begin{bmatrix}\boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i \hat{\mathbf{u}}_i \\ \boldsymbol{\omega}_{\lambda(i)} \times (\boldsymbol{\omega}_{\lambda(i)} \times \boldsymbol{r}_i) + 2\boldsymbol{\omega}_{\lambda(i)} \times (\dot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i) + \dot{q}_i \hat{\mathbf{u}}_i \times (\dot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i)\end{bmatrix}.$$

The three terms in the right hand-side of each acceleration equation are similar between both prismatic and revolute formulations, and indeed, each have a simpler term associated with them such that ultimately, both the prismatic and revolute formulations can be expressed by a single equation:

1. The left terms are the spatial accelerations of the parent body as witnessed from the child body's perspective. Therefore the terms are both equal to ${}^{\mathcal{F}_i}\check{\mathbf{X}}_{\mathcal{F}_{\lambda(i)}} \check{\boldsymbol{a}}_{\lambda(i)}$, where ${}^{\mathcal{F}_i}\check{\mathbf{X}}_{\mathcal{F}_{\lambda(i)}}$ is the spatial transformation from the parent frame $\mathcal{F}_{\lambda(i)}$ to the child frame $\mathcal{F}_i$.[7]

2. The middle terms are the ones dependent on the joint variable accelerations $\ddot{q}_i$. The vector in each respective term is called the *spatial joint axis* of the joint, and it is equal to

$$\check{\boldsymbol{s}}_i = \begin{bmatrix}\mathbf{0} \\ \boldsymbol{u}_i\end{bmatrix} \quad \text{and} \quad \check{\boldsymbol{s}}_i = \begin{bmatrix}\boldsymbol{u}_i \\ \boldsymbol{u}_i \times \boldsymbol{d}_i\end{bmatrix}$$

[7] From now on, ${}^{\mathcal{F}_i}\check{\mathbf{X}}_{\mathcal{F}_{\lambda(i)}}$ will instead be denoted as ${}^{i}\check{\mathbf{X}}_{\lambda(i)}$ for simplicity.

for prismatic and revolute joints, respectively. Conceptually, a prismatic joint is really just a 3-D unit vector indicating the allowable direction of translation, whereas a revolute joint is a 3-D unit vector indicating the allowable axis of rotation. In the same vein of thinking, the spatial joint axis is just a 6-D vector indicating both in parallel.

3. The third term involves all the velocity-induced accelerations. It is called the *spatial Coriolis force*, but includes both centripetal and Coriolis effects. If the joint is prismatic, it is equal to

$$\check{\boldsymbol{c}}_i = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{\omega}_{\lambda(i)} \times (\boldsymbol{\omega}_{\lambda(i)} \times \boldsymbol{r}_i) + 2\boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i \hat{\mathbf{u}}_i \end{bmatrix}. \tag{4.16}$$

and if the joint is revolute, it is equal to

$$\check{\boldsymbol{c}}_i = \begin{bmatrix} \boldsymbol{\omega}_{\lambda(i)} \times \dot{q}_i \hat{\mathbf{u}}_i \\ \boldsymbol{\omega}_{\lambda(i)} \times (\boldsymbol{\omega}_{\lambda(i)} \times \boldsymbol{r}_i) + 2\boldsymbol{\omega}_{\lambda(i)} \times (\dot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i) + \dot{q}_i \hat{\mathbf{u}}_i \times (\dot{q}_i \hat{\mathbf{u}}_i \times \boldsymbol{d}_i) \end{bmatrix}. \tag{4.17}$$

Note that this vector effectively describes the acceleration that a body will experience due to the velocities of its ancestors; which is crucial if we are to use articualted-bodies to approximate the complex yet lively motion of plants.

The final equation for the spatial acceleration of a rigid body is thus

$$\check{\boldsymbol{a}}_i = {}^i\check{\mathbf{X}}_{\lambda(i)} \check{\boldsymbol{a}}_{\lambda(i)} + \ddot{q}_i \check{\boldsymbol{s}}_i + \check{\boldsymbol{c}}_i. \tag{4.18}$$

4.4.3 Second pass: the articulated-body equations of motion

The 'articulated-body algorithm' has a bit of a misnomer (hence why I personally prefer to call it Featherstone's algorithm); it is not named such merely because it operates on articulated bodies, which many algorithms do, but rather because its main idea is to construct each simulation step what are called the *articulated-body equations of motion* for each body in the system. An articulated-body equation of motion belongs to a single rigid body, called the *handle*, and it is the equation of motion that the handle *perceives* to have when we temporarily sever it from its parent, and take into complete consideration the dynamical effects that the entire *subtree*[8] of the handle has on it. What this means is that instead of just calculating the Newton-Euler rigid-body equation of motion for each body in the tree,

$$\check{\boldsymbol{f}} = \check{\boldsymbol{I}}\check{\boldsymbol{a}} + \check{\boldsymbol{p}}, \tag{4.19}$$

which is incomplete, the articulated-body algorithm is interested in the calculation of the more complete *articulated-body equation of motion* for each body in the tree,

$$\check{\boldsymbol{f}} = \check{\boldsymbol{I}}^A\check{\boldsymbol{a}} + \check{\boldsymbol{p}}^A. \tag{4.20}$$

In Equation (4.20), $\check{\boldsymbol{I}}^A$ and $\check{\boldsymbol{p}}^A$ are the *articulated-body inertia tensor* and the *articulated-body bias force*, respectively (hence the superscript ^{A}). The articulated-body inertia tensor represents the *apparent* inertia of the handle body if we took into account how the inertia of its descendants affects it [9]. This means that the instantaneous positions and orientations of each body in the handle's subtree contribute to its articulated-body inertia, leading to a system in which each rigid-body typically has an unique articulated-body inertia. Likewise, the articulated-bias force encapsulates the opposing force that the handle body experiences due to the motion of its descendants, i.e., it is the force that would need to be applied at

[8] A subtree of a body is the articulated body consisting of all the descendants of the body, with the body acting as the base

[9] Note that the inertia of the handle's ancestors does not directly affect the handle's inertia itself

the handle body to 'cancel out' the motion the body is inheriting from its subtree. Equation (4.19) is incomplete because it assumes the rigid body is in isolation. Equation (4.20) is complete because it does not assume the body is in isolation; it captures the dynamical effects (inertia and velocity-induced accelerations) of the handle's descendants. This idea - combined with the computation of the spatial Coriolis force from last section - are the key reasons why Featherstone's algorithm is a candidate algorithm for producing lively motion that is often neglected in plant animation works.

It is important to note that the articulated-body equation above is only valid if the handle body - the body to whom the equation refers to - is assumed to be temporarily detached from its rigid-body system. It's of course true that the handle body experiences non-inertial effects due to the motion of it's ancestors, however, prior to the computation of $\check{\boldsymbol{I}}^A$ and $\check{\boldsymbol{p}}^A$, the parental bias effects affecting the handle, $\check{\boldsymbol{c}}_i$, will have already been pre-computed in a base-to-tip pass consisting of the equations derived in the previous section (4.4.1).

It is a non-trivial task to compute the bias force[10] of kinematic trees because a domino-like effect of non-inertial terms are accumulated throughout the depths of the handle's subtree. If both $\check{\boldsymbol{I}}^A$ and $\check{\boldsymbol{p}}^A$ are known for any particular body in the system, however, then the acceleration of that body may be calculated directly (assuming that the external force on that body, $\check{\boldsymbol{f}}$, is known).

Deriving the articulated-body inertia and bias force

In this section, we aim to prove the existence of, and derive, the variables $\check{\boldsymbol{I}}^A$ and $\check{\boldsymbol{p}}^A$ for all bodies in a kinematic tree. An algorithm will ultimately be constructed based off the derived equations for $\check{\boldsymbol{I}}^A$ and $\check{\boldsymbol{p}}^A$; the articulated-body algorithm. This problem is summarized as follows:

[10] Also called the ficticious force, pseudo force, and zero-acceleration force, amongst others

Problem: Calculation of articulated-body quantities

Given an abstract kinematic tree with N_B bodies, prove the existence of, and derive, the spatial matrix $\check{\boldsymbol{I}}_i^A$ and spatial vector $\check{\boldsymbol{p}}_i^A$ for all $1 \leq i \leq n$ such that

$$\check{\boldsymbol{f}}_i^I = \check{\boldsymbol{I}}_i^{\check{\boldsymbol{A}}} \check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i^A,$$

where $\check{\boldsymbol{f}}_i^I$ is the spatial force of body i's inboard joint (the joint connecting it to its parent), and $\check{\boldsymbol{a}}_i$ is the unknown output spatial acceleration of body i.

Since it confused me for some time, I thought I'd note that it helped to think about the equation $\check{\boldsymbol{f}}_i^I = \check{\boldsymbol{I}}_i^{\check{\boldsymbol{A}}} \check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i^A$ as saying the following two equivalent things:

1. If the net force my parent is giving me is $\check{\boldsymbol{f}}_i^I$ (resolved at my local coordinate frame), what acceleration $\check{\boldsymbol{a}}_i$ will I experience, taking into account the inertia $\check{\boldsymbol{I}}_i^{\check{\boldsymbol{A}}}$ and bias force $\check{\boldsymbol{p}}_i^A$ that I *perceive* to have due to my children?

2. If I currently have acceleration $\check{\boldsymbol{a}}_i$, what force $\check{\boldsymbol{f}}_i^I$ will I propagate to my parent, taking fully into account the influences of my subtree?

This calls for an inductive proof over the kinematic tree, but this time starting at the leaves of the tree and recursively working our way down to the base. This can be achieved by finding a recurrence relationship similar to what we did in Section 4.4.1, but this time having the following form:

Base case: $\check{\boldsymbol{I}}_i^A$ and $\check{\boldsymbol{p}}_i^A$ is known if i is the index of a leaf of the kinematic tree.

Recursive case: if i is not the index of a leaf, then $\boldsymbol{I}_i^{\check{\boldsymbol{A}}}$ and $\check{\boldsymbol{p}}_i^A$ can be stated in terms of all $\check{\boldsymbol{I}}^A{}_j$ and $\check{\boldsymbol{p}}_j^A$ such that $j \in \mu(i)$ (recall $\mu(i)$ contains all children of body i).

In other words, we need to prove that $\check{\boldsymbol{I}}_i^A$ and $\check{\boldsymbol{p}}_i^A$ exist trivially for the leaves of the kinematic-tree, and we need to prove that any other body's $\check{\boldsymbol{I}}_i^A$ and $\check{\boldsymbol{p}}_i^A$ can be stated in terms of its children.

Base case

If i is a leaf body of the kinematic tree, then there are no children attached to it and thus its articulated inertia equals its isolated inertia, $\check{\boldsymbol{I}}_i^{\boldsymbol{A}} = \check{\boldsymbol{I}}_i$. Likewise, if i is a leaf body, there are no outbound attached bodies to contribute towards i's articulated-body bias force, so we may also conclude that the articulated-body bias force for a leaf body is also its isolated counterpart, $\check{\boldsymbol{p}}_i^A = \check{\boldsymbol{p}}_i$. Therefore, if i is a leaf body,

$$\check{\boldsymbol{f}}_i^I = \check{\boldsymbol{I}}_i^A \check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i^A = \check{\boldsymbol{I}}_i \check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i,$$

meaning that $\check{\boldsymbol{I}}_i^A$ and $\check{\boldsymbol{p}}_i^A$ always exist for the base case. Note that the equation above is just the spatial formulation of the Newton-Euler equations as seen in Example 2.

Inductive case

For the inductive case, our inductive hypothesis will be to assume that the theorem holds on all the children of body i, that is, there exists an $\check{\boldsymbol{I}}_j^A$ and $\check{\boldsymbol{p}}_j^A$ such that

$$\check{\boldsymbol{f}}_j^I = \check{\boldsymbol{I}}_j^A \check{\boldsymbol{a}}_j + \check{\boldsymbol{p}}_j^A \quad \forall j \in \mu(i).$$

The idea will be to start with the canonical Newton-Euler equation for body i, and, through a set of derivations, to reach the desired equation required to fulfill the proof:

$$\check{\boldsymbol{f}}_i = \check{\boldsymbol{I}}_i \check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i \xrightarrow{\textit{derivation}} \check{\boldsymbol{f}}_i^I = \check{\boldsymbol{I}}_i^A \check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i^A$$

If solved, the derivation will ultimately yield the equations for $\check{\boldsymbol{I}}_i^A$ and $\check{\boldsymbol{p}}_i^A$. First, we note that $\check{\boldsymbol{f}}_i$ is composed of two different types of forces acting on body i:

1. The force from the single inboard joint $\check{\boldsymbol{f}}_i^I$. The inboard joint is the one that connects body i to its parent.
2. The sum of forces of the outboard joints, $\sum_{j\in\mu(i)} \check{\boldsymbol{f}}_i^{O_j}$. The outboard joints are the ones connecting body i to its children.

Note that we will assume that all of these forces are expressed (or resolved) at body i's coordinate frame, that is, a linear component along the center of mass, and an angular component about the center of mass. This means that we can plug in these forces into the isolated Newton-Euler equation for body i which is expressed in the reference frame of i:

$$\check{\boldsymbol{f}}_i = \check{\boldsymbol{f}}_i^I + \sum_{j\in\mu(i)} \check{\boldsymbol{f}}_i^{O_j} = \check{\boldsymbol{I}}_i\check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i,$$

and rearranging:

$$\check{\boldsymbol{f}}_i^I = \check{\boldsymbol{I}}_i\check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i - \sum_{j\in\mu(i)} \check{\boldsymbol{f}}_i^{O_j}.$$

We have now matched the left-hand-side of the equation ($\check{\boldsymbol{f}}_i^I$) to what is desired. To proceed, we note that by Newton's third law, the outboard force between two bodies must be equal and opposite to the inboard force between those same two bodies, as long as both forces are expressed in the same reference frame. The outboard forces of body i are thus actually negative the inboard forces of its children!

$$\check{\boldsymbol{f}}_i^{O_j} = -\,{}^i\check{\boldsymbol{X}}_j\check{\boldsymbol{f}}_j^I.$$

We can now combine the two above equations as follows:

$$\check{\boldsymbol{f}}_i^I = \check{\boldsymbol{I}}_i\check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i + \sum_{j\in\mu(i)} {}^i\check{\boldsymbol{X}}_j\check{\boldsymbol{f}}_j^I.$$

The child forces $\check{\boldsymbol{f}}_j^I$ are not trivial because they are influenced by all the bodies in j's

respective subtree. However, by the inductive hypothesis, we have stated that we already know $\check{\boldsymbol{f}}_j^I = \check{\boldsymbol{I}}_j^A \check{\boldsymbol{a}}_j + \check{\boldsymbol{p}}_j^A$. Therefore, we can state the above equation as:

$$\check{\boldsymbol{f}}_i^I = \check{\boldsymbol{I}}_i \check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i + \sum_{j \in \mu(i)} {}^i\check{\boldsymbol{X}}_j (\check{\boldsymbol{I}}_j^A \check{\boldsymbol{a}}_j + \check{\boldsymbol{p}}_j^A).$$

The next step is to note that we can express the spatial acceleration of a child in terms of the spatial acceleration of its parent plus the spatial acceleration of the joint connecting them. This is done using the previously derived equations for acceleration propagation in Equation (4.18):

$$\check{\boldsymbol{a}}_j = {}^j\check{\mathbf{X}}_i \check{\boldsymbol{a}}_i + \ddot{q}_j \check{\boldsymbol{s}}_j + \check{\boldsymbol{c}}_j.$$

Plugging this into our current equation for $\check{\boldsymbol{f}}_i^I$, we get,

$$\check{\boldsymbol{f}}_i^I = \check{\boldsymbol{I}}_i \check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i + \sum_{j \in \mu(i)} {}^i\check{\boldsymbol{X}}_j \left[\check{\boldsymbol{I}}_j^A ({}^j\check{\mathbf{X}}_i \check{\boldsymbol{a}}_i + \ddot{q}_j \check{\boldsymbol{s}}_j + \check{\boldsymbol{c}}_j) + \check{\boldsymbol{p}}_j^A \right],$$

which can be re-arranged to group up common terms like so:

$$\check{\boldsymbol{f}}_i^I = \left(\check{\boldsymbol{I}}_i + \sum_{j \in \mu(i)} {}^i\check{\boldsymbol{X}}_j \; \check{\boldsymbol{I}}_j^A \; {}^j\check{\boldsymbol{X}}_i \right) \check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i + \sum_{j \in \mu(i)} {}^i\check{\boldsymbol{X}}_j \left[\check{\boldsymbol{p}}_j^A + \check{\boldsymbol{I}}_j^A \check{\boldsymbol{c}}_j + \left(\check{\boldsymbol{I}}_j^A \check{\boldsymbol{s}}_j \right) \ddot{q}_j \right]. \tag{4.21}$$

In order to continue with the inductive proof over Equation (4.21), we note that the remaining unknown is $\ddot{q}_j$. It can be addressed by making use of the following term:

$$Q_j = \begin{bmatrix} s_{j_1} & \cdots & s_{j_6} \end{bmatrix} \begin{bmatrix} f_{j_1} \\ \vdots \\ f_{j_6} \end{bmatrix} = \check{\boldsymbol{s}}_j^T \check{\boldsymbol{f}}_j^I.$$

This will take the dot product of the spatial joint axis with the spatial inboard force, and will return the *magnitude* of the force exerted through the joint in question. We continue

by substituting the acceleration propagation of all j's into their respective articulated-body equations,

$$\check{\boldsymbol{f}}_j = \check{\boldsymbol{I}}_j^A({}^j\check{\mathbf{X}}_i\check{\boldsymbol{a}}_i + \ddot{q}_j\check{\boldsymbol{s}}_j + \check{\boldsymbol{c}}_j) + \check{\boldsymbol{p}}_j^A,$$

and pre-multiplying both sides by $\check{\boldsymbol{s}}_j^T$ to make use of Q_j,

$$\begin{aligned}\check{\boldsymbol{s}}_j^T\check{\boldsymbol{f}}_j &= \check{\boldsymbol{s}}_j^T\left[\check{\boldsymbol{I}}_j^A({}^j\check{\mathbf{X}}_i\check{\boldsymbol{a}}_i + \ddot{q}_j\check{\boldsymbol{s}}_j + \check{\boldsymbol{c}}_j) + \check{\boldsymbol{p}}_j^A\right]\\ &= Q_j = \check{\boldsymbol{s}}_j^T\check{\boldsymbol{I}}_j^A({}^j\check{\mathbf{X}}_i\check{\boldsymbol{a}}_i + \ddot{q}_j\check{\boldsymbol{s}}_j + \check{\boldsymbol{c}}_j) + \check{\boldsymbol{s}}_j^T\check{\boldsymbol{p}}_j^A,\end{aligned}$$

which can now be used to find an equation for $\ddot{q}_j$:

$$\ddot{q}_j = \frac{Q_j - \check{\boldsymbol{s}}_j^T\ \check{\boldsymbol{I}}_j^A\ {}^j\check{\mathbf{X}}_i\check{\boldsymbol{a}}_i - \check{\boldsymbol{s}}_j^T\left(\check{\boldsymbol{p}}_j^A + \check{\boldsymbol{I}}_j^A\check{\boldsymbol{c}}_j\right)}{\check{\boldsymbol{s}}_j^T\check{\boldsymbol{I}}_j^A\check{\boldsymbol{s}}_j}. \tag{4.22}$$

We can now plug in this equation for $\ddot{q}_j$ into our current equation for $\check{\boldsymbol{f}}_i^I$ (4.21) and rearragnge to couple up common terms:

$$\begin{aligned}\check{\boldsymbol{f}}_i^I = &\left[\check{\boldsymbol{I}}_i + \sum_{j\in\mu(i)} {}^i\check{\boldsymbol{X}}_j\left(\check{\boldsymbol{I}}_j^A - \frac{\check{\boldsymbol{I}}_j^A\check{\boldsymbol{s}}_j\check{\boldsymbol{s}}_j^T\check{\boldsymbol{I}}_j^A}{\check{\boldsymbol{s}}_j^T\check{\boldsymbol{I}}_j^A\check{\boldsymbol{s}}_j}\right){}^j\check{\boldsymbol{X}}_i\right]\check{\boldsymbol{a}}_i\\ &+ \check{\boldsymbol{p}}_i + \sum_{j\in\mu(i)} {}^i\check{\boldsymbol{X}}_j\left[\check{\boldsymbol{p}}_j^A + \check{\boldsymbol{I}}_j^A\check{\boldsymbol{c}}_j + \frac{\check{\boldsymbol{I}}_j^A\check{\boldsymbol{s}}_j\left[Q_j - \check{\boldsymbol{s}}_j^T\left(\check{\boldsymbol{p}}_j^A + \check{\boldsymbol{I}}_j^A\check{\boldsymbol{c}}_j\right)\right]}{\check{\boldsymbol{s}}_j^T\check{\boldsymbol{I}}_j^A\check{\boldsymbol{s}}_j}\right].\end{aligned} \tag{4.23}$$

Note that this equation has the desired form

$$\check{\boldsymbol{f}}_i^I = \check{\boldsymbol{I}}_i^A\check{\boldsymbol{a}}_i + \check{\boldsymbol{p}}_i^A,$$

meaning Equation (4.23) has yielded the coefficients necessary to complete the proof:

$$\check{\boldsymbol{I}}_i^A = \check{\boldsymbol{I}}_i + \sum_{j\in\mu(i)} {}^i\check{\boldsymbol{X}}_j\left(\check{\boldsymbol{I}}_j^A - \frac{\check{\boldsymbol{I}}_j^A\check{\boldsymbol{s}}_j\check{\boldsymbol{s}}_j^T\check{\boldsymbol{I}}_j^A}{\check{\boldsymbol{s}}_j^T\check{\boldsymbol{I}}_j^A\check{\boldsymbol{s}}_j}\right){}^j\check{\boldsymbol{X}}_i \quad \text{and} \tag{4.24}$$

$$\check{\boldsymbol{p}}_i^A = \check{\boldsymbol{p}}_i + \sum_{j\in\mu(i)} {}^i\check{\boldsymbol{X}}_j\left[\check{\boldsymbol{p}}_j^A + \check{\boldsymbol{I}}_j^A\check{\boldsymbol{c}}_j + \frac{\check{\boldsymbol{I}}_j^A\check{\boldsymbol{s}}_j\left[Q_j - \check{\boldsymbol{s}}_j^T\left(\check{\boldsymbol{p}}_j^A + \check{\boldsymbol{I}}_j^A\check{\boldsymbol{c}}_j\right)\right]}{\check{\boldsymbol{s}}_j^T\check{\boldsymbol{I}}_j^A\check{\boldsymbol{s}}_j}\right]\square \tag{4.25}$$

4.4.4 Third pass: solving for accelerations

The third pass consists of computing the accelerations of all the joints and bodies of the system. This has been trivialized because at this point we have the articulated inertias and articulated bias forces for each body in the system:

$$\begin{aligned}\ddot{q}_i &= \frac{Q_i - \check{\boldsymbol{s}}_i^T\ \check{\boldsymbol{I}}_i^A\ {}^i\check{\mathbf{X}}_{\lambda(i)}\check{\boldsymbol{a}}_{\lambda(i)} - \check{\boldsymbol{s}}_i^T\left(\check{\boldsymbol{p}}_i^A + \check{\boldsymbol{I}}_i^A\check{\boldsymbol{c}}_i\right)}{\check{\boldsymbol{s}}_i^T\check{\boldsymbol{I}}_i^A\check{\boldsymbol{s}}_i}. \\ \check{\boldsymbol{a}}_i &= {}^i\check{\mathbf{X}}_{\lambda(i)}\check{\boldsymbol{a}}_{\lambda(i)} + \ddot{q}_i\check{\boldsymbol{s}}_i + \check{\boldsymbol{c}}_i.\end{aligned} \tag{4.26}$$

A final pseudocode for the third pass, as well as the first and second passes over the kinematic tree, is presented in the following section.

4.4.5 Final algorithm

The pseudocode for the final algorithm is provided in Algorithm 1. There are some things to point out with the algorithm:

1. N_B is the number of bodies in the system.

2. In lines 6 and 7, we initiate the articulated-body quantities to their isolated counter parts.

3. In line 7, we add negative the external force to $\check{\boldsymbol{p}}_i$ because external forces may be treated as non-inertial forces, we just need to define it with respect to i's frame. Any forces generated due to user interaction (pulling, wind, etc.) are handled here.

4. In line 15, we set the acceleration of the root body to be the opposite of gravity. This acceleration will be propagated to the rest of the system as a non-inertial acceleration. This means that the rest of the bodies will experience it as normal gravity!

Algorithm 1 Articulated Body Algorithm

1: **procedure** ArticulatedBodyAlgorithm()
2:
3: **for** $i = 1$ to N_b **do** ▷ First pass, from base to leaves
4: $\check{\mathbf{v}}_i =$ Equation (4.12) or (4.13)
5: $\check{\boldsymbol{c}}_i =$ Equation (4.17) or (4.16)
6: $\check{\boldsymbol{I}}_i^A = \check{\boldsymbol{I}}_i$
7: $\check{\boldsymbol{p}}_i^A = \check{\boldsymbol{p}}_i - {}^i\check{\boldsymbol{X}}_0 \check{\mathbf{f}}_i^x$ /*add external force */
8: **end for**
9:
10: **for** $i = N_b$ to 1 **do** ▷ Second pass, from leaves to base
11: $\check{\boldsymbol{I}}_i^A + =$ Equation (4.24) minus $\check{\boldsymbol{I}}_i$
12: $\check{\boldsymbol{p}}_i^A + =$ Equation (4.25) minus $\check{\boldsymbol{p}}_i$
13: **end for**
14:
15: $\check{\mathbf{a}}_0 = -\check{\mathbf{g}}$
16: **for** $i = 1$ to N_B **do** ▷ Third pass, from base to leaves
17: $\ddot{q}_i = \frac{Q_i - \check{\boldsymbol{s}}_i^T \ \check{\boldsymbol{I}}_i^A \ {}^i\check{\mathbf{X}}_{\lambda(i)} \check{\boldsymbol{a}}_{\lambda(i)} - \check{\boldsymbol{s}}_i^T \left(\check{\boldsymbol{p}}_i^A + \check{\boldsymbol{I}}_i^A \check{\boldsymbol{c}}_i \right)}{\check{\boldsymbol{s}}_i^T \check{\boldsymbol{I}}_i^A \check{\boldsymbol{s}}_i}$
18: $\check{\boldsymbol{a}}_i = {}^i\check{\mathbf{X}}_{\lambda(i)} \check{\boldsymbol{a}}_{\lambda(i)} + \ddot{q}_i \check{\boldsymbol{s}}_i + \check{\boldsymbol{c}}_i.$
19: **end for**
20:
21: **end procedure**
22:

Chapter 5

L-Systems

The previous chapter discussed the forward dynamics problem of an abstract articulated body. This chapter introduces modeling with L-systems, and how one can model an L-system object such that it explicitly represents an articulated body.

Section 5.1 introduces the elementary elements of L-systems that were used in this thesis, and Section 5.2 provides a generalized overview of how one can create explicit articulated bodies with L-systems. A reader well-versed in L-systems may find that they can skip Section 5.1.

5.1 L-systems

Lindenmayer systems, or L-systems for short, are formal grammars that excel at representing *self-similar* structures. A self similar structure is one that is exactly or approximately similar to a part of itself. Such structures are abundant in nature, and to name a few, can be witnessed in the geometry of snowflakes, the patterning of branches, leaves, and inflorescences, and even in macroscopic scales such as the arrangement of rivers and mountain ranges.

The central concept of L-systems is that of *rewriting*, which is the replacement of parts of a simple initial object using a set of *rewriting rules* or *productions* [69, Chapter 1]. If one uses these rewriting rules over several iterations, then a simple original object can turn into a beautifully intricate self-similar structure.

L-systems were conceived as a means to model multicellular life, and originated in Aristid Lindenmayer's work in 1968 [46], where he developed the essential elements of L-systems [47]. Since then, L-systems have been built upon to support the modeling of more complex structures. The types of L-systems used in this thesis would fall under *bracketed parametric L-systems*, whose description is the main focus of this section. A more in-depth description of the different types of L-systems and their applications in botanical image synthesis is presented, for example, in *The Algorithmic Beauty of Plants* [69].

5.1.1 0L-systems

0L-systems are the simplest types of L-systems necessary to model self-similar structures, and were formally introduced by Lindenmayer in [47]. An 0L-system is an ordered triplet,

$$G = < V, \omega, P >, \tag{5.1}$$

where

1. V is a nonempty set of letters called the the *alphabet* of the system.
2. $\omega \in V^+$ is a non-empty word called the *axiom*.
3. $P \subset (V \times V^*)$ is a finite *set of productions*.

The alphabet V represents all the possible *symbols* that an L-system *word* (or *string*) may be comprised of for a particular object at an instant in time. It is common practice, for example, to have certain symbols in V correspond to distinct organs of the object being modeled. The non-empty string ω simply dictates the starting L-system string of the object in question. Finally, a production is denoted as $(a, \chi) \in P$ and is typically written as $a \rightarrow \chi$, where $a \in V$, and χ is a (possibly empty) word over V (that is, $\chi \in V^*$). In the notation $a \rightarrow \chi$, a is called the *predecessor*, and χ is called the *successor*, of the production.

Once an 0L-system has been defined as above, then starting with the axiom ω, it will attempt to *match* each letter in ω against each production in P in parallel. A production $a \rightarrow \chi$ matches symbol $s \in \omega$ if and only if $s \equiv a$, at which point s is replaced with χ. We say that the L-system has completed a *derivation step* once we have attempted to match each letter, resulting in a new string. It is at this point that the user may decide to perform another derivation step, but this time using the new string as input instead of the axiom.

The '0' in the name 0L-system refers to the fact 0L-systems are **not** *context sensitive*. A context sensitive L-system is one in which a letter's neighbours can potentially determine whether a production succeeds. Context sensitive L-systems may not have been used in the plant simulations presented in this thesis - and are hence not described in this chapter - but they are definitely supported by the presented simulation methodology.

5.1.2 Parametric 0L-systems

Parametric L-systems extend the original concept of L-systems by associating numerical parameters with the symbols representing plant components. Parametric L-systems operate on *parametric words*, which are strings of modules consisting of letters with associated *parameters*. The letters belong to the alphabet V - which is the same as with 0L-systems -

but each letter is now accompanied by a finite number of parameters belonging to the set of *real numbers* $\mathbb{R}$. A module with letter $A \in V$ and parameters $a_1, a_2, ..., a_n \in \mathbb{R}$ is denoted as $A(a_1, a_2, ..., a_n)$. Formally, a *parametric 0L-system* is defined as an ordered quadruplet,

$$G =< V, \Sigma, \omega, P >, \tag{5.2}$$

where

1. V is a nonempty set of letters called the the *alphabet* of the system

2. Σ is the *set of formal parameters*

3. $\omega \in (V \times \mathbb{R}^*)^+$ is a non-empty parametric word called the *axiom*

4. $P \subset (V \times \Sigma^*) \times \mathcal{C}(\Sigma) \times (V \times \mathcal{E}(\Sigma)^*)^*$ is a finite set of *productions*. Noting that Σ is the set of formal parameters present in the L-system, then $\mathcal{C}(\Sigma)$ is the set of all correctly constructed *logical* expressions operating on Σ, and $\mathcal{E}(\Sigma)$ is the set of all correctly constructed *arithmetic* expressions operating on Σ.

A production in a parametric 0L-system is denoted as $(a, C, \chi) \in P$ and is typically written as $a : C \rightarrow \chi$. This is identical to the definition of the non-parametric productions given above, except for two further caveats:

1. The number of real-valued *actual* parameters of the letter being matched has to be the same as the number of *formal* parameters in the production's predecessor (that is, the module `A(0.1, 0.5)` would not match with a production looking for modules of form `A(a,b,c)`).

2. The logical expression $C \in \mathcal{C}(\Sigma)$ is called the *condition*; the condition is a boolean-valued function of the parameters in a. The production only succeeds if C evaluates to **true**. The condition can be empty, in which case the production can again be denoted as $a \rightarrow \chi$.

If the two extra conditions are met, then the production succeeds on the letter in question, and a is replaced with χ. The rest of the logic between parametric and non-parametric 0L-systems is the same.

5.1.3 Turtle interpretation of L-system strings

The description of parametric 0L-system presented thus far suffices to mathematically represent a plethora of self-similar objects, however, it is missing a way in which to graphically represent them. In order to do so, we can use *turtle geometry* [1] as a means to graphically represent L-system words [67, 66]. The logic here is to give a graphical representation of each symbol in the word its own position and orientation with respect to some global reference frame $\mathcal{F}_{\mathcal{O}}$, which is done by using a *turtle* to keep track of positions and orientations with respect to the previous symbol of the word.

A turtle is formally defined to be a position vector $\mathbf{p} = (x, y, z)^T$, and three mutually orthonormal vectors, $\hat{\mathbf{H}}, \hat{\mathbf{L}}, \hat{\mathbf{U}}$, which indicate the turtle's *heading*, its direction to its *left*, and its direction *up*, respectively (note how this also defines a coordinate frame as used by Featherstone's algorithm). These three vectors together represent the turtle's current *orientation*. The position and orientation together are called the turtle's *pose*. The active rotation of a turtle from an orientation, $[\,\hat{\mathbf{H}}\ \ \hat{\mathbf{L}}\ \ \hat{\mathbf{U}}\,]$, to a new one, $[\,\hat{\mathbf{H}}'\ \ \hat{\mathbf{L}}'\ \ \hat{\mathbf{U}}'\,]$, can be expressed by the formula:

$$[\,\hat{\mathbf{H}}'\ \ \hat{\mathbf{L}}'\ \ \hat{\mathbf{U}}'\,] = [\,\hat{\mathbf{H}}\ \ \hat{\mathbf{L}}\ \ \hat{\mathbf{U}}\,]\,\mathbf{R},$$

where $\mathbf{R}$ is a 3×3 rotation matrix. The rotations along any of the $\hat{\mathbf{H}}$, $\hat{\mathbf{L}}$, and $\hat{\mathbf{U}}$ vectors can be represented by the following matrices:

$$\mathbf{R}_{\mathrm{U}}(\alpha) = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_{\mathrm{L}}(\alpha) = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

$$\mathbf{R}_{\mathrm{H}}(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$

The idea is to define a starting position and orientation of the L-system's turtle, and to introduce symbols to the L-system's alphabet, V, that can be used to move and rotate the turtle. We therefore formally introduce the following parametric symbols to the alphabet V:

F(x)	Move forward a step of length x, drawing a line segment between the previous and current turtle positions.
f (x)	Move forward a step of length x without drawing a line segment.
$+(\delta)$	Turn left by angle δ using rotation matrix $\mathbf{R_U}(\delta)$.
$-(\delta)$	Turn right by angle δ using rotation matrix $\mathbf{R_U}(-\delta)$.
$\&(\delta)$	Pitch down by angle δ using rotation matrix $\mathbf{R_L}(\delta)$.
$\wedge(\delta)$	Pitch up by angle δ using rotation matrix $\mathbf{R_L}(-\delta)$.
$\backslash(\delta)$	Roll left by angle δ using rotation matrix $\mathbf{R_H}(\delta)$.

$/(\delta)$ Roll right by angle δ using rotation matrix $\mathbf{R_H}(-\delta)$.

$|(\delta)$ Turn around using rotation matrix $\mathbf{R_U}(180°)$.

5.1.4 Bracketed 0L-systems

The parametric 0L-systems described thus far can only model organisms with non-branching architectures because the letters in a string are arranged in a strictly linear fashion. In order to represent branching structures, we can use the concept of *strings with brackets*. A bracketed string is one in which left and right brackets, '[' and ']', are added to the alphabet, V, of the L-system in question. Any sub-string $u \in V^+$ in the L-system string enclosed by two brackets is a branch. Note that u may have sub-branches of its own.

Turtle geometry can be easily accommodated into a bracketed L-system environment by introducing the following functionality to the symbols '[' and ']':

[Push the current state of the turtle onto a pushdown stack. The information saved on the stack contains the turtle's position and orientation.

] Pop a state from the stack and make it the current state of the turtle.

5.2 Articulated-body L-systems

The main premise behind the methodology presented in this thesis is that L-systems are ideal in representing explicit structures of articulated bodies, whose dynamics problem can then be simulated by Featherstone's algorithm. The bare-bones description of bracketed parametric L-systems provided in this chapter suffices to create these types of L-systems, which will be referred to as *Articulated-body L-systems*.

Technically speaking, all L-systems are discrete branching structures and thus they may all be thought of as representing articulated bodies. However, by articulated-body L-systems, we are instead referring to L-systems modeled in a strict fashion such that at any time during its time evolution, the L-system string of an articulated-body L-system must *explicitly* contain the following information:

1. The physical properties of each individual rigid-body in the system (mass, center of mass, inertia tensor)
2. The physical properties of each individual joint in the system (type of joint, default value of joint variables(s)).
3. The connectivity of the system (what is connected to what? where does a joint connect relative to its adjacent bodies?).

An articulated body is a collection of rigid bodies that may be connected together by joints. Logically, this implies that the L-system should be composed of modules corresponding to rigid bodies and modules corresponding to joints. However, we can't just mash rigid-body and joint modules into a string in a random order: the underlying structure would likely not conform to the articulated-body structure required by Featherstone's algorithm as described in Section 4.3. Additionally, what exactly should the information stored in each rigid-body and joint module be?

This section is thus concerned with describing the structure of articulated-body L-systems as employed in this thesis, which are L-systems designed such that - after any number of derivation steps - the resulting structure of the L-system string will conform to the formal description of articulated-bodies as provided in Section 4.3.

It should be noted that the modeling program used in thesis was L-studio 4.0 [41], which is a Windows version of the *virtual laboratory* [27] plant modeling suite. However, the only thing

of importance is that there be an accordance between the output of the modeling program and the input of the external physics program. This means that we are primarily concerned with the *structure* of the output L-system string, and not the specific modeling method nor the software that was used to create it.

5.2.1 Structure of articulated-body L-system strings

We only need two definitions from which to infer the permissible structures of articulated-body L-systems. The first definition belongs to articulated-bodies, whereas the second definition belongs to L-systems:

1. A joint represents a connection between exactly two rigid bodies.
2. The location of a module in an L-system string defines its connectivity.

By taking definitions (1) and (2) above we can deduce that a joint module `J` must **always** have a body module `B` immediately to its 'left' and immediately to its 'right' in the L-system string. This condition leads to the three following rules:

1. We cannot have two consecutive joint modules in the string (`BJJB`, `BJ[JB]`, `B[J]J`).

    Note that the spherical joints we will be using *are* modeled as three consecutive revolute joints, but each of these revolute joints are technically connected via volume-less rigid bodies (even then, this is handled within the physics library meaning we do not need to consider this when constructing the L-system).

2. We cannot have two consecutive body modules in the string (`BB`, `B[B]`).

    The reason for this is because the L-system is saying 'there are two modules next to each other, therefore they are connected', however, Featherstone's algorithm is saying 'they can't be connected because there is no joint between them', meaning

there is a conflict between definitions. One *could* interpret this as the two rigid bodies being glued together by a 0-DoF joint, but such assumptions were not carried out in this thesis.

3. It is not possible to branch immediately after a joint module.

   This is because branching immediately after a joint module will always either bring forth two consecutive joint modules (`BJ[JB]B`, `BJ[B]JB`) or associate a joint as a connection between three bodies (`BJ[B]B`), neither of which are allowed. Note that we are assuming the absence of 'needless branching' (e.g. the string `BJ[B]`).

The following are thus examples of permissible strings:

`BJB`,

`BJBJB`,

`B[JB]JB`,

`B[JB[JB]JB]JB`.

Notice how all permissible strings start with a body module; it corresponds to the immovable root body in Featherstone's algorithm. The only practical reason why we need to include it in the string is because - as will be explained in Chapter 6 - every joint needs two adjacent bodies in order to compute its spring constant. The base body therefore represents the immovable section of the plant's stem that is buried in the ground. The immovable joint j_0, however, is not needed in my implementation and is thus omitted from my articulated-body L-system interpretation.

Everything described thus far boils down to the following definition for the *structure* of an articulated-body L-system:

> **Definition of Articulated-body L-system structure:** An articulated-body L-system is composed of modules `B` representing rigid bodies, and modules `J`

representing joints. At any point in time, all base-to-tip or tip-to-base traversals of the articulated-body L-system must start with a B module which may then be followed by any number of JB pairs.

Another way of putting it is that all strings representing a single base-to-tip or tip-to-base traversal of the tree - such that they are unbranched strings - must match the regular expression B(JB)*. It should be pointed out that the indexing scheme required by Featherstone's algorithm is readily available by the above structure. This is because a left-to-right traversal of any proper articulated-body L-system string will always start with a B module corresponding to the root body with index 0, and will always be followed by any number of JB pairs sharing the same index:

$$\mathtt{B}_0,$$

$$\mathtt{B}_0\mathtt{J}_1\mathtt{B}_1,$$

$$\mathtt{B}_0\mathtt{J}_1\mathtt{B}_1\mathtt{J}_2\mathtt{B}_2,$$

$$\mathtt{B}_0\,[\mathtt{J}_1\mathtt{B}_1]\,\mathtt{J}_2\mathtt{B}_2,$$

$$\mathtt{B}_0\,[\mathtt{J}_1\mathtt{B}_1\,[\mathtt{J}_2\mathtt{B}_2]\,\mathtt{J}_3\mathtt{B}_3]\,\mathtt{J}_4\mathtt{B}_4.$$

Finally, it should be stated that articulated-body L-systems *are* allowed to contain any other types of modules *and* they may be located wherever the modeler desires; there will be no conflicts as long as none of these modules represent rigid-bodies nor joints.

We have effectively discussed the permissible structures of articulated-body L-system strings, but have provided no details regarding the formal parameters that each rigid body and joint module must have. These parameters are immediately discussed.

5.2.2 Parameters of rigid-body and joint modules

In theory, we may use any types of joints to connect two rigid bodies (revolute, prismatic, spherical, planar, etc.). After all, Featherstone's algorithm as derived in Chapter 4 makes no

assumptions on joint types. This is because any desired joint may be modeled as a composition of prismatic and revolute joints that are each connected via volume-less rigid bodies. In practice, however, we are using articulated-bodies to approximate the dynamics of plants and thus only need to capture the relative rotations between connected rigid bodies. This means that all joints will be spherical, and thus any mention of a joint hereon will assume a spherical joint.

With this assumption at hand, we are now interested in figuring out the parameters that belong to rigid-body modules versus the parameters that belong to (spherical) joint modules. Certain parameters are obvious (e.g. a rigid-body should have a parameter indicating its mass, each joint should contain the default values of its joint variables), whereas others are not.

The main issue in deciding these parameters arises from the geometrical quantities required by Featherstone's algorithm. We recall that every joint and rigid body has a local reference frame embedded within it, and additionally, that a body's frame is located at the body's center of mass and is oriented according to the body's principle axes of inertia (Figure 5.1). Since Featherstone's algorithm is all about information transfer, then it is required that all spatial transformations between connected frames be obtainable from the parameters present in the rigid-body and joint modules. It is not obvious, however, the parameters that each joint and body module must have such that the transformations between reference frames are adequately represented. Keep in mind that throughout all of this, we are trying to maintain the important notion of *locality*, which means that a module should only contain information that 'belongs' to itself.

We can derive these parameters by analyzing how turtle geometry is used to graphically represent L-systems; a turtle represents a coordinate frame, and the symbols controlling a

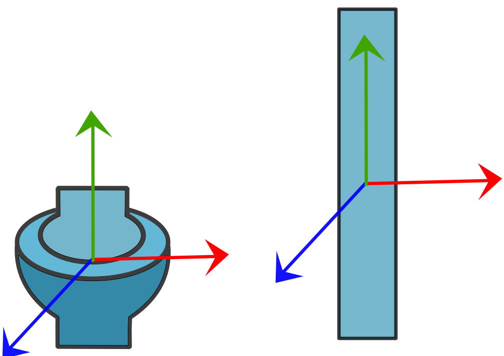


Figure 5.1: The local reference frames of rigid bodies and joints.

turtle either represent a change in position (`F`, `f`) or a change in orientation (`+`, `-`, `&`, `^`, \, `/`, `|`). There is thus a nice analogy present here: rigid-body modules correspond to changes in positions (similar to `F` and `f`), whereas spherical joints correspond to changes in orientation (`+`, `-`, `&`, `^`, \, `/`, `|`). This means that a rigid-body module should contain the information required to translate between its inboard joint and each of its outboard joints, whereas a joint module should contain the information required to rotate between the two bodies it connects. For simplicity, I will assume that all outboard joints connect at the same location within a body's local reference frame[1].

We can thus deduce that a rigid-body module must have the following parameters:

1. One scalar denoting the mass of the rigid body.
2. Three scalars denoting the translation from the inboard joint to the location of the center of mass.

   They should be given with respect to the inboard joint's frame.

3. Three scalars denoting the translation from the center of mass to the location of the outboard joints.

[1]There is no 'theoretical' reason why we must assume this; its just cleaner to present (and implement) a paradigm in which each body has a single 'socket' where its inboard joint connects and a single 'socket' where all its outboard joints connect.

They should be given with respect to the center of mass frame. Note how the three scalars from (2) plus the three scalars from (3) denote the translation from the inboard joint to the outboard joint(s).

4. Three scalars denoting the diagonal elements of the body's moment of inertia tensor about the body's **inboard joint frame**.

   The inertia tensor has to be given with respect to the axes of the inboard joint frame because those are the axes about which the body rotates. We can find such an inertia tensor by first computing it about the body's center of mass and then translating it to the inboard joint location via the parallel axis theorem (this works because the principle-axes-aligned center of mass frame and the inboard joint's frame always share the same orientation, meaning that the inertia tensor of the body about the inboard joint's frame will also be diagonal).

We can also deduce that a (spherical) joint module must have the following parameters:

1. Three scalars denoting the rest angles of the spherical joint in each direction of rotation. This is how branching angles are specified.

2. Three scalars denoting the current angles (relative to the rest angles) of the spherical joint in each direction of rotation (these will act as the initial generalized coordinates, $\mathbf{q}$, of the system). Note how the three scalars from (1) plus the three scalars from (2) fully describe the rotation imposed by the joint.

3. Three scalars denoting the spring constants of the spherical joint in each direction of rotation. The chapter that follows (Chapter 6) will describe how these spring constants were derived in this thesis.

5.2.3 Note on small rigid bodies

When we export our L-system string into our Featherstone's simulator, we are essentially taking a 'snapshot' of a developmental model at some point in its lifetime. It could be the case that this snapshot will include infantile modules that had just been created at the time of the snapshot; these modules, if turned into rigid bodies, may have a volume and/or mass so tiny such that they may introduce numerical instabilities into the physical simulation. It was therefore found useful to prune these infantile modules (with respect to some user-defined threshold) in a post-processing step before feeding the string into the dynamics library.

Chapter 6

Representing continuous branching structures with articulated bodies

At this point we have looked at the following topics:

1. How to simulate the dynamics of articulated bodies forward in time using the articulated-body algorithm (Chapter 4).

2. How to construct 'articulated-body L-systems', which are L-systems that generate structures of articulated bodies such that they can be simulated by the articulated-body algorithm (Chapter 5).

This chapter is concerned with the last piece of the puzzle, which is

3. How to make these articulated-body simulations move like real plants.

In order to do so, we need to look at the equations behind real-life plant motion, and how we can best go about approximating them with articulated-body dynamics. Section 6.1 provides a succinct summary of the discretized deformation scheme employed in this thesis, and poses the main problem to be addressed in the chapter. Section 6.2 reviews the basics of deformation with a focus on the deformation types modeled in this thesis

(bending and torsion). Section 6.3 presents continuous and discretized *bending* deformations, whereas Section 6.4 does the same but for *torsional* deformations. Section 6.5 discusses the correctness of representing spherical joints by Euler angles, and finally, Section 6.6 describes the implementation and limitations of damping as employed in this thesis.

6.1 Summary of deformation paradigm employed

An articulated body is a collection of rigid bodies that may be connected together by joints, where a joint is a connection between exactly two bodies. The body closer to the base is called the joint's *proximal* body whereas the body further from the base is called the joint's *distal* body. We will assume that every joint is spherical as per the discussion in Section 5.2. This leads to the paradigm in which the rigid-bodies encapsulate translations between its adjacent joints, and (spherical) joints encapsulate rotations between its adjacent bodies.

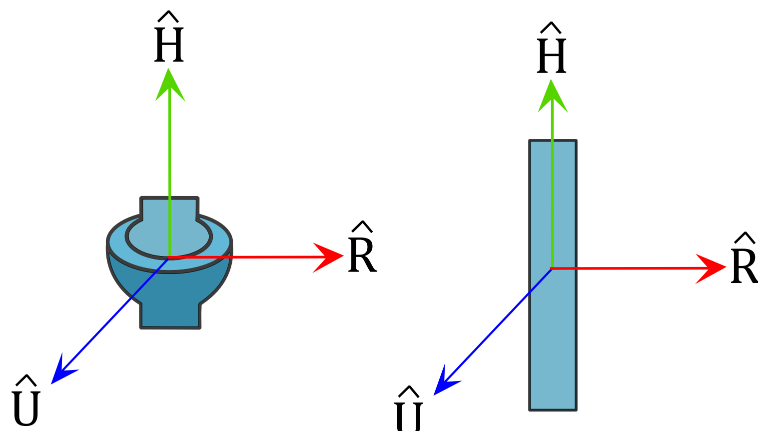


Figure 6.1: The local reference frames of rigid bodies and joints.

Every body and joint in the system shall have a local coordinate frame embedded within it, and we will call its axes $\hat{\mathbf{R}}$, $\hat{\mathbf{H}}$, and $\hat{\mathbf{U}}$ (Figure 6.1). They have identical function to the $\hat{\mathbf{H}}$, $\hat{\mathbf{L}}$, and $\hat{\mathbf{U}}$ frames described in Section 5.1.3, but have been rearranged to more faithfully represent the coordinate frame paradigm carried out in this thesis.

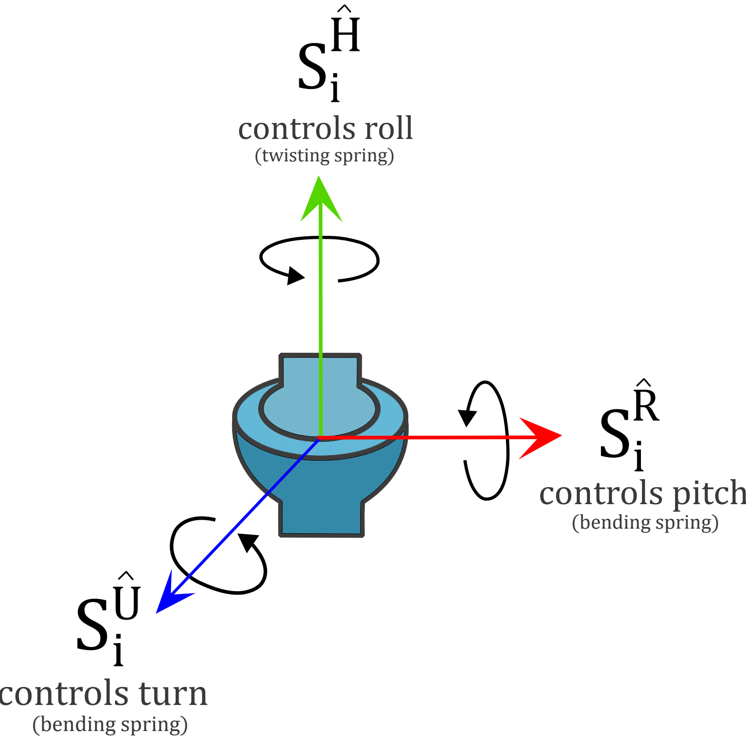


Figure 6.2: A spherical joint j_i and its local coordinate frame. Each spherical joint will be approximated as three perpendicular revolute joints each acting about one of the axes. Each revolute joint will have an angular spring associated with it: $s_i^{\hat{\mathbf{R}}}$, $s_i^{\hat{\mathbf{H}}}$, and $s_i^{\hat{\mathbf{U}}}$.

In order to make many things simpler, **every rigid-body in my articulated-body L-systems is a cylindrical internode** (and thus has constant length and radius). The main reason for this is because it is not obvious how to derive the spring constants for a spherical joint that's connecting rigid bodies with abstract shapes. If both of the rigid bodies are cylindrical internodes, however, then we can derive our spring constants by discretizing well-studied bending equations of continuous rods.

Since the body-fixed axes are aligned with the principal axes of inertia, then, since all of our rigid-bodies are cylinders, two axes will point radially and one axis will point longitudinally. If we choose our coordinate frame such that the heading vector $\hat{\mathbf{H}}$ always points longitudinally, then the spring $s_i^{\hat{\mathbf{H}}}$ will always correspond to twisting and the springs $s_i^{\hat{\mathbf{R}}}$ and $s_i^{\hat{\mathbf{U}}}$ will always correspond to bending (Figure 6.2). **This is the methodology carried out in this thesis**.

The rest of this chapter is mainly interested in the problem of finding the spring constants $k_i^{\hat{\mathbf{R}}}$, $k_i^{\hat{\mathbf{H}}}$, and $k_i^{\hat{\mathbf{U}}}$ corresponding to the joints $s_i^{\hat{\mathbf{R}}}$, $s_i^{\hat{\mathbf{H}}}$, and $s_i^{\hat{\mathbf{U}}}$, such that our articulated-bodies will realistically approximate the elasticity of real plants.

6.2 Deformation types

The motion of continuous materials is generally described through the concepts of *stresses* and *strains*. In general terms, stress $\boldsymbol{\sigma}$ is defined as force $\mathbf{F}$ per unit area $\mathbf{A}$,

$$\boldsymbol{\sigma} = \frac{\mathbf{F}}{\mathbf{A}},$$

and arises from externally applied forces. Strain is proportional to stress, and is equal to the ratio of deformation experienced by the body in the direction of the force with respect to the initial dimensions of the body. External forces may either be *surface forces* or *body forces*. Surface forces act on the external boundary of the plant, such as those from wind particles and water droplets, whereas body forces are those that act on the volume of the plant, such as gravity. Once deformed, the internal forces that bind the plant will attempt to restore the plant to its original pose. These forces will be referred to as the plant's *restoration forces*.

This behaviour in which the plant attempts to return to it's original shape is called *elastic deformation*, however, a plant under the influence of overwhelming stresses may deform permanently (*plastic deformation*), or it may outright fracture. In this thesis, I will only be considering elastic deformations, which means that the plants will always attempt to restore themselves to their original pose.

Restoration forces emanate from the composition of the molecular structures that make up a plant, but since we are limited to articulated bodies, deformation will instead be modeled

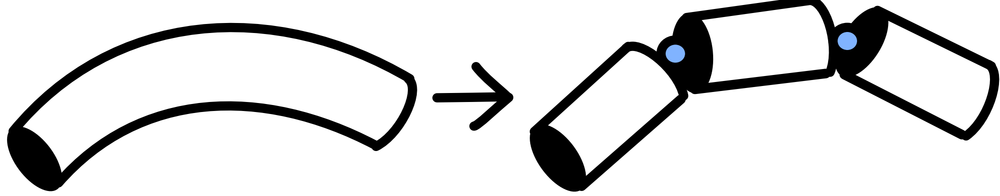


Figure 6.3: Rotations at the joints being used to represent the deformation of a continuous body

at the macroscopic scale, and will be represented by rigid bodies that have been successively rotated with respect to each other (Figure 6.3). The discrete elements in charge of the rotation between bodies will be three-dimensional spring-like joints. The main hypothesis here is that relative bending between successive bodies at the joints will approximate continuous deformation.

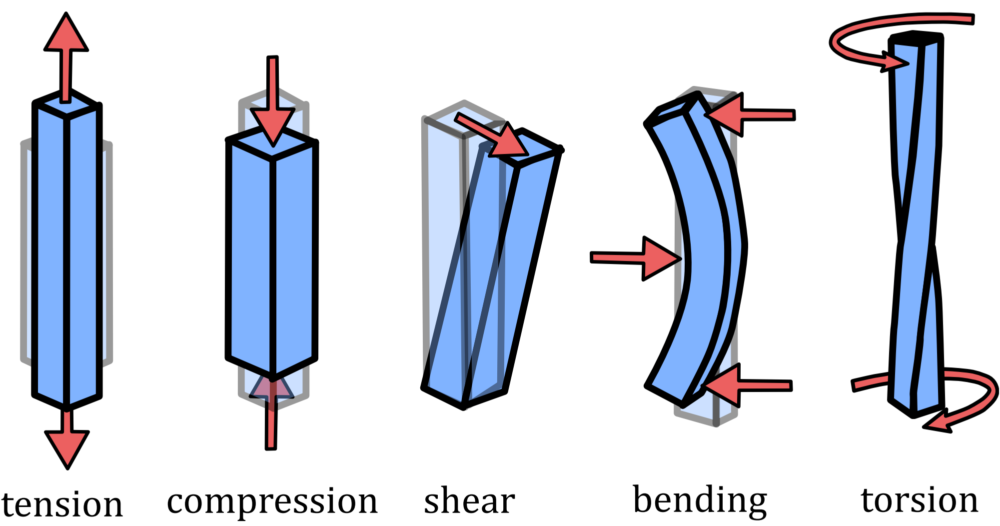


Figure 6.4: Types of elastic deformation

There are five types of elastic deformation that all continuous materials adhere to at the macroscopic level: tension, compression, shearing, bending, and torsion (Figure 6.4). In this thesis, I only tackle bending and torsion. It *is* the case that bending and torsion (especially bending) are the most visually prevalent in the common motions of plants, however, a superior physically-based model would still capture tension, compression, and shear, and thus the modeling of these classes of deformations remains as future work.

Pure bending[1] and torsion have the following definitions:

1. **Pure bending**: occurs when a torque is applied perpendicular to the internode's longitudinal axis such that no axial, shear, nor torsional forces are introduced.

2. **Torsion**: occurs when a torque is applied parallel to the internode's longitudinal axis.

The next two sections go over the biomechanical equations that govern each of the above forms of elastic deformation, as well as how we can approximate them using rigid bodies connected by angular springs. The biomechanical equations to be presented are summary of the derivations provided by Niklas in [57, Chapter 5] as well as [55, Chapter 3].

6.3 Continuous pure bending

Let us consider a cylindrical beam of length l with a circular cross section of radius r, and let us assume it has been bent into a section of a circle by a moment M such that it has constant curvature. This condition is called *pure bending* as long as the beam does not experience simultaneous axial, shear, nor torsional forces. The equation governing this interaction is

$$M = EIK, \tag{6.1}$$

where M is the bending moment, E is the *modulus of elasticity* or *Young's modulus* of the internode, I is the *second moment of area*[2] of the internode's cross-sectional surface, and K is the resulting *curvature* to which the internode has been bent. The equation indicates that the magnitude of the bending moment M required to bend a beam to a curvature K is proportional to the product of the elastic modulus E and the second moment of area I. The product EI is called the *flexural rigidity* of the internode, and can be thought of as a

[1] From hereon, bending will be referred to as 'pure bending' for clarity.
[2] Not to be confused with the moment of inertia, or inertia tensor, which shares the same symbol.

mapping between torque and bending.

E can be thought of as the internode's resistance to bending due to the composition of its material; a metallic rod, for example, would bend less than a rubber rod of the same shape if both were acted upon by the same moment M. The modulus of elasticity E is defined to be the quotient of stress and strain:

$$E = \frac{stress}{strain} = \frac{\sigma}{\epsilon}.$$

The values for E used in the plant simulations showcased in Chapter 9 were adapted from the empirical data provided by Niklas in [56].

I can be thought of as the internode's resistance to bending due to it's cross-sectional shape; given a moment M, a cylindrical internode with a large radius would greater resist M than a cylindrical internode with the same Young's modulus but a smaller radius. I can be directly calculated, and is equal to the following equation:

$$I = \int_{area} d^2 dA, \tag{6.2}$$

where dA is an area differential of the volume's cross-sectional surface, and d^2 is the squared distance of dA from the *neutral plane*. As an intuitive understanding of the above equation and the neutral plane, let's look at a continuous cylindrical internode that has been bent from its rest orientation (Figure 6.5). Since the convex portion of the beam is experiencing tension (in red) and the concave portion of the beam is experiencing compression (in blue), then there must exist a plane along the beam called the *neutral plane* (drawn as a black line) that experiences no tension nor compression. The further away an area differential dA is from the neutral plane, the more tension or compression it experiences. However, this also means that it is harder to bend an object whose accumulative cross-sectional area is

concentrated away from the neutral plane (Figure 6.6).

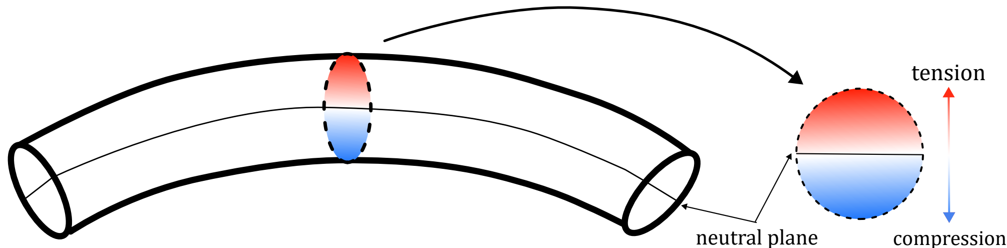


Figure 6.5: A bent internode experiences tension and compression.

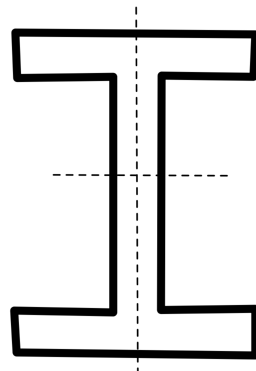


Figure 6.6: I-beams are universal in construction and engineering because they maximize the second moment of area whilst minimizing cross-sectional surface area. This means that they are relative sturdy for their low material costs.

The value of the second moment of area is generally different when measured against different neutral planes of the same object. This is just like how the (mass) moment of inertia can take different values when measured against different axes of rotation of the same rigid body. Therefore, for simplicity, we will assume the following characteristics regarding pure bending:

1. All internodes have circular cross-sections.

2. The neutral plane of all these continuous internodes perfectly cuts the internode into two semi-cylindrical internodes.

With these assumptions, one may carry out the integration in Equation 6.2 to get that the second moment of area of a cylindrical beam of uniform radius r is

$$I = \frac{\pi}{4} r^4. \tag{6.3}$$

6.3.1 Discretized pure bending

The concept of strain is inapplicable to rigid bodies because they cannot deform. Our solution will instead be to divide a continuous rod into smaller, discrete elements (in our case, cylindrical rigid bodies) and connecting them using revolute springs. Subsequently, we can determine the appropriate spring constants by discretizing the equation $M = EIK$ at the spring locations. The approach in doing so has been adapted from Power et al. [65].

The key idea here is that we can replace the curvature at the location of a spring, i, by its discrete approximation $K_i \approx \theta_i / L_i$, where L_i is the average (axial) length of the joint's adjacent cylindrical rigid bodies and θ_i is the angle between them (Figure 6.7). Note that θ_i is assumed to be the joint's current angle *with respect* to the joint's rest angle. Additionally, since the generalized coordinates $\mathbf{q}$ of the system also represent the current joint angles with respect to the rest angles, then $\theta_i \equiv q_i$, where q_i is the generalized coordinate corresponding to the angular joint in question.

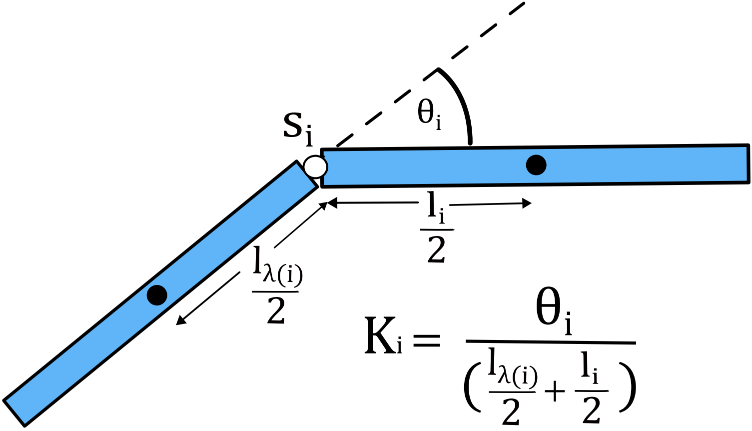


Figure 6.7: Calculating the discrete curvature K_i between two rigid bodies.

In the same vein of thinking, we can take the second moment of area at the spring location I_i to be the average second moment of area of both its adjacent bodies,

$$I_i = \frac{\frac{\pi}{4} r_{\lambda(i)}^4 + \frac{\pi}{4} r_i^4}{2} = \frac{\pi}{8} r_{\lambda(i)}^4 + \frac{\pi}{8} r_i^4, \tag{6.4}$$

where $r_{\lambda(i)}$ and r_i are the radiuses of the joint's proximal and distal cylindrical rigid bodies, respectively.

Lastly, there is no direct physical analogy between the modulus of elasticity E in the continuous case and E_i in the discrete case. This is because E describes the relationship between stress and strain in a continuous material, which is a concept absent in articulated-bodies as rigid bodies cannot deform. However, since we want the overall deformation behaviour of the articulated-body to be as close as possible to that of the continuous cylinder, we proceed by defining that $E_i = E$. The values of E used in my animations were adapted from the empirical studies carried out by Niklas in [56]. In other words, I did not use biomechanically precise values for E in my plant simulations, and instead calculated them via trial-and-error by using the values in that paper as a starting point.

By putting this all together, we get that the bending torque τ_i generated by a bending angular spring bent to an angle of θ_i away from its rest angle is

$$\tau_i = -E_i I_i K_i = -E_i \left(\frac{\pi}{8} r_{\lambda(i)}^4 + \frac{\pi}{8} r_i^4 \right) \left(\frac{2\theta_i}{l_{\lambda(i)} + l_i} \right), \tag{6.5}$$

where $l_{\lambda(i)}$ and l_i are the lengths of the joint's proximal and distal bodies, respectively, $r_{\lambda(i)}$ and r_i are the radiuses of the joint's proximal and distal bodies, respectively, θ_i is the angle between the internodes with respect to the rest angle, and E_i is an user-defined constant. The minus sign appears because τ_i is the *opposing* restoration force generated by the deformed spring.

We recall from the discussion in Section 6.1 that each spherical joint in the system s_i has two springs that correspond to bending motion, which are $s_i^{\hat{\mathbf{R}}}$ and $s_i^{\hat{\mathbf{U}}}$. Therefore, those are the two springs that will operate on the discrete bending equation above ($s_i^{\hat{\mathbf{H}}}$ operates on the *twisting* equation which is derived in the next section). The springs have corresponding spring constants $k_i^{\hat{\mathbf{R}}}$ and $k_i^{\hat{\mathbf{U}}}$, which can be obtained by coupling up all the constants in Equation 6.5:

$$k_i^{\hat{\mathbf{R}}} = k_i^{\hat{\mathbf{U}}} = E_i \left(\frac{\pi}{8} r_{\lambda(i)}^4 + \frac{\pi}{8} r_i^4 \right) \left(\frac{2}{l_{\lambda(i)} + l_i} \right). \tag{6.6}$$

This symbolic representation of k_i along with the fact that $\theta_i \equiv q_i$ allows the bending spring equations to reach their simplest form:

$$\begin{aligned} \tau_i^{\hat{\mathbf{R}}} &= -k_i^{\hat{\mathbf{R}}} q_i^{\hat{\mathbf{R}}}, \\ \tau_i^{\hat{\mathbf{U}}} &= -k_i^{\hat{\mathbf{U}}} q_i^{\hat{\mathbf{U}}}, \end{aligned} \tag{6.7}$$

which is the angular version of Hooke's law. In the equation above, $\tau_i^{\hat{\mathbf{R}}}$ and $\tau_i^{\hat{\mathbf{U}}}$ are the restoration torques generated about a spherical spring's $\hat{\mathbf{R}}$ and $\hat{\mathbf{U}}$ axes, respectively, whereas $q_i^{\hat{\mathbf{R}}}$ and $q_i^{\hat{\mathbf{U}}}$ are their respective joint angles. Section 8.3 goes over an example cantilever simulation that provides quantifiable insight regarding the correctness of approximating real bending using the above restoration forces.

6.4 Continuous torsion

Let's again take a continuous cylindrical internode as an example. Whereas pure bending consists of a compression along the concave portion of the internode and tension along the convex portion of the internode, torsion can be thought of as twisting (shearing strain) between successive 'circular slices' of the internode along its longitudinal axis.

The equation for torsion has identical form to equation $M = EIK$, and looks as follows:

$$T = GJ\frac{d\theta}{dl} \tag{6.8}$$

where T is the axial torque acting along the longitudinal axis of the internode, G is the *shear modulus* of the internode's material, J is the internode's *polar second moment of area*, θ is the twist amount (in radians), and l is the length over which the twisting takes place.

Just like the modulus of elasticity E represents resistance to bending due to the composition of the tissue, the shear modulus G represents resistance to shearing - and as a consequence - twisting. The polar second moment of area is thus the torsional counterpart to the (planar) second moment of area; it is the internode's resistance to torsion due to it's cross-sectional shape. However, whereas bending exhibited a *neutral plane* that experienced no deformation, torsional deformation exhibits a *neutral axis* only. The neutral axis is the longitudinal axis about which the internode is twisting. Just like we assumed that the neutral plane cuts the cylindrical internode into two semi-cylindrical ones, we will assume that the neutral axis perfectly runs along the cylindrical internode's centroidal axis.

The equation for the polar moment of area is the same as the equation for the second moment of area, the only difference being that we are measuring distances from the neutral axis as opposed to the neutral plane:

$$J = \int_{area} d^2 dA.$$

where d is the distance of the area differential dA from the neutral axis. An internode with uniform radius r has

$$J = \frac{\pi}{2} r^4. \tag{6.9}$$

We note that it is twice as large as the planar second moment of area ($I = (\pi/4)r^4$), which is a key reason why it is easier to bend plants than it is to twist them. Section 9.2.2 provides a constructive example of the importance of shape in the plant simulations.

6.4.1 A note on the values used for the shear modulus of elasticity

The shear modulus of elasticity G may be used to represent a material's resistance to twisting. The values for G used in my plant simulations were calculated from their Young's modulus counterpart E via the following equation ([8, Chapter 2], [55, Chapter 2]):

$$G = \frac{E}{2(1+\nu)}. \tag{6.10}$$

This equation relates E and G through the *Poisson's ratio* ν of the plant material in question, where the Poisson's ratio relates lateral strain to axial strain:

$$\nu = -\frac{\text{lateral strain}}{\text{axial strain}}. \tag{6.11}$$

In the context of our continuous cylindrical internodes, ν can tell us how much the cylinder will 'fatten' when its main axis is squeezed, or how much the internode will 'slim' when its main axis is stretched. Most materials have a value of ν between 0 (highly undeformable) to 0.5 (highly deformable). The main issue here is that the equation is typically only applicable to engineering materials such as metals and plastics. This is because the equation assumes the following three things:

1. it assumes that the material is linearly elastic.
2. it assumes that the material is homogeneous (composed of the same material throughout its volume).

3. it assumes that the material is isotropic (the material is equivalent when measured against any direction).

This is rather problematic because plant stem tissue, considered as a material, may exhibit non-linear elastic behaviours, and is both heterogeneous and highly anisotropic, which begs the question: “to what extent does this approximation affect the plant simulations?”. With all this said, however, it is non-trivial to find better values for G because the relation between stresses and strains and the material moduli of anisotropic materials must be empirically determined [55, Chapter 2]. Furthermore, these measurements may be affected by external factors such as temperature, humidity, and nutrient availability.

In conclusion, the twisting motion exhibited by my simulations operating on this approximation was deemed plausible enough (see Section 9.2.1) to the point that other components of the thesis took precedent over the use of this approximation. This means that a superior method in finding twisting spring constants was not pursued, and it remains as future work to quantify the error produced by this approximation, and perhaps even to find a better alternative altogether (which may mean empirically determining the material properties in order to more faithfully simulate the motion of a desired plant).

6.4.2 Discretized torsion

We need to find the spring constant that will best approximate the torsional restoration torques between two rigid bodies that have been twisted with respect to each other. We proceed to do this by discretizing the equation $T = GJ\frac{d\theta}{dl}$ at the locations of the springs i, analogously to how we did the bending equation.

We can replace the differential equation $d\theta/dl$ by its discrete approximation θ_i/L_i, where θ_i is the twist angle between the joint's adjacent bodies, and L_i is the length over which the

twisting takes place. This length can be computed as the average length of both adjacent cylindrical bodies:

$$\frac{d\theta}{dl} \approx \frac{\theta_i}{L_i} = \frac{2\theta_i}{l_{\lambda(i)} + l_i}, \tag{6.12}$$

where $l_{\lambda(i)}$ and l_i are the lengths of the joint's proximal and distal cylindrical rigid bodies, respectively. We again recall that θ_i is assumed to be the twisting angle *with respect* to the joint's rest twisting angle, meaning that $\theta_i \equiv q_i$, where the generalized coordinate q_i is the spring's corresponding joint angle.

Furthermore, we can take the polar second moment of area at the spring location J_i to be the average polar second moment of area of both its adjacent bodies,

$$J_i = \frac{\frac{\pi}{2} r_{\lambda(i)}^4 + \frac{\pi}{2} r_i^4}{2} = \frac{\pi}{4} r_{\lambda(i)}^4 + \frac{\pi}{4} r_i^4, \tag{6.13}$$

where $r_{\lambda(i)}$ and r_i are the radiuses of the joint's proximal and distal cylindrical rigid bodies, respectively.

Finally, the discussion for G_i is the same as the previous discussion for E_i; there is no direct analogy between the value of the shear modulus of elasticity G in the continuous case and G_i in the discrete case, so, since we desire similar behaviour, we set $G_i = G$. As explained in Section 6.4.1, the shear modulus G_i at the location i of a spring was heuristically obtained from its Young's modulus counterpart E_i via the equation

$$G_i = \frac{E_i}{2(1 + \nu_i)}, \tag{6.14}$$

where ν_i is the Poisson's ratio of the continous plant material at the location of the spring. Materials generally experience a value of ν between 0 (undeformable; diamond, ceramics, carbon fiber composites) and 0.5 (highly deformable; rubber, skin, cartilage). All the plant

models simulated in Chapter 9 represent small, herbaceous plants consisting of medium rigidity (when compared to both extremes), and thus the range of values used for ν was between 0.2 and 0.3, which is faithful to empirical studies of comparable specimen ([36, 58], [55, Chapter 2]).

By putting this all together, we get that the twisting torque τ_i generated by an angular spring bent to an angle of θ_i away from its rest angle is

$$\tau_i = -G_i J_i \frac{\theta_i}{L_i} = -\left(\frac{E_i}{2(1+\nu_i)}\right)\left(\frac{\pi}{4}r_{\lambda(i)}^4 + \frac{\pi}{4}r_i^4\right)\left(\frac{2\theta_i}{l_{\lambda(i)} + l_i}\right), \tag{6.15}$$

where $l_{\lambda(i)}$ and l_i are the lengths of the joint's proximal and distal bodies, respectively, $r_{\lambda(i)}$ and r_i are the radiuses of the joint's proximal and distal bodies, respectively, θ_i is the angle between the internodes with respect to the rest angle, and both E_i and ν_i are user-defined constants. The minus sign again appears because τ_i represents the *opposing* restoration force generated by the spring.

Finally, since the spring acting about the twisting axis of a spherical joint is $s_i^{\hat{\mathbf{H}}}$, we can conclude from Equation 6.15 that its corresponding spring constant, $k_i^{\hat{\mathbf{H}}}$, is equal to

$$k_i^{\hat{\mathbf{H}}} = \left(\frac{E_i}{2(1+\nu_i)}\right)\left(\frac{\pi}{4}r_{\lambda(i)}^4 + \frac{\pi}{4}r_i^4\right)\left(\frac{2}{l_{\lambda(i)} + l_i}\right). \tag{6.16}$$

This symbolic representation of $k_i^{\hat{\mathbf{H}}}$ along with the fact that $\theta_i \equiv q_i$ allows the torsional spring equation to reach the simple form

$$\tau_i^{\hat{\mathbf{H}}} = -k_i^{\hat{\mathbf{H}}} q_i^{\hat{\mathbf{H}}}, \tag{6.17}$$

which is again the angular version of Hooke's law. In the equation above, $\tau_i^{\hat{\mathbf{H}}}$ is the torsional restoration torque generated about a spherical spring's $\hat{\mathbf{H}}$ axis, and $q_i^{\hat{\mathbf{H}}}$ is its corresponding joint angle. Section 9.2.1 presents an example plant simulation that explicitly showcases

torsional movement as guided by the spring constant derived above.

6.5 Infinitesimal rotations

We have presented a discretization scheme in which a continuously deformable rod can be represented by rigid bodies connected with 3-D spherical joints. Additionally, we have mentioned that the 3-D rotation of the spherical joint will be approximated via three consecutive 1-D rotations along the three pairwise orthogonal coordinate axes of the spherical joint. This method of representing 3-D rotations is known as *Euler angles* [31, Chapter 4 section 4]. In our implementation, the three 1-D rotations correspond to a roll (rotation about $\hat{\mathbf{H}}$), pitch (rotation about $\hat{\mathbf{R}}$), and yaw (rotation about $\hat{\mathbf{U}}$). However, Euler rotations are not commutative [31, Chapter 4 section 7], so which sequence of 1-D rotations should we employ in order to yield the most realistic motion?

The theoretical answer is that we can actually pick whichever order of rotations we wish because, whilst finite rotations are not commutative, *infinitesimal* rotations are [31, Chapter 4 section 8]. An infinteseimal rotation is an orthogonal transformation of coordinate axes in which the components of a vector are almost the same in both sets of axes. Therefore, since we are only ever advancing the system's joint angles forwards in time by an incremental amount dt, an argument can be made that rotation order is not critical. **This is the argument employed in this thesis**, such that the rotation order $\mathrm{R}^{\hat{\mathbf{R}}} \rightarrow \mathrm{R}^{\hat{\mathbf{H}}} \rightarrow \mathrm{R}^{\hat{\mathbf{U}}}$ was the one arbitrarily chosen. This rotation scheme is referred to as *intrinsic* XYZ *Euler angles*, where XYZ refers to the rotation order (which corresponds to $\hat{\mathbf{R}}\hat{\mathbf{H}}\hat{\mathbf{U}}$ in my implementation), and the keyword intrinsic refers to the fact that the axes rotate with each other (as opposed to *extrinsic* Euler angles).

Having said this, however, it should be noted that the argument is flawed and there is ample room for future work in this area (e.g. what about a quaternion representation of the spherical joints, as described in [25, Chapter 4]?). The reason for this is that whilst dt *is* minuscule, we cannot say that it is infinitesimal. As a consequence, we cannot guarantee that advancing the articulated-body forward in time by dt will result in infinitesimal rotations. Indeed, it was found that the simulations could gain energy every simulation step if dt was not small enough, which is severe in my simulations because they advance their state forwards in time roughly one-hundred thousand times per second, meaning that the tiniest addition of energy in a single simulation step will result in a catastrophic explosion of the system's state all within a fraction of a second. It should be disclosed that large dt can cause numerical instabilities for multiple reasons and not just the finite rotation issue (Section 9.4.2 discusses efficiency and stability of the methodology), and therefore it is not clear to me the extent to which the Euler-angle approximation contributes towards the simulation's instabilities.

For context, all the simulations employed in subsequent chapters employ a value of dt between 0.01s and 0.0000001s, with the cantilever example in Section 8.3 requiring the smallest dt in order to remain stable.

6.6 Damping

The restoration forces as presented in the previous sections will ensure that the plant model always attempts to reorient itself to its *rest pose*, where the rest pose refers to the image an articulated-body takes as a whole when all of its generalized coordinates are zero, that is, when $\mathbf{q} = \mathbf{0}$. The magnitude of the restoration forces are generally large enough to cause overshooting (which is when the plant swings past its rest pose), which is correct behaviour because overshooting is the cause of the common oscillatory motion of real plants. We have

not, however, discussed any means by which to dissipate the mechanical energy in the system, which is problematic for two distinct reasons:

Biomechanical reason: real plants dissipate mechanical energy which explains why they eventually stop oscillating. Therefore, if we do not model this, our plant simulations will oscillate much more (and possibly indefinitely) than their real-life counterparts, which is not natural behaviour.

Numerical stability reason: As briefly discussed in the previous section, plant simulations operating on the methodology presented in this thesis can exhibit severe numerical instabilities. A common source of the instability is the subtle yet catastrophic addition of energy into the system due to numerical error. A system that dissipates energy will therefore be more stable as a result.

We therefore seek to introduce *damping forces* into the system whose job are to dissipate mechanical energy. There are several sources of damping in real plant motion, however, the methodology implemented in this thesis only models *linear velocity-dependent damping at the joints*. One could say that this type of damping mimics the internal friction that real plants experience when they deform, but this is dishonest in my implementation because my spring constants were determined by trial-and-error as opposed to through biomechanical insights. I believe there to be ample room for improvements regarding damping in my methodology.

The implementation for this type of damping can be derived by first noting that each simulation step, every angular spring i in the system is effectively computing its restoration torque via the Hooke's law equations as described in Sections 6.3 and 6.4:

$$\tau_i = -k_i q_i, \tag{6.18}$$

where τ_i is the restoration torque generated by the angular spring, k_i is the spring constant

as derived in Sections 6.3 and 6.4, and q_i is the current spring angle (away from rest angle) of the respective angular spring. Therefore, to damp the system, we can introduce the typical linear velocity-dependent damping term, $-b_i\dot{q}_i$, to Equation 6.18:

$$\tau_i = -k_i q_i - b_i \dot{q}_i, \tag{6.19}$$

where b_i is the linear *damping constant* of the spring, and $\dot{q}_i$ is the instantaneous angular velocity of the spring. The damping constants used throughout the simulations in Chapter 8 and Chapter 9 were all determined manually on a trial-error basis; the simulations were ran against many damping constants until the motion looked as realistic as possible. It remains as future work to figure out an automatic way with which to figure out this damping constant from biomechanical principles.

Note on drag forces

Another common source of damping comes from drag forces, which in these simulations, corresponds to *air resistance*. Air resistance acts on all objects moving through a fluid (or gas), and does so in the opposite direction of the object's motion. This is because the moving object is attempting to push away the molecules of the space in its direction of travel, and as a consequence, these disturbed molecules are applying an equal and opposing force on the object.

It was my impression that air resistance would be trivially implemented by calculating the external drag forces acting on each rigid body, and adding them to the system's $\mathbf{f}_{ext}$ vector. However, every attempt at implementing drag forces led to the same result: the drag forces were introducing severe instabilities into the system.

For context, my air resistance implementations worked perfectly for a single-pendulum, and

it was even proved that, for the one-body problem, certain constants existed such that air resistance could be equivalent to the previously described method of damping at the joints. However, any further addition of rigid-bodies resulted in increasingly unstable systems, which is troubling because damping forces are supposed to stabilize physically-based animation models and not the opposite. The issue is definitely that the propagating drag forces introduced energy into the system, what is not obvious, however, is why this was the case. It is unclear to me whether I've severely overlooked something, or whether more consideration needs to be given regarding the nature in which drag forces are handled in an articulated-body implementation.

As a consequence, the incorporation of drag forces into my thesis was eventually scrapped, and I instead exaggerated the nodal damping forces in order to account for the missing drag forces.

Chapter 7

System Overview

The main programming contribution in this thesis is not a standalone application, but rather an object-oriented C++ library nicknamed the *L-system dynamics library* (LSDL). The purpose of LSDL is to simulate the dynamics of L-system objects using articulated bodies, and is designed to be easily integrated into any C++ projects. LSDL is a direct extension to Martin Felis' open source *Rigid Body Dynamics Library* (RBDL)[28], which implements the data structures and algorithms presented in Featherstone's *Rigid Body Dynamics Algorithms* [25]. A subset of RBDL's functionality is exactly what is presented in Chapter 4, which is the functionality on which LSDL is built on.

Section 7.1 presents the key RBDL components that were used to develop LSDL, Section 7.2 presents LSDL, and Section 7.3 describes two sample graphical applications that operate on LSDL, which were implemented throughout the completion of the thesis.

7.1 RBDL

RBDL is an efficient C++ physics library that implements the data structures and algorithms described in Featherstone's *Rigid Body Dynamics Algorithms* [25]. One of these algortihms is

the articulated-body algorithm which was the topic of Chapter **??**. This section is concerned with explaining exactly what functionality was adopted from RBDL in the development of LSDL.

There are three key RBDL components that were used in the creation of LSDL:

1. Its math library that implements 6-D vector algebra.
2. Its `PhysicsModel` data structure that stores all the data of an abstract articulated-body.
3. Its `PhysicsModel::ForwardDyamics()` method that runs the articulated-body algorithm on an object of type `PhysicsModel`.

7.1.1 6-D Vector Algebra Library

RBDL includes the implementation of a 6-D vector algebra library that facilitates the development of code involving spatial algebra. It provides the data structures and operations shown in Section 4.4.2, namely, the data structures for spatial matrices and spatial vectors along with their operations.

7.1.2 The `PhysicsModel` data structure

The `PhysicsModel` data structure contains all the information necessary to describe an abstract articulated body. It is mainly composed of two basic data structures: the `Body` data structure and the `Joint` data structure.

The `Body` data structure contains data such as its mass, center of mass, and inertia tensor (all in body coordinates), and the `Joint` data structure contains data defining which directions

the joint allows motion in.

The remaining contents in the PhysicsModel class are just as described in Section 4.3. As a summary, each instance of a PhysicsModel class stores the following information:

1. The rigid bodies in the system.

2. The joints in the system.

3. The connectivity between the rigid-bodies and joints.

4. The gravity acting on the system.

In addition, it always stores the most up-to-date values of the following information:

1. The dynamic state of the articulated body ($\mathbf{q}$, $\dot{\mathbf{q}}$, $\boldsymbol{\tau}$, and $\mathbf{f}_{ext}$).

2. The generalized accelerations $\ddot{\mathbf{q}}$ of the articulated body.

3. The instantaneous absolute spatial transformation matrix ${}^{i}\check{\boldsymbol{X}}_0$ of each body.

4. The instantaneous absolute spatial velocity $\check{\mathbf{v}}_i$ and spatial acceleration $\check{\mathbf{a}}_i$ of each body.

Note that `RBDL` never manipulates the generalized input forces $\boldsymbol{\tau}$ nor the spatial external input forces $\mathbf{f_{ext}}$; they are supposed to be set by the application using `RBDL`.

7.1.3 The `PhysicsModel::ForwardDynamics()` method

The `PhysicsModel::ForwardDynamics()` method belongs to all objects of type PhysicsModel. When the method is called, it uses as input the model's current dynamic state ($\mathbf{q}$, $\dot{\mathbf{q}}$, $\boldsymbol{\tau}$, and $\mathbf{f}_{ext}$) to update the model's acceleration vector $\ddot{\mathbf{q}}$ via the articulated-body algorithm. The procedure in doing all of this is exactly as derived in the latter half of Chapter 4.

7.2 LSDL

The *L-system dynamics library* (LSDL) is the main programming component completed in this thesis. The job of LSDL is to simulate the dynamics of models generated by L-systems using articulated-body dynamics. From hereon, we will assume that any L-system in question will have the format described in Section 5.2.

The most important class contained in LSDL is the `PlantModel` class, as it contains all the functionality needed to create and simulate the dynamics of L-system objects. The `PlantModel` class inherits RBDL's `PhysicsModel` class, with the intention of using RBDL's `PhysicsModel::ForwardDynamics()` method on objects of type `PlantModel`. In other words, the `PlantModel` class is simply an extension to the `PhysicsModel` class that incorporates functionality specific to simulating the dynamics of plants. There are two key methods that are exposed by the `PlantModel` class:

`PlantModel::PlantModel(string lsystem_string)`: The constructor of the `PlantModel` class that takes in an L-system string as input. This is the method that should be called by an external program whenever a new plant instance should be created in the simulation. The format of the input L-system string should be as described in Section 5.2.

`PlantModel::PhysicsStep(float dt)`: When called, the plant model advances its positional state variables ($\mathbf{q}$, $\dot{\mathbf{q}}$) in time by the amount `dt`. It also computes the absolute positions and velocities of each rigid body.

An external program using LSDL is intended to do so through the `PlantModel` interface, treating it as a black box. Algorithm 2 depicts the structure that a sample program using LSDL might take. Note that this sample program does not yet include user interaction nor visuals; each of these topics will be talked about later in this chapter.

Algorithm 2 An example usage of LSDL. Each iteration of the while loop advances the system 0.01s forward in time.

```
1: procedure MAIN(string lsystem_string)
2:
3:     PlantModel model ← new PlantModel(lsystem_string)
4:     float dt ← 0.01
5:
6:     while true do
7:         model->PhysicsStep(dt)
8:     end while
9:
10: end procedure
```

The two sub-sections that follow go over the `PlantModel(string lsystem_string)` method and the `PhysicsStep(float dt)` methods.

7.2.1 The `PlantModel::PlantModel(string lsystem_string)` method

The job of the `PlantModel::PlantModel(string lsystem_string)` method is to sequentially parse (from left to right) all of the modules in the input L-system string, `lsystem_string`, and to add any encountered joint and rigid body modules to the articulated body. It does not matter how the input `lsystem_string` was generated; only that it have a permissible structure as per Section 5.2. The pseudocode for the method is presented in Algorithm 3.

In Algorithm 3, the method `PhysicsModel::AddNewBody(Body parent, Joint joint, Body child)` is an RBDL method that takes in a `Body` object called `parent`, a `Joint` object called `joint`, and another `Body` object called `child`. The method connects the `child` body to the `parent` body via the joint `joint`, effectively adding the `child` body to the articulated-body.

It is worthwhile to clarify the following:

- RBDL's `PhysicsModel::AddNewBody(Body parent, Joint joint, Body child)` method

Algorithm 3 Constructor method for a plant object.

```
1: procedure PlantModel::PlantModel(string lsystem_string)
2:
3:     Stack⟨Body⟩ parentStack ← empty
4:
5:     Body parent ← null
6:     Joint joint ← null
7:     Body child ← null
8:
9:     for each Module module in lsystem_string do[1]
10:
11:        if module.type == Joint then
12:            joint ← m
13:        else if module.type == Body then
14:            child ← m
15:            AddNewBody(parent, joint, child)
16:            parent ← child
17:        else if module.type == NewBranch then
18:            parentStack.add(parent)
19:        else if module.type == EndBranch then
20:            parent ← parentStack.takeLast()
21:        else
22:            continue
23:        end if
24:
25:    end for
26:
27: end procedure
```

requires that the position and orientation of the `joint` be specified with respect to both the `parent` and `child` bodies. The reason why this geometrical information does not need to be included in the parameters of the method is because it can be inferred from the formal parameters of the `parent`, `child`, and `joint` modules (as per the discussion in Section 5.2).

- Note the use of a stack to create branching structures.

7.2.2 The `PlantModel::PhysicsStep(float dt)` method

The job of the `PlantModel::PhysicsStep(float dt)` method is to advance the plant model's state variables ($\mathbf{q}$ and $\dot{\mathbf{q}}$) in time by the amount `dt`. It uses the currently-stored generalized forces $\boldsymbol{\tau}$ and external forces $\mathbf{f}_{\mathbf{ext}}$ of the plant model to do so. The pseudocode is presented in Algorithm 4.

Algorithm 4 Pseudocode for the PhysicsStep method

```
 1: procedure PlantModel::PhysicsStep(float dt)
 2:     ComputeForces()
 3:     ForwardDynamics()
 4:     TimeIntegrate(dt)
 5:     ComputePositions()
 6: end procedure
 7:
 8: procedure PlantModel::TimeIntegrate(float dt)
 9:     q̇ ← q̇ + q̈ · dt
10:     q ← q + q̇ · dt
11: end procedure
12:
```

- The `ComputeForces()` method is in charge of computing the plant's restoration and damping forces as derived in Chapter 6, and storing them in $\boldsymbol{\tau}$.

- The `ForwardDynamics()` method has already been described; it uses as input the model's current dynamic state ($\mathbf{q}$, $\dot{\mathbf{q}}$, $\boldsymbol{\tau}$, and $\mathbf{f}_{ext}$) to update the model's acceleration

vector $\ddot{\mathbf{q}}$ via the articulated-body algorithm.

- The `TimeIntegrate(float dt)` method uses the newly-computed value of $\ddot{\mathbf{q}}$ along with the input time step `dt` to determine $\dot{\boldsymbol{q}}$ and $\boldsymbol{q}$ at a point `dt` further in time. The time integration scheme employed is semi-implicit forward Euler.

- Lastly, the `ComputePositions()` method is called at the very end, and is in charge of updating the transformation matrix of each rigid body with respect to the global reference frame $\mathcal{F}_{\mathcal{O}}$. An external application may visualize the articulated-body after the `ComputePositions()` statement has been called, and can do so by making use of these updated transformation matrices.

7.3 Example applications using LSDL

Two LSDL applications were created throughout the development of this thesis. The first is a simple OpenGL program that served as testing grounds for the simulations, and the second is an Unreal Engine 5 [30] application that was used to create the final renderings found in Chapter 9. It should be noted that there is no real difference between the underlying structure of these applications, as they both have the structure described in Algorithm 2.

7.3.1 OpenGL application

The structure of the example OpenGL application is illustrated in Algorithm 5. We first note that it is a direct extension to Algorithm 2. Let's look at the components of the pseudocode:

- The OpenGL application takes an L-system string as its single input, which is no different from Algorithm 2.

- The `SetupOpenGLGraphicsEnvironment()` function constitutes the typical setup phase that any graphical application adheres to. This includes creating the window, binding

Algorithm 5 The layout of the OpenGL application.

```
1: procedure MAIN(string lsystem_string)
2:
3:     SetupOpenGLGraphicsEnvironment();
4:     model ← new OpenGLPlantModel(lsystem_string)
5:
6:     while true do
7:         HandleUserInput(model);
8:         model->PhysicsStep(dt)
9:         RenderPlant(model);
10:    end while
11:
12: end procedure
```

input, compiling shaders, and setting up the camera.

- In line 4 we create our plant model. The `OpenGLPlantModel` class inherits the `PlantModel` class as described in Section 7.2, meaning that line 4 also runs the `PlantModel` constructor (Algorithm 3), which creates the articulated-body from the provided `lsystem_string`. The `OpenGLPlantModel` includes any data structures specific to an OpenGL interpretation of the plant model, which in my implementation, only consisted of storing the triangular meshes used for rendering the model's rigid bodies.

- All application-specific user interaction features reside in the `HandleUserInput(model)` function, which includes essentials such as camera movement, camera zoom, and closing the application. More importantly, however, the `HandleUserInput(model)` function includes all user input that results in the generation of external forces (pulling, moving the base), as well as user input that results in a manipulation of simulation constants (gravity, damping, elasticity, etc.). I figured it best to present the interaction methods employed in this thesis alongside their graphical simulations, therefore the explanations for these user interactions are scattered throughout Chapter 9. Forces generated inside this function should either be added to the plant model's $\boldsymbol{\tau}$ vector (if they are generalized forces) or $\mathbf{f_{ext}}$ vector (if they are global external forces). These forces will

be used by the `PhysicsStep(dt)` method when computing the new positions of the plant model.

- Lastly, `RenderPlant(OpenGLPlantModel model)` is an application-specific function that takes in an object of type `OpenGLPlantModel` and renders it to the scene. Algorithm 6 shows the pseudocode for the `RenderPlant(model)` function. The `transformation` member variable in line 4 is equal to the body's global transformation matrix, ${}^{i}\check{\boldsymbol{X}}_{0}$, and the `mesh` member variable in line 5 is an object of type `Mesh`, which is a OpenGL-specific data structure storing all the relevant data corresponding to the body's geometric mesh. This includes data such as vertices, faces, textures, UV coords, vertex normals, and material properties. Finally, the `RenderMesh(Mesh mesh, mat4x4 transform)` function within the `RenderPlant` function consists of the typical GPU function calls that bind the active vertices and active transform of the GPU to those contained in the parameters `mesh` and `transform`, respectively.

Algorithm 6 The RenderPlant() method

```
1: procedure Render(OpenGLPlantModel m)
2:
3:     for each RigidBody b in m do
4:         mat4x4 transformation ← b.transformation
5:         Mesh mesh ← b.mesh
6:         RenderMesh(mesh, transformation)
7:     end for
8:
9: end procedure
```

7.3.2 Unreal Engine 5 application

The underlying structure of the Unreal Engine 5 (UE5) [30] application is depicted in Algorithm 7. We can see that the OpenGL and UE5 applications are identical in form.

The Unreal engine was thus used mainly for its graphics, and here are the following key UE5 tools that were used:

Algorithm 7 The layout of the UE5 application.

```
1: procedure MAIN(string lsystem_string)
2:
3:     SetupUE5GraphicsEnvironment();
4:     model ← new OpenUE5PlantModel(lsystem_string)
5:
6:     while true do
7:         HandleUserInput(model);
8:         m.PhysicsStep(dt)
9:         RenderPlant(model);
10:    end while
11:
12: end procedure
```

Real-time global illumination: all the simulations showcased in Chapter 9 were rendered using UE5's real-time global illumination, which they've called **Lumen**.

Bi-directional reflectance distribution function (BRDF) materials: The geometric mesh models and corresponding textures making up the plant models in Chapter 9 were personally modeled using using blender [12]. However, their materials were created within UE5, which offers an easy way to set material properties such as roughness, randomness, specularity, and even subsurface scattering.

Instanced static-meshes: Only a single internode/leaf/flower mesh is ever loaded into GPU memory for any particular plant (including their textures and materials). This is done via the engine's support for *instancing*. Then, each render frame, I can provide different transformation matrices for each instanced rigid body; these transformation matrices are the ones that are output each frame by Featherstone's algorithm (the programming connection between my thesis and unreal engine is exactly this).

Timeline editor: The user interaction for the comparison simulations in Section 9.1.5 and 9.2.3 were scripted using UE5's timeline editor. It enabled me to easily re-run the same same pull motion against different material properties (e.g. rigidity, damping)

until the motion was accurate.

Chapter 8

Validations

The main goal for this thesis is to create *plausible* simulations of procedurally-generated plants. However, it's still important to perform some test simulations in order to provide quantifiable validity over the presented methodology. In order to do so, this chapter will be concerned with the following example simulations:

- **Single physical pendulum**: A single rigid-body constrained to rotate in 2D.
- **Double physical pendulum**: A link of two rigid-bodies constrained to rotate in 2D.
- **Cantilever beam**: A link of n rigid-bodies connected together by revolute joints, deflecting under its own weight.

The three subsections that proceed tackle each of these simulations.

8.1 The physical pendulum

A rigid body that is free to swing under its own weight about a fixed horizontal axis of rotation is known as a *physical pendulum* or *compound pendulum* (Figure 8.1).

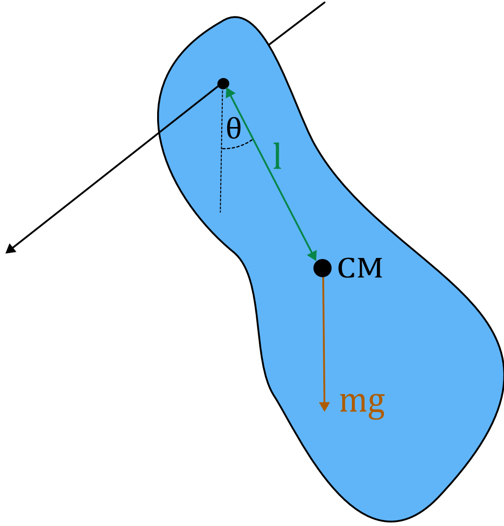


Figure 8.1: Free body diagram of a 2D physical pendulum

The equation of motion governing this system is simple and can be derived by hand. The idea is that we can validate the implementation presented in this thesis by animating two identical pendulums in parallel: one operating on the manually derived equation and the other operating on the `L-System Dynamics Library` (LSDL, the physics library created in this thesis). Presumably, if both approaches are equivalent as they should be, the simulations will be identical. Actually, since there are no plant-motion related components in this simulation, we are essentially only testing the base features of LSDL which are provided by Martin Felis' `Rigid-Body Dynamics Library` (`RBDL`) [28]. Since RBDL already has a much more in-depth suite of test cases [28], this example is more-so validating the following:

1. It validates that the process from modeling an L-system and exporting it into LSDL works as intended (the examples that I will show in this chapter were all created with L-systems).

2. It validates that no errors were introduced when building LSDL on top of RBDL.

We now continue by deriving the equation of motion. If we constrain the body's oscillations within two dimensions, then, in the absence of non-inertial forces, the equation for the angular acceleration of the body $\ddot{\theta}_1$ can be obtained from the angular version of Newton's

2nd law:

$$\tau = I\ddot{\theta}_1, \tag{8.1}$$

where τ is the total torque applied to the body about its rotation axis, I is the moment of inertia of the body with respect to its rotation axis, and $\ddot{\theta}_1$ is the resulting angular acceleration the body experiences. We also know that τ is the moment generated by the gravitational force acting on the pendulum. In the 2-D case, it is equal to the scalar quantity $\tau = d \times F$, where $d = l$ is the distance from the center of mass to the rotation axis, and $F = -mg\text{sin}\theta_1$ is the force acting in the direction of rotation:

$$\tau = -mgl \sin \theta_1. \tag{8.2}$$

Combining and rearranging Equations 8.1 and 8.2 gives us the angular acceleration of the body at it's joint:

$$\ddot{\theta}_1 = \frac{-mgl}{I} \sin \theta_1. \tag{8.3}$$

If we take the same pendulum and animate it using LSDL, the same equation of motion will presumably be calculated:

$$\ddot{\theta}_2 = ABA(model, \theta_2) = \frac{-mgl}{I} \sin \theta_2. \tag{8.4}$$

In order to test this, we shall simulate two identical pendulums with identical initial conditions. The only difference will be that one will operate on the equation of motion derived by hand (8.3), and the other on the equation of motion derived by Featherstone's algorithm (8.4). Note that both simulations need to operate on the same time integration scheme in order for the comparison to be valid. I used semi-implicit forward Euler integration for the time evolution of both systems:

$$\dot{\theta}_i = \dot{\theta}_i + \ddot{\theta}_i \Delta t,$$
$$\theta_i = \theta_i + \dot{\theta}_i \Delta t. \tag{8.5}$$

Figure 8.2 depicts three snapshots of both simulations running in parallel. The pendulum on the left is computing accelerations using `RBDL`'s implementation of Featherstone's articulated-body algorithm, whereas the one on the right is computing accelerations using the equations derived above. Both simulations are using the same time integration scheme (the semi-implicit Euler integration mentioned above). The first snapshot is at $t = 0$, the second snapshot is at $t = 16s$, and the third snapshot is taken at $t = 49s$. We see that the simulations do not appear to diverge.

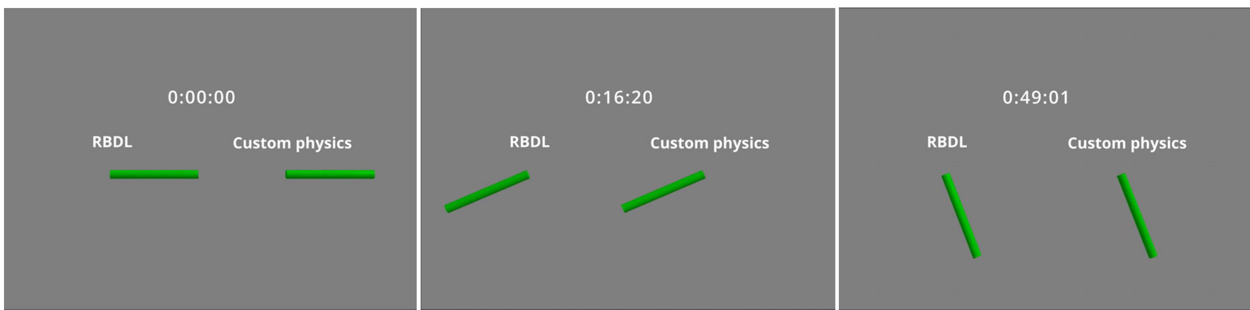


Figure 8.2: Snapshots of two single physical pendulums simulated side by side.

Indeed, the program was left running overnight and yet they did not diverge, which cements the hypothesis that both simulations are equivalent (and additionally, deterministic). Using different time integration schemes on both simulations would likely cause them to eventually diverge.

8.2 Double physical pendulum

The single physical pendulum is trivial because there are no non-inertial accelerations acting on the system. The next system worth testing against is therefore a chain of two rigid-bodies called the *double physical pendulum* (Figure 8.3).

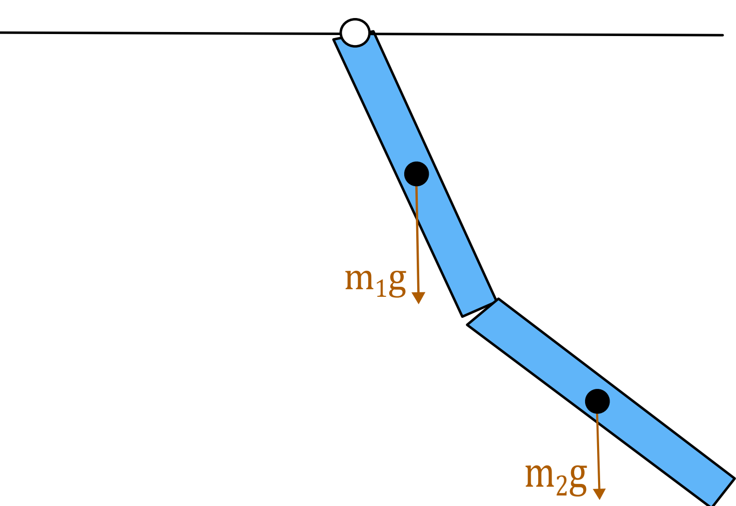


Figure 8.3: A 2-D double physical pendulum

We can begin by manually deriving the equations of motion for a double physical pendulum, just as was the case with the single physical pendulum. After that, we can compare the motion of the system using these derived equations versus the motion of the same system using the articulated-body algorithm. We note that these equations, which can be represented by the two functions f_1 and f_2, will both depend on all the instantaneous positions and velocities of the system:

$$\begin{aligned}\ddot{\theta}_1 &= f_1(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2),\\ \ddot{\theta}_2 &= f_2(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2).\end{aligned} \tag{8.6}$$

These functions can be derived by hand using Lagrangian mechanics, albeit through a lengthy and tedious process. The derivation has been moved to appendix A to keep this section clean. The resulting equations of motion for $\ddot{\theta}_1$ and $\ddot{\theta}_2$ are

$$\begin{aligned}\ddot{\theta}_1 &= \frac{J_x \cos(\theta_1 - \theta_2)\ddot{\theta}_2 + J_x \sin(\theta_1 - \theta_2)\dot{\theta}_2^2 + \mu_1 \sin\theta_1}{-J_a},\\ \ddot{\theta}_2 &= \frac{J_x \cos(\theta_1 - \theta_2)\ddot{\theta}_1 + J_x \sin(\theta_1 - \theta_2)\dot{\theta}_1^2 + \mu_2 \sin\theta_2}{-J_b}.\end{aligned} \tag{8.7}$$

The terms J_x, J_a, J_b, μ_1, and μ_2 are all constants and are described in appendix A. Note that

$\ddot{\theta}_1$ and $\ddot{\theta}_2$ are still in both equations, however, the system can easily be solved by substituting one equation into the other and isolating for the respective $\ddot{\theta}_i$ terms (not shown).

With our derived equations for $\ddot{\theta}_1$ and $\ddot{\theta}_2$, we again proceed by running two simulations in parallel: one using these derived equations, and the other using LSDL. Figure 8.4 shows several snapshots of the simulations. The pendulum on the left is calculating accelerations via LSDL whereas the one on the right is calculating accelerations via the derived equations above. Note that this system also has no plant-motion components, so we are still essentially testing RBDL's base features.

Figure 8.4: Two double physical pendulums side by side. Notice the chaotic motion starting at the 4th frame.

The simulations are visually indistinguishable from each other for the first couple seconds, but at some moment (around $t = 7.20s$), they diverge. In the subsequent seconds, the simulations completely deviate from one another. Therefore, if both algorithms are indeed calculating the same algebraic expressions for $\ddot{\theta}_1$ and $\ddot{\theta}_2$, why do the simulations diverge? Are the algorithms not calculating the same equations for $\ddot{\theta}_1$ and $\ddot{\theta}_2$ after all?

The answer is that whilst it *is* possible that that there are algebraic differences between both algorithms (due to human error), it is significantly more likely that the deviation is due to numerical error instead. This is because the double pendulum is a *chaotic* system, meaning that a small change in its current state can result in large difference in a later state. Since each simulation uses a different sequence of arithmetic operations (addition, subtraction, multiplication, division) on its path to compute $\ddot{\theta}_1$ and $\ddot{\theta}_2$, then - even if both algorithms are algebraically equivalent - there will still be numerical differences present between their calculations. For example, the following is the instantaneous acceleration of $\ddot{\theta}_1$ at $t = 0$ of both pendulums in figure 8.4:

RBDL: $\ddot{\theta}_1$ at $t = 0 : -126.12857413787080$ (rad/s^2).

Custom physics: $\ddot{\theta}_1$ at $t = 0 : -126.12856820663652$ (rad/s^2).

We can see that it is a very slight difference. However, since thousands of iterations are being computed every second, and since each iteration depends on the previous iteration, then the resulting accumulated numerical differences between both simulations will definitely cause the chaotic systems to diverge.

For context, two identical double-pendulum simulations - both operating on LSDL - were left running overnight and did not diverge with respect to each other, meaning that the algorithm is deterministic. After that, three double-pendulum simulations were ran side-by-side, each described as follows:

1. The first simulation used LSDL with normal initial conditions, that is, $\theta_1(0) = \theta_2(0) = \dot{\theta}_1(0) = \dot{\theta}_2(0) = 0$.

2. The second simulation used LSDL with a ten-thousandth radian change in the initial angle of the first body, $\theta_1(0) = 0.0001$, but the other initial conditions unchanged, $\theta_2(0) = \dot{\theta}_1(0) = \dot{\theta}_2(0) = 0$.

3. The third simulation using the derived equations with normal initial conditions, $\theta_1(0) = \theta_2(0) = \dot{\theta}_1(0) = \dot{\theta}_2(0) = 0$.

It was found that simulations (1) and (2) diverged with respect to each other before simulations (1) and (3) did. Since simulations (1) and (2) would never diverge at all without a perturbation, it means that the difference between the LSDL simulation (1) and the derived equations simulation (3) is less significant than pertubing the initial angle of one of the pendulums by one ten-thousandth of a radian: heavily implying that the error is numerical as opposed to algebraic.

8.3 Cantilever beam

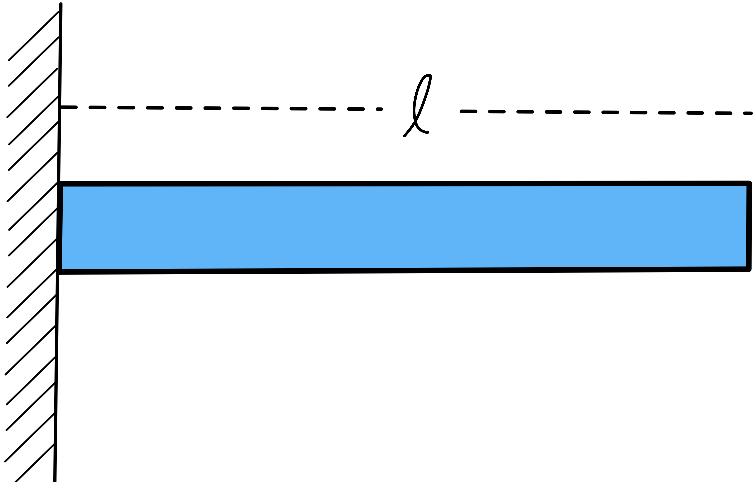


Figure 8.5: A cantilever beam of length l supported on its left end.

A cantilever beam is defined as a slender beam that is fixed at one end while extending horizontally (Figure 8.5). The deflection of cantilever beams resulting from applied transverse loads is an extensively studied topic in the field of structural engineering. In this section, we aim to compare the deflections obtained from our simulation with the expected deflections predicted by established equations. This comparison will help validate the methodology

presented in Chapter 6 for approximating real-world beam deformations via rigid-bodies connected by angular springs.

Let us assume our continuous cantilever to be a cylindrical beam of length l, radius r, density ρ, and modulus of elasticity E. Let us also assume all of these values are constants. Given these conditions, there are several cantilever beam equations that may be used to test our simulations, but we will be picking the scenario in which there is an uniformly distributed load acting on the beam. Additionally, we will take self-load (the load due to gravity) to be the only load acting on the cantilever. The equation for the maximum vertical deflection of such a beam is commonly taught in structural engineering due to its pedagogical nature, and can be derived by integrating the bending-moment equation $M = EIK$ (from which we also derived our spring constants in Chapter 6). This derivation is presented in [32, Chapter 8], and its outcome is the equation

$$\delta_y = \frac{wl^4}{8EI}, \tag{8.8}$$

where

- δ_y is the maximum vertical deflection (from the horizontal) of the beam's floating tip.
- w is the uniformly distributed load per unit length acting along the beam. We will simply take this to be the length-normalized force of gravity, $w = \frac{F_g}{l} = \frac{mg}{l}$.
- l is the length of the beam.
- E is the Young's modulus of the beam.
- I is the second moment of area of the beam's cross-sectional area. For circular cross sections as in our example, $I = \frac{1}{4}mr^4$.

It is important to note that this equation only works for *small* deflections. The reason for this is that the assumption was made that the cantilever would only experience small

curvature when carrying out the integration over $M = EIK$. This is because otherwise, the system would be non-linear and would not have a nice, closed-form solution as in Equation 8.8. This is analogous to how a pendulum's angle as a function of time only exhibits a closed-form solution if the small-angle approximation is assumed. In order to conform to such assumptions, the cylindrical cantilever being simulated has the following parameters: $l = 1.0\text{m}$, $r = 0.01\text{m}$, $E = 8100\text{MPa}$, and $\rho = 923\text{kg/m}^3$ for the cylinder's density. The values are arbitrary and were only chosen such that the cantilever's expected deviation would be much smaller than its length. By plugging these values into Equation 8.8, we see that the expected deflection of the free tip of the beam is

$$\delta_y = \frac{wl^4}{8EI} = 0.0055893\text{m},$$

which is miniscule compared to its length of 1.0m, meaning the use of the equation is warranted. The deflection of such a cantilever beam was simulated by discretizing the metre-long continuous beam into N_B rigid-bodies. The N_B rigid bodies were connected together by revolute springs whose spring constants were computed via the methods discussed in Chapter 6. The simulations were left oscillating in an underdamped status until they reached static equilibrium. It was at this point that the vertical deflection of the cantilevers' end point was measured. The following results were obtained with N_B = 10, 25, 100, 250, and 500:

n	simulated δ_y(m)	expected δ_y(m)	deviation(m)
10	0.0067664	0.0055893	0.0011771
25	0.0060488	0.0055893	0.0004595
100	0.0057051	0.0055893	0.0001158
250	0.0056376	0.0055893	0.0000483
500	0.0056155	0.0055893	0.0000262

We can see that doubling the number of bodies results in roughly half the amount of error, implying that the computed deviation does indeed converge towards the expected deviation.

More rigorous testing would be required to get further information regarding the correctness of the spring constants, for example, would the simulation 'overshoot' the expected deviation as N_b gets much bigger? What about validating the deformation scheme against large deformations? For the purposes of plausible plant animations at the macroscopic scale however, these results are promising.

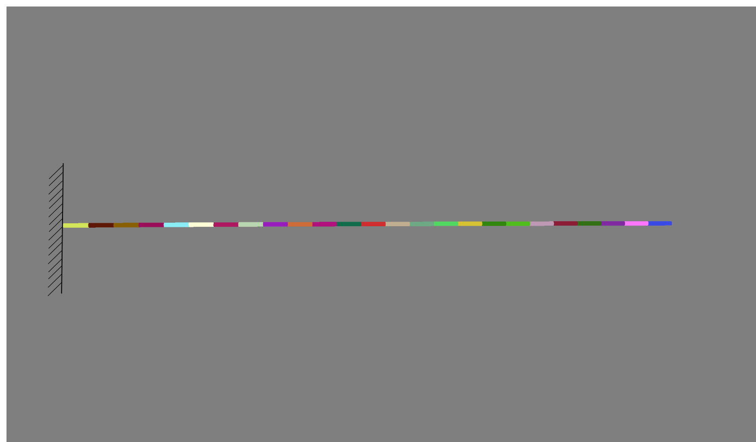


Figure 8.6: A cantilever beam simulation with 25 bodies.

Figure 8.6 shows how the cantilever represented by 25 rigid bodies looks after it has reached static equilibrium. Note that the length of the beam is 1m, so a deflection of 0.006m is not easily noticeable to the naked eye.

Efficiency and stability of the cantilever simulations

One might've wondered the following question: *Why didn't you test the cantilever model against 100,000 bodies?* The answer is that the simulations get *significantly* slower as the fidelity of the model increases. The remainder of this discussion was moved to Section 9.4.2 in order to join all discussions on efficiency and stability.

Chapter 9

Results

The methodology employed in this thesis for creating procedural plant models and simulating their dynamics has been discussed. This chapter presents some sample animations illustrating the results of these methods.

9.1 Plant simulations - monopodial structures

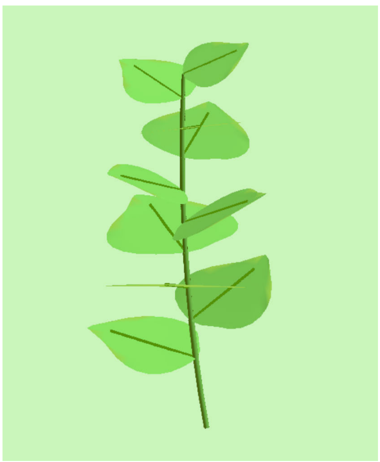


Figure 9.1: A monopodial plant structure modeled via an L-system.

The biological structure of many simple yet elegant plants can be described as a single-stemmed structure with leaves branching out in phyllotactic patterns. They are called

monopodial plant structures and can accurately be modeled with L-systems. Figure 9.1 illustrates an L-system object of an abstract monopodial plant model whose leaves are branching out in a *spiral* phyllotactic pattern. This section will be dedicated to showcasing several sample animations of monopodial plant structures. It should be noted that this section and future ones will not discuss the techniques used to model the illustrated plants because - other than the fact that they are modeled with an articulated-body L-system format in mind - the underlying L-system modeling techniques employed all fall under previous work (e.g. [69, Chapter 3]).

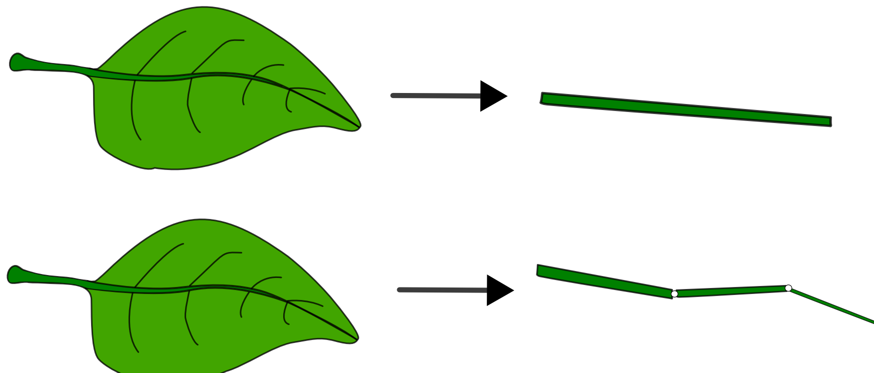


Figure 9.2: (top): A leaf being approximated by a single cylindrical internode. (bottom): A leaf represented by several cylindrical internodes.

This section is instead concerned with showcasing the methodology presented in this thesis through the simulation of monopodial plant structures. There is one problem that needs to be addressed first, however. The leaves in Figure 9.1 are not cylinders, meaning that we can not use the methodology from Chapter 6 to calculate the spring constants of its connecting node. Actually, it is simply not obvious what the springs constants should be between rigid bodies of abstract shapes. In order to bypass this problem, the dynamics of any non-cylindrical organs of a plant such as the leaves and flowers will be approximated by cylindrical rigid-bodies (e.g. Figure 9.2, Figure 9.3). All the leaves in my plant models are approximated by a single rigid body.

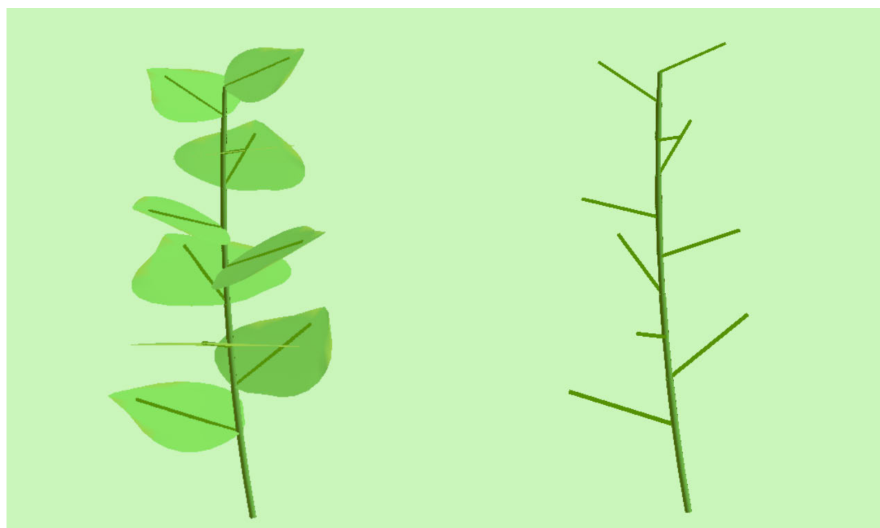


Figure 9.3: (left) The plant that is rendered versus (right) the actual appearance of its articulated-body.

All the shape and material properties used in my models were adapted from [56]. In the study, Niklas determines the following properties of 76 herbaceous plant species:

1. Stem radius (m).
2. Plant height (m).
3. Bulk tissue density ρ (kgm^3).
4. Young's modulus E (MPa).

The material properties in this study provided a good basis with which to calculate the following parameters of the plant models:

1. The length, radius, and density of the cylindrical rigid-bodies composing the plant's stem.
2. The elasticity properties (E/G) and second moment of area I needed to calculate the spring constants between the cylindrical rigid-bodies composing the plant's stem.

The parameters related to the rigid bodies and spring constants for other organs such as leaves and plants were also adapted from the study, but were manually adjusted to better

represent the motion of the respective organ. For example, the rigidity of a spring connecting a leaf and an internode was generally down-scaled by an user-defined constant because the connection between a leaf and an internode should be more flexible than the connection between two internodes. It remains as future work to find an automatic way to find the spring constants for connections between abstract types of organs/rigid-bodies.

9.1.1 Example: gravity

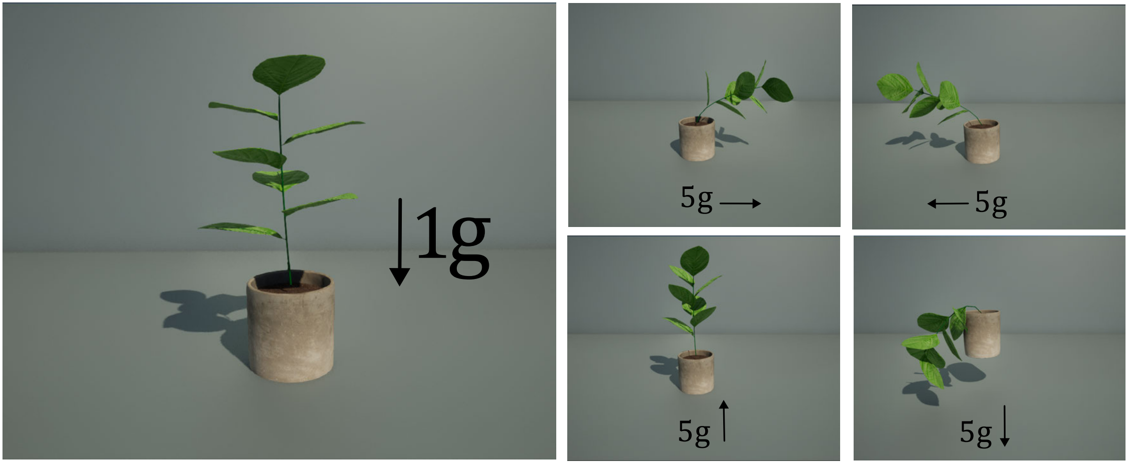


Figure 9.4: (left): The rest pose of a broad-leaf plant experiencing normal gravity. (right): The rest poses of the same plant if it were to experience the 5x force of gravity in other directions.

Gravity can be implemented by applying a downwards gravitational force at the center of mass of each rigid body in the system. However, a different approach is typically used with Featherstone's algorithm; since the algorithm excels at the propagation of accelerations, we can set the acceleration of the immovable root body to 9.81m/s^2 upwards, meaning that the rest of the articulated-body will experience it as an acceleration going 9.81m/s^2 downwards. Figure 9.4 showcases the effects of varying gravitational force on a simple broad-leaf plant.

9.1.2 Example: moving base

A key motivator behind using Featherstone's algorithm for the dynamics of plants is its ability to properly capture non-inertial accelerations, which is is an essential component of real plant motion. An easy way to showcase this is by accelerating and decelerating the base of the plant such that all resulting motion is non-inertial. The problem is that the current implementation does not allow for a movable base.

In order to proceed, we need to extend the articulated-body algorithm to account for a floating base. A floating-base system is one in which the base is a moving body, and can be implemented by installing a 6-DoF joint between the root body and the origin of the world [53, Chapter 4].

Figure 9.5: Simulating the movement of a potted plant by only accelerating its base. (left column): rightwards and then leftwards. (middle column): forwards and backwards. (right column): counter-clockwise motion.

With a floating-base implementation at hand, we can proceed by representing the now-movable base with a pot, and moving the pot around the scene to see the resulting plant motion (Figure 9.5). Notice how the plant jerks opposite to the direction of acceleration,

this can most clearly be seen in the left column pictures.

The acceleration and deceleration of the pot is controlled by user input. This is done by defining two user constants: one for the magnitude of acceleration of the pot (due to user input), and the other the magnitude in deceleration of any already-moving pot (mimicking deceleration due to kinetic friction). Additionally, the arrow keys are mapped to different directions of acceleration; pressing an arrow key accelerates the plant in the respective direction, and releasing the key stops acceleration in that respective direction. The plant decelerates automatically because deceleration is always being applied to a moving plant, proportionate to the deceleration constant. The deceleration constant mimics the roughness of the terrain on which the pot slides.

9.1.3 Example: pulling

The two previous examples (gravity and movement of the base) are examples of dynamical plant motion arising only from non-inertial effects. We are now interested in plant motion arising from physical interactions as well. This includes environmental forces such as wind, rain, wildlife interaction, and human interaction. All of these forces together make up the external force vector, $\mathbf{f_{ext}}$, which is an input into Featherstone's algorithm. In this section, we shall showcase such motion by pulling a virtual plant.

Pulling was implemented through the use of a 3-D cursor; the cursor is used to virtually 'click and pull' desired parts of the plant. The main issue here is that a mouse can only map 2 dimensions, so we used a 3-D haptic device to control the movement of the 3-D cursor.

The algorithm to do so is as follows:

1. **When the cursor button is pressed**: Store the cursor's current 3-D position, $\mathbf{p_o}$,

as well as the closest rigid-body to this position, b. b is the rigid body that will be pulled.

2. **While the cursor button is held**: The linear force $\mathbf{F}$ acting on b is equal to the current position of the cursor minus $\mathbf{p_o}$ (times some user-defined constant). Add this linear force to the elements of $\boldsymbol{f}_{ext}$ corresponding to b. We may also set the haptic feedback force to be the opposing force, $-\mathbf{F}$, if desired.

3. **When the cursor button is released**: Set the corresponding elements of $\boldsymbol{f}_{ext}$ to $\mathbf{0}$.

Figure 9.6 provides snapshots of a monopodial plant being pulled at its tip.

Figure 9.6: (top row): progressively pulling a small herbaceous plant near at its tip. (bottom row): the resulting motion after releasing.

9.1.4 Example: modifying elasticity

The presented methodology excels in its ability to capture the motion of plants undergoing different material properties. As an example, Figure 9.7 shows different snapshots of a rose whose Young's modulus has been severely reduced, resulting in the rose being unable to support its own weight. Such techniques could be used to simulate the motion of withered

or damaged plants.

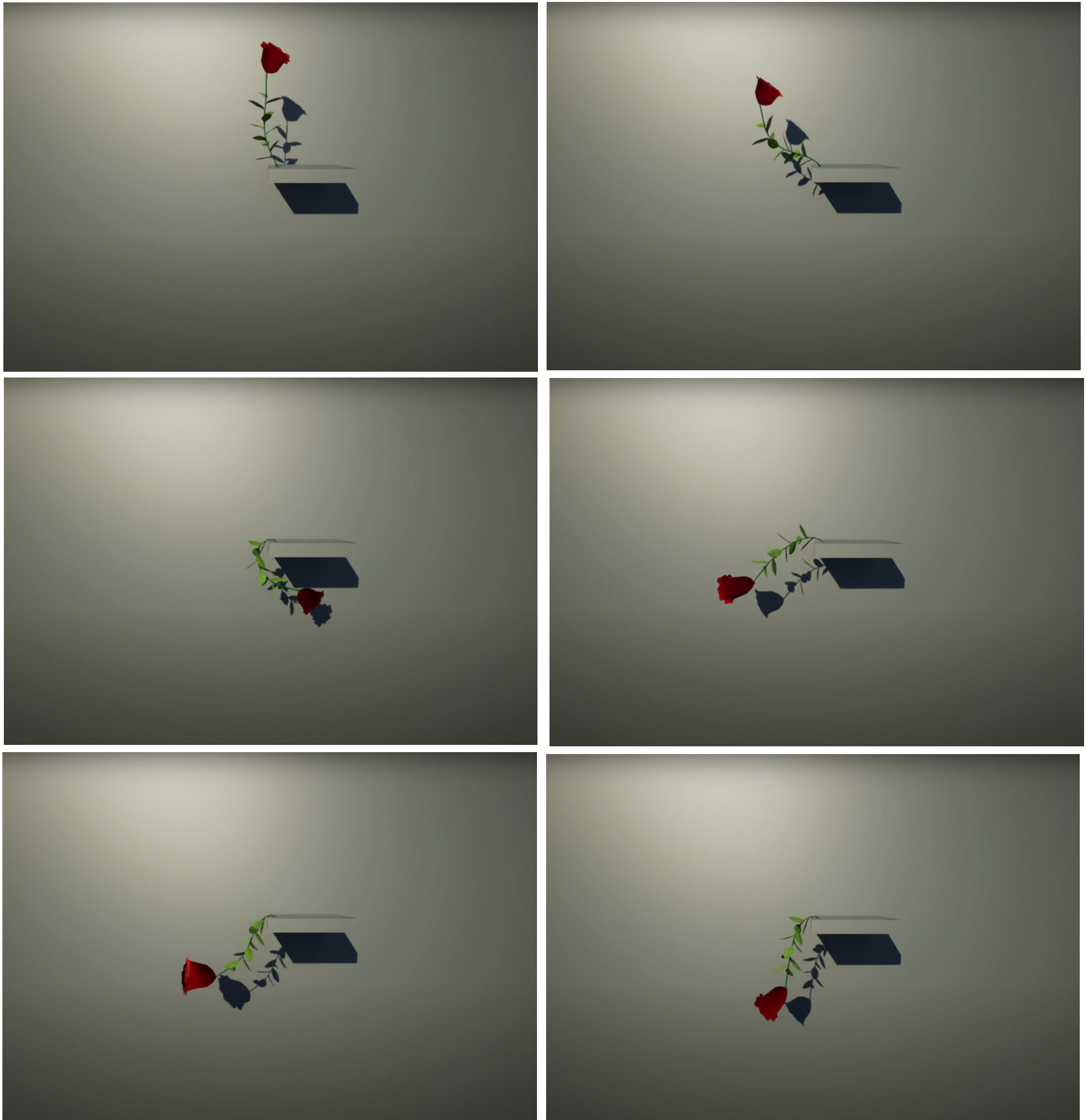


Figure 9.7: A rose with reduced Young's modulus falling under its own weight.

9.1.5 Example : motion comparison against a real monopodial plant

The previous examples showcased the proposed methodology by analyzing specific plant motions, however, the plant models themselves were still abstract. I therefore also compared

the proposed methodology against real plant motion.

The comparison experiment was carried out by first capturing a video of a real plant being pulled, and then modeling and animating the same plant motion thereafter. The plant in question was purchased from a local supermarket, and has a monopodial structure with leaves branching out in a spiral phyllotactic pattern (Figure 9.8).

Figure 9.8: Side and top views of a tropical plant.

The articulated-body L-system string of the plant model was constructed manually in order to get as accurate a model as possible. This was done by measuring the individual components of the plant and creating rigid-bodies out of these measurements. The stem was approximated with eight cylindrical rigid-bodies, and each leaf was approximated by a single cylindrical rigid-body. The densities and spring constants of the internodes were adapted from Niklas's measurements [56] just as the previous examples, but again, the densities and

spring constants of the leaves required further alteration on a trial-and-error basis in order to make their motion realistic. Figure 9.9 provides several snapshots of the real and synthesized motion side-by-side.

Figure 9.9: Comparison of real and synthesized plant motion. The plant is first pulled (frames 1-2) and released (frame 3).

One can see from Figure 9.9 that, if fed the proper shape and material properties, the proposed methodology can create surprisingly accurate motion, and actually, the extent to which the animations matched was completely unexpected due to all of the uncertainties and approximations employed in the thesis. A direct future work consists of working on an automatic way to determine these material properties (a closely related previous work in this area is [93]).

9.2 Plant simulations - structures with higher-order branches

Figure 9.10: Growth time stamps of an inflorescent plant. The yellow spheres indicate flower placement and size.

We also explored the animations of herbaceous plants with inflorescences. This is because their motion is typically interesting and unique due to the propagation of motion from one branch to another. The L-system modeling techniques used to modeled such plants are those presented in [68]. In the paper, a single developmental model is proposed that accounts for the restricted range of inflorescence types observed in flowering herbaceous plants. Additionally, many flowering plants first grow a main stem with leaves (vegetative stage) before developing their flowers (flowering stage). We can therefore model a great variety of flowering herbaceous plants by first growing a main monopodial stem (representing the vegetative stage) and developing an inflorescence thereafter (representing the flowering stage). Figure 9.10 shows snapshots of a growing abstract inflorescent plant model and Figure 9.11 indicates the vegetative and flowering parts of the plant.

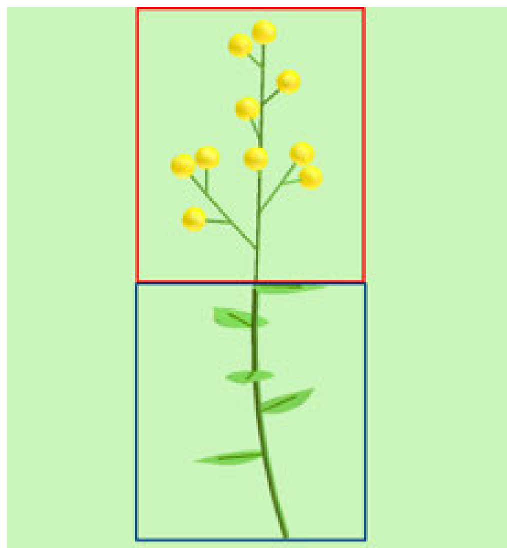


Figure 9.11: Many flowering plants can be modeled as a vegetative part (blue square) and a flowering part (red square).

9.2.1 Example: twisting motion

Twisting can occur when different forces (wind, gravity) act on asymmetric structures, causing torques. It can also occur when the forces are asymmetric themselves. Twisting motion is not as visually prevalent as bending motion because plants are generally harder to twist than to bend, meaning that bending motion visually overwhelms twisting motion. Therefore, in order to showcase the methodology's ability to represent twisting motion, we can proceed with the following contrived example:

1. Apply a slight external twisting torque along each internode of the plant's main stem. This torque could be the result of asymmetrical environmental forces, but for our purposes, its magnitude is an user-defined constant.

2. Continue applying the torques until the plant settles (static equilibrium).

3. Release the torques. This will instantaneously be followed by the plant 'swinging back' towards its rest pose.

Twisting can most easily be observed in plants with laterally extending branches, which is why I chose to carry out the example above against a ‘T’-shaped inflorescent plant. Figure 9.12 shows front and top-down views of a ‘T’-shaped plant’s response to the axially-applied external torques, and Figure 9.13 shows the resulting oscillatory motion directly after the time of torque release.

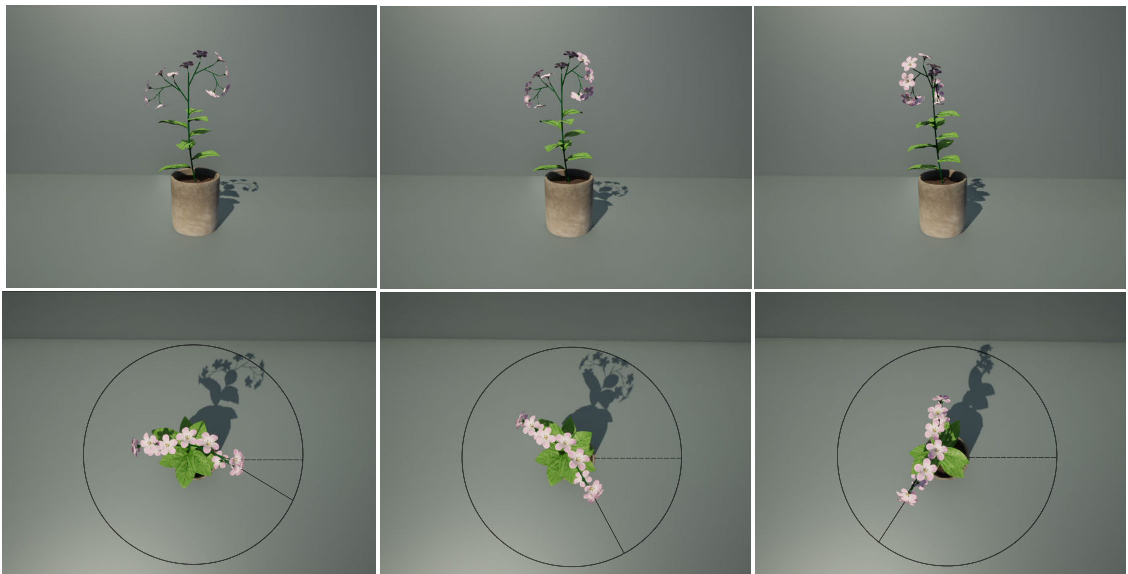


Figure 9.12: (top row): twisting a plant in incremental amounts. (bottom row): top-down view of the same motion, the twist amount is represented by the angle between the dashed line and solid line.

One important thing to point out from the angular motion showcased in this section is that straight lateral branches will take on a curved shape when the plant twists. This can be explained through the angular notion of rotational inertia, which states that mass gets progressively harder to rotate the further away it is from the axis of rotation. In the angular motion showcased in Figures 9.12 and 9.13, the axis of rotation of the system as a whole is approximately equal to the longitudinal axis of the main monopodial stem. This means that it will get progressively more difficult to angularly accelerate rigid bodies the further away they are from the main stem, which leads to motion in which the distal bodies will always be ‘lagging behind’ the proximal bodies. Therefore, lateral branches will appear curved if

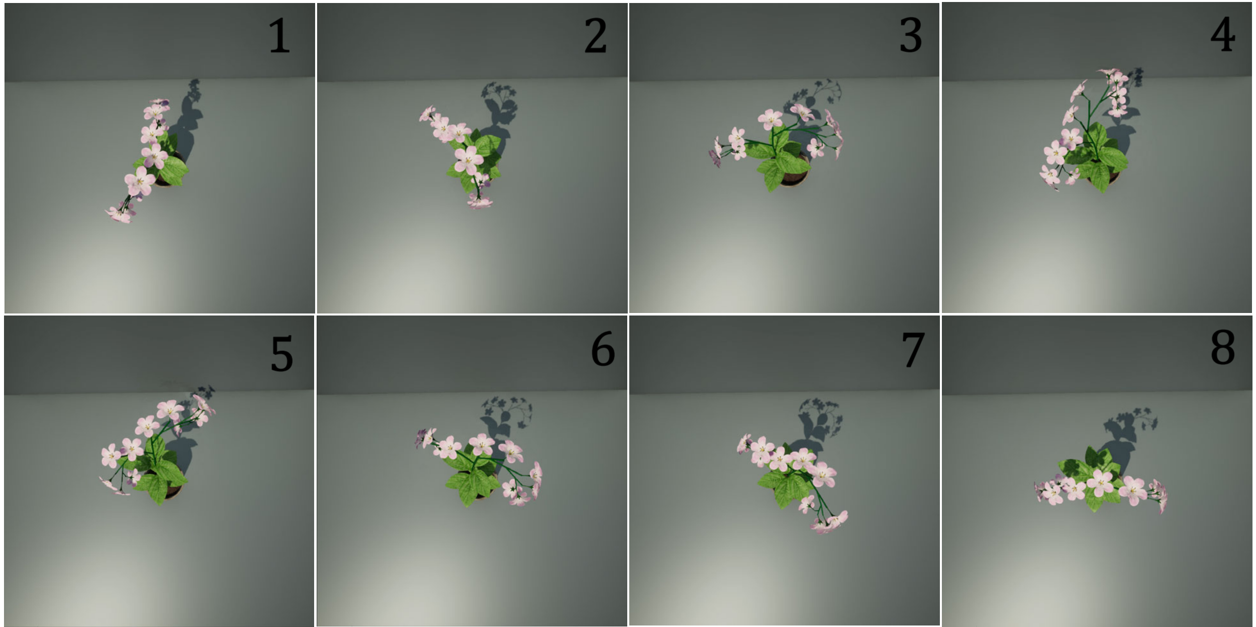


Figure 9.13: Snapshots showing the oscillatory motion of a twisted plant at the time of release.

observed from a top-down-view, which I find intriguing because we are essentially visualizing the moment of inertia at play! Figure 9.14 showcases the same simulation as in figures 9.12 and 9.13, but this time the plant has exaggeratedly long lateral branches such that the curved shape is more obvious.

Figure 9.14: (left): An inflorescent plant with exaggerated laterally-extending branches at rest. (middle): The plant's main stem is twisted in the direction of the arrow. (right): The plant's curved appearance shortly after the release of the external torques; the arrows denote instantaneous direction of motion.

9.2.2 Example: the importance of shape

The motion of a plant is heavily influenced by its shape. In these simulations, shape refers to the length and radii of the plant's cylindrical internodes. Figure 9.15 depicts the effects that varying radii (top) and varying lengths (bottom) have on the range of motion that a plant may have. Thicker plants bend less because internodes with larger cross-sectional areas are more difficult to bend, whereas longer plants bend more because the angular springs will struggle to restore the relative orientations between internodes whose mass is concentrated further away from the joint.

9.2.3 Example : motion comparison against a real branching plant

Finally, the motion produced by the proposed methodology was compared against the motion of a real branching plant. This was done in an identical manner to the monopodial plant comparison example from Section 9.1.5.

The plant chosen for this example was a Phalaenopsis Orchid as pictured in Figure 9.16. It was picked because of its two distinctive flowering lateral branches, meaning that unique and complicated motion will arise from the transfer of motion from one branch to the other.

The comparison experiment was again carried out by first capturing a video of the real orchid plant being pulled, and then modeling and animating the same plant motion thereafter. The orchid model was constructed manually in order to get as accurate a model as possible, which was again done by measuring the individual components of the plant and creating rigid-bodies out of these measurements. Motion comparison of both real and synthesized animations is provided in Figure 9.17.

One can see from Figure 9.17 that the virtual motion closely follows the real motion despite all the chaotic motion.

Figure 9.15: (top): Three plants experiencing equal bending torques. The radii of the plants vary ascendingly from left to right, but the plants are otherwise identical. (bottom): The plants are again experiencing equal bending torques. The lengths of the plants vary ascendingly from left to right, but the plants are otherwise identical.

Figure 9.16: Side and top views of the orchid.

Figure 9.17: Comparison of real and synthesized orchid plant motion.

Finally, for context, both the orchid comparison video in this section and the tropical plant

comparison video from Section 9.1.5 each took about ∼24 hours to complete:

- ∼12 hours modeling and texturing the plant's flowers, leaves, and internodes using Blender [12].
- ∼4 hours modeling the plant's L-system. The dimensions the rigid-bodies were determined empirically in order to create matching models.
- ∼4 hours creating and setting up shading materials, lighting conditions, camera settings, and other scene-related components in the Unreal Engine 5 [30] editor.
- ∼2 hours setting up the timeline editor for the pulling motion, and using it to re-create the motion in the video. The main components that required trial-and-error in order to match in these comparison simulations were the damping and spring constants for leaves and flowers.
- ∼2 hours of post processing (positioning the videos side-by-side, scaling them accordingly, making sure their animation times are aligned).

9.3 Model parameters

The purpose of this short section is to disclose exactly the number of tunable parameters in my models. For starters, every cylindrical rigid-body needs exactly three parameters:

1 scalar for the length of the cylinder.

1 scalar for the radius of the cylinder.

1 scalar for the density of the cylinder.

Furthermore, every spherical joint requires the following parameters:

1 scalar for the Young's modulus that will be used to generate the joint's spring constants.

3 scalars for the three default joint angles of the joint.

3 scalars for the three initial joint angles of the joint.

1 scalar for the damping constant of the joint.

It is possible to have a different Young's modulus and damping constant in each direction of rotation, but I did not make use of this fact. Every single one of these parameters must be given if the model is created by hand, which means there are $11 \times N_B$ parameters. This is definitely time consuming even for small herbaceous plants, and poses many problems when working with larger models as pointed out in [93] in their work on procedural parameters. However, a procedurally-grown plant generally only requires a single scalar for every parameter type (i.e. 'maximum internode length'), and uses this single parameter in conjunction with a developmental rule to generate the lengths of all internodes. This means that the actual number of parameters needed is substantially lower than $11 \times N_B$.

Finally, there are two global parameters pertaining to the simulation:

1 scalar for the timestep dt.

1 scalar for the number of physics substeps.

9.4 Accuracy and efficiency

An important question laying at the heart of the thesis is to verify how useful and appropriate it is to use Featherstone's algorithm for animating plant motions, and to compare how it performs against other methods. The purpose of this section is to address these questions, and does so by taking into account the results presented throughout this thesis.

The discussion is best broken down into two: a discussion on *accuracy* and a discussion on *efficiency*. Accuracy refers to Featherstone's algorithm ability to approximate real plant motion, whereas efficiency refers to how quickly and reliably the simulations can be performed.

It is important to note that the model I've presented is not a perfect implementation of a 'physically-based plant model operating on Featherstone's algorithm' (whatever this 'perfect' model may be). This is because - due to the complex nature of the project - several approximations were employed that might have affected the potential accuracy and efficiency of the method; could a quaternion representation of the joints have provided significant stability and hence efficiency? How about a different integration scheme or damping? It is therefore possible that these approximations could skew the discussion of 'the suitability of Featherstone's algorithm for plant motions', but I am nevertheless confident that the *overall* strengths and weaknesses of the method in representing plant motion have been clearly identified through the work on this thesis.

9.4.1 Accuracy

In the context of this thesis, accuracy refers to how well the Featherstone model is able to approximate real plant motion. An accurate model would closely match the actual motion of a plant, while an inaccurate model would differ significantly from the real motion. By construction of the problem, the accuracy of any physically-based method may be decomposed into two components:

1. the accuracy of the discrete model employed, and
2. the accuracy of the simulation method employed;

both of which are immediately discussed.

Accuracy of the discrete plant model

By the 'accuracy of the discrete plant model', we refer to the faithfulness by which the virtual, discrete plant model approximates the real, continuous plant. This depends on two things: the discretization scheme employed (e.g. 1-D rigid links? 3-D rigid bodies? 3-D finite elements?) and the *precision*, or granularity, of the models (i.e. the actual number of discrete elements used); we are not yet concerned with the accuracy of the simulation method itself.

In any physically-based animation project, the chosen discretization scheme typically dictates the range of motions replicable by the model, regardless of simulation scheme used. Chapter 2 reviewed some common discretization schemes that have been used in plant animation works. For example, by using 3-D spherical springs, the presented model can adequately capture torsional plant motion (Section 9.2.1) whereas previous models that only use 2-D bending springs cannot (e.g. [79, 64]). On the contrary, the presented model cannot handle shearing deformations (as discussed in Chapter 6), whereas methods derived from 3-D continuum mechanics generally can (e.g. [6, 93]).

The accuracy of the discrete plant model also depends on its granularity or *precision*. In our context, a more precise model would include more rigid-bodies and more spherical springs in order to better approximate the real plant motion. The cantilever example for instance (Section 8.3), showed how the accuracy of the model increased as the number of rigid-bodies increased.

Accuracy of the simulation method

By the 'accuracy of the simulation method', we refer to a simulation method's ability in approximating real motion *with respect* to the inaccuracies that have already been forfeited due to the chosen discretization scheme. In our context, we are interested in the extent

by which Featherstone's algorithm can be used to make articulted-bodies *bend* and *twist* like real plants; these being the specific motions that can be captured by the discretization scheme used in this thesis.

The accuracy of a simulation method is a complex and multi-dimensional issue, meaning that it is unlikely that a single metric may be used to assess the accuracy of the entire model. This is because the accuracy of the simulation method depends on the implementation details of *all* its components such as external forces, spring constants, damping forces, and the time integration scheme employed; many of which depend on each other.

One can instead proceed by assessing the accuracy of a physically-based simulation method in *pieces*, and there are two common ways that this is done:

1. by using analytical solutions to compare the simulations with expected behaviours, or

2. by comparing the simulations with real-world observations.

The tests performed in Chapter 8 provided an analytical notion of accuracy; the double-pendulum example (Section 8.2) showed Featherstone's ability in capturing chaotic motion, whereas the cantilever example (Section 8.3) showed the method's ability in representing the deformation of real objects. The ability of Featherstone's algorithm to capture secondary motion as analyzed in the double-pendulum example (and observed in the moving-base example) should not be understated, as it is the key reason why the method can capture the intricate and subtle motion of real plants.

On the other hand, the visual results presented in this chapter provided an observational notion of accuracy. **It is my impression that the motion comparison results presented in Sections 9.1.5 and 9.2.3 speak for themselves in terms of the accuracy achievable by Featherstone's algorithm**. They show that - when fed appropriate shape and

material parameters - Featherstone's algorithm is capable in creating highly accurate plant motion. The other motion examples showcased in the chapter were not compared against real plants, but nevertheless showcase the method's ability in representing a plethora of motion patterns observed in real plants.

9.4.2 Efficiency

Efficiency refers to how quickly and reliably the Featherstone's algorithm plant-simulations can be performed, and in short, **Featherstone's algorithm struggles in efficiently simulating the dynamics of large and/or rigid plants**. This was without a doubt the key hurdle that complicated every step of the thesis' development.

The problem can be constructively elaborated upon by continuing the discussion on the efficiency and stability of the cantilever beam simulation (Section 8.3). In the example, the deformation of a unit-length cantilever beam bending under gravity was analyzed by modeling it with an increasing number of rigid-bodies, up to a maximum of 500. The question was: why wasn't the experiment tested with a much larger number of rigid-bodies?

The answer was that the simulations get *significantly* slower as the fidelity of the geometric model increases. **The reason for this is *not* that Featherstone's algorithm is slow**; Featherstone's algorithm is O(n) where n is the degrees of freedom of the system, and therefore its speed in calculating $\ddot{\mathbf{q}}$ is linearly proportional to the number of generalized coordinates in the system[1]. For context, any plant model presented in this chapter can run over one-hundred thousand iterations of Featherstone's algorithm per second and still maintain interactive frame-rates (30 FPS+).

[1] It can be parallelized, but the time increase would depend on the branching structure of the tree.

The speed issues instead emerge because Featherstone's algorithm - if used to simulate the dynamics of elongated structures - is extremely *unstable* [22]. An unstable computational algorithm is one in which a small change in the input may result to large changes in the output. This is problematic because we are using the output of the algorithm $\ddot{\mathbf{q}}$ to estimate the state of the system at a point later in time; in an unstable system, smaller time steps must be taken in order to accurately approximate the real motion of the plant, hence the speed issues. This means that the inefficiency of the simulations lies in the attempt to numerically represent highly non-linear motion, as opposed to an inherent flaw with Featherstone's algorithm itself.

It should be noted that this issue is catastrophic because a small error in one iteration will lead to a larger error in the next one; since tens to hundreds of thousands of iterations are being computed per second, the simulation will explode in the blink of an eye.

For example, the 500-body cantilever simulation required a time step dt in the *millionths* of a second to be numerically stable. This instability can be conceptualized by considering the analytical leap in complexity from a one-body physical pendulum and a two-body physical pendulum (as analyzed in Chapter 8), and then considering that our example above is essentially a 500-body physical pendulum with the additional instability coming from the 500 stiff springs connecting it. The following chart shows the maximum stable time step dt for the cantilever tests from Section 8.3:

n	maximum stable dt(s)
10	0.0002
25	0.00003
100	0.000001
250	0.0000003
500	0.00000007

We can see that the time step needed to remain stable is *not* linearly proportional to n. This drastic increase in complexity is known, and a more comprehensive study on the topic is presented by Featherstone in [22]. In general, the more complicated an articulated-body is, the more unstable its dynamics simulation becomes meaning that a smaller time step must be taken, hence the slow simulation speeds. Complexity in this context refers to the number of bodies in the system, the geometry of the system (that is, the shape and size of the rigid-bodies and where things are connected), and the rigidities of the spring constants in the system.

These numerical instabilities are the reason why the methodology presented in this thesis can only simulate plants of relatively small size in real-time. For context, the plant simulations in this chapter had values of n between $\sim$50 and $\sim$100. It was found that a plant model with $n \approx 100$ was around the largest a model could get whilst still maintaining stable interactive speeds. A performance evaluation of RBDL, including a comparison against another state-of-the-art multibody dynamics library (SimBody [81]), is provided by its creator Martin Felis in [28].

Measuring simulation speeds

We conclude the topic of efficiency by discussing the speeds of the plant simulations presented in this chapter. Let us define a *physics step* to consist of a single call to the method `PlantModel::PhysicsStep(float dt)` as presented in Section 7.2.2 (presented again in Algorithm 8 for convenience).

Furthermore, let us define a *rendering step* to be the application-specific method that renders a plant model to the screen, such that the pseudocode of the plant simulatior takes on its 'canonical' form as shown in Algorithm 9.

Algorithm 8 Pseudocode for the PhysicsStep method

```
1: procedure PlantModel::PhysicsStep(float dt)
2:     ComputeForces()
3:     ForwardDynamics()
4:     TimeIntegrate(dt)
5:     ComputePositions()
6: end procedure
7:
8: procedure PlantModel::TimeIntegrate(float dt)
9:     q̇ ← q̇ + q̈ · dt
10:    q ← q + q̇ · dt
11: end procedure
12:
```

Algorithm 9 The canonical form of the plant simulator.

```
1: procedure main(string lsystem_string)
2:
3:     PlantModel m ← new PlantModel(lsystem_string)
4:     float dt ← user constant
5:
6:     while true do
7:         m->PhysicsStep(dt)
8:         m->RenderingStep()
9:     end while
10:
11: end procedure
```

We note that both the PhysicsStep and RenderingStep methods are $O(n)$, which means that any modern system is able to simulate the plant simulations at interactive frames, even for large values of n such as 100 000. However, since the value of dt has to be so tiny in order to maintain stability, the simulated plant will appear to be moving in slow-motion when compared to its real counterpart. The common (and employed) way to circumvent this is to run many PhysicsStep iterations per RenderingStep iteration (Algorithm 10). This will jeopardize the visible frames-per-second (FPS) of the simulation, but can greatly increase the speed of motion. The speed of the simulations can thus be quantified by the two following co-dependent metrics:

FPS: The number of visible frames-per-second (FPS).

relative_speed: How fast the plant moves with respect to the real plant.

Algorithm 10 The canonical form of the plant simulator that incorporates physics sub-stepping.

```
 1: procedure MAIN(string lsystem_string)
 2:
 3:     PlantModel m ← new PlantModel(lsystem_string)
 4:     float dt ← user constant
 5:     int substeos ← user constant
 6:
 7:     while true do
 8:         for i = 1 to substeps do
 9:             m->PhysicsStep(dt)
10:         end for
11:         m->RenderingStep()
12:     end while
13:
14: end procedure
```

The metrics FPS and relative_speed are related by the equation

$$\texttt{relative_speed = substeps * FPS * dt}, \tag{9.1}$$

where the variables of the equation are described as follows:

relative_speed (unitless): The relative speed by which the simulated plant appears to move with respect to the real plant. A relative_speed of 2 would mean that the simulated plant motion will appear to move twice as fast as the real plant motion.

substeps (unitless): The number of physics steps per rendered frame.

FPS (s^{-1}): The number of visible rendered frames per second.

dt (s): The constant delta time by which the system always advances its state. This generally has to be exceptionally small to maintain stability.

The substeps and dt constants are user-defined. The values for relative_speed and FPS are a by-product of these constants (FPS also depends on the time it takes to process input and render the scene, but are assumed to be uncontrollable for the context of this discussion). The value for dt was chosen first in any particular simulation, and it was chosen to be the largest value such that the simulation will be numerically stable. The value dt generally needs to be small, which means that relative_speed will be small as a consequence. In order to combat this, substeps needs to be raised so that more physics simulation steps are computed per rendered frame. Raising the value of substeps will affect FPS, however, meaning that substeps can't get too high otherwise FPS may fall below interactive levels. The values for substeps used in the simulations in this chapter varied between 100 for simple models and 2000 for the most complicated ones. Additionally, I aimed to have a minimum FPS of 30 for all my simulations.

Figure 9.18 provides performance metrics of various plant simulations presented in this chapter. The plant simulation with $n = 99$ was found to be the largest possible such that it could achieve a relative_speed close to 1. The orchid simulation that has $n = 150$ could only achieve a maximum relative_speed of 0.67, meaning that at best, its real-time simulation will move two-thirds the speed of the real orchid.

	n	dt	substeps	FPS	relative speed
	45	4.5x10^{-5}	135	render only 165 physics only 1066 Combined 165	without rendering 6.478 with rendering 1.002
	51	4.0x10^{-5}	151	render only 165 physics only 838 Combined 165	without rendering 5.064 with rendering 0.997
	63	3.0x10^{-5}	201	render only 165 physics only 522 Combined 165	without rendering 3.151 with rendering 0.995
	99	1.5x10^{-5}	480	render only 165 physics only 142 Combined 135	without rendering 1.028 with rendering 0.976
	150	1.5x10^{-5}	1500	render only 165 physics only 30 Combined 30	without rendering 0.691 with rendering 0.671

Figure 9.18: A table showing performance metrics of several simulations presented in this chapter.

Chapter 10

Conclusion and future work

This thesis presented a physically-based model for simulating the dynamics of procedurally-generated L-system plants. The key programming component implemented for the completion of this thesis is the *L-system dynamics library* (LSDL), a general purpose object-oriented C++ library that allows one to simulate the dynamics of abstract L-system plants via Featherstone's articulated-body algorithm. LSDL was validated by using it to simulate the dynamics plants with varying structures and material properties, including side-by-side comparisons of real-world plant motion. LSDL was also tested against a series of physical tests that provided quantifiable validation of correctness. Additionally, the presented simulations were subjected to various types of real-time user interactions, showcasing the versatility of the proposed methodology. The resulting plant simulations were shown to capture the flexibility and liveliness of real plant motion, including seldomly modeled movements such as twisting and secondary motion. Additionally, it was specifically pointed out how the articulated-body algorithm's ability to capture non-inertial effects heavily contributed towards the realism of the synthesized motion. Finally, the approximations and heuristic assumptions present in the methodology were pointed out throughout the thesis in order to clarify the contributions and limitations of the presented work. All in all, the work presented in this thesis has shown to be a promising stepping stone towards an unified model in the

physically-based modeling and animations of procedurally-generated plants.

The remainder of this chapter lists (some) of the many research problems that arose throughout the development of this thesis.

10.1 Future work: accuracy of the model

The first category of research problems consists of problems related to making the articulated-body plant motion more realistic. An attempt has been made to sort them ascendingly by perceived level of difficulty.

- There is currently no damping due to air. What is the proper way to implement this in articulated bodies?
  - I attempted to implement air damping several times throughout the development of the thesis but was never able to get it quite right, specifically, I found that the propagation of drag forces only led to chaotic 'jittery' motion as opposed to the desired damping motion.
- My method allows for any external forces to act on any number of rigid bodies of the plant, meaning it is easily extensible to represent motion due to vector fields (wind, plant immersed in water, etc.). However, the instability of the system (especially if acted upon by many external forces) is troubling. What to do here?
- The damping constant at the joints is currently set through trial and error. What is an automatic way to determine a good constant? Also, different organs probably require different nodal damping constants?
- Does a certain consideration need to be taken regarding spring constants at branching points?

  - In the current method, children bodies of the same parent are unaware of each other, meaning that no dynamical behaviour is captured regarding the geometry of plants at their branching points (Figure 10.1). This geometry provides extra rigidity at the branching points; how to capture it using spring constants?

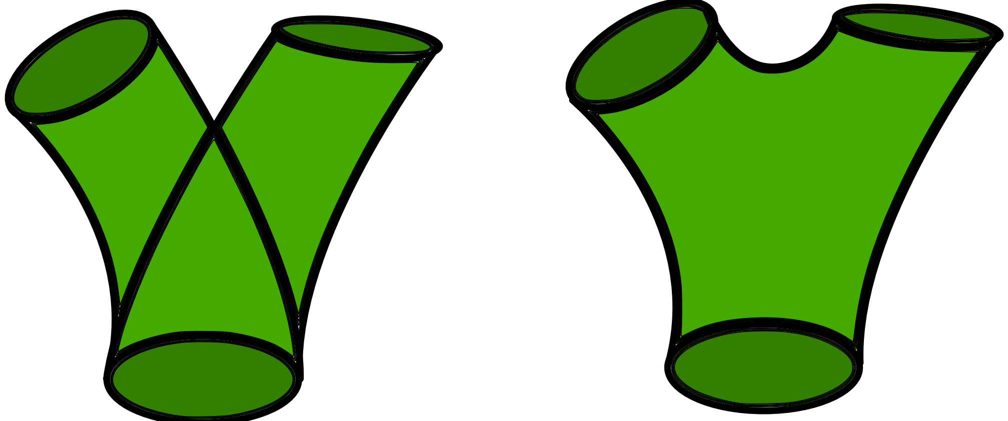


Figure 10.1: My methodology treats branching points as independent segments (left), whereas in reality, plants exhibit unique geometry at the branching points (right).

- What is the best spring constant to properly represent a continuous internode with discretized segments?
  - Keep in mind that branches/stems are inhomogeneous in both density and Young's modulus (in both lateral and axial directions); my method does not handle either of these.
  - Densities and moduli of elasticity also change over time and have different values in different parts of the plant. How to model this? For example, the *pipe model* may be used to calculate the plausible radii of child branches with respect to the radius of the parent branch. Does a similar rule exist for the Young's modulus?
- How to handle joints between different types of rigid bodies? For example, what would be the best spring constant between a stem and a leaf? Do we need to introduce petioles?

  - All organs are currently represented by cylindrical internodes in order to avoid this problem.

- The Young's moduli used in my simulations were all adapted from the observations provided in [56]. Is there a way to automatically generate these values for internodes? How about for leaves, flowers, and other organs?

- The shear modulus for a plant was obtained from its Young's modulus via an equation that assumes that the material is isotropic, which plants are not. How would one address this limitation?

10.2 Future work: efficiency

The second category of research problems is concerned with making the articulated-body plant simulations faster, such that we can interactively simulate more complex models, or interactively simulate many models in the same scene. Note that as per the discussion in Section 9.4.2, efficiency and stability are effectively intertwined.

- It is known that one can use quaternions to represent spherical joints as opposed to Euler angles. Would using quaternions affect stability? And if so, would it increase it or decrease it?

- Could we get significant speedups via multi-threading?

- Can we get significant speedups using different time integrating schemes? It is known that forward (or explicit) time-integration schemes are not appropriate for chaotic systems. How would other time-integration schemes handle these simulations?

  - For example, it is known that *backward differentiation formula* (BDF) methods excel at numerically integrating stiff mechanical systems. The BDF refers to a family of implicit methods that provides high stability when integrating chaotic

systems such as ours.

- What about using hardware-accelerated extension to Featherstone's algorithm?
  - This is non-trivial because modern hardware-accelerated methods, such as GPU-programming methods, excel on *decoupled* systems where the equations are all independent of each other. The matrix representing the equations of motion of a plant can exhibit *sparsity* depending on its branching structure, but ultimately, the system cannot be fully decoupled because the dynamics of a body will always depend on the bodies attached to it.

10.3 Future work: coupling of growth and dynamics

The final research question I will be mentioning is the one that speaks the most to me, and it regards the coupling of growth and dynamics.

Back in Section 2.3, we discussed a few works in the field of physically-based developmental plant modeling whose growth and dynamics models classified as coupled systems. What this means is that to some extent, these works implemented a mechanism by which long-term growth could affect dynamical behaviour, and dynamical behaviour could affect long-term growth. It is important to research these types of developmental models because capturing a plant's growth response to mechanical (Newtonian) forces is just as essential as capturing its growth response to other 'classes' of environmental influences such as light availability, gravitational effects, ambient temperature, or water intake to name a few. Actually, it is possible that a certain level of physical abstraction exists for which all of these growth influences can be captured by a single comprehensive physically-based growth model (e.g. a physically-based model operating at the microscopic scale), meaning that they could ulti-

mately be ‘equivalent’. Unfortunately however, we will likely not see such a computational model for plant growth anytime soon.

We are therefore interested in the following question:

> “How can we extend the methodology presented in this thesis such that growth and dynamics are coupled?”

The main drawback of the paradigm employed in this thesis can be explained through its state diagram as illustrated in Figure 10.2. In this methodology, an L-system model is grown via a desired number of derivation steps (growth), it is then exported to the physics model, whom may then subject the model to any number of physics steps (dynamics). The main issue with this method is that there is no information transfer from the physics model back to the L-system model, meaning that whatever happens in the dynamics calculations can never affect subsequent growth.

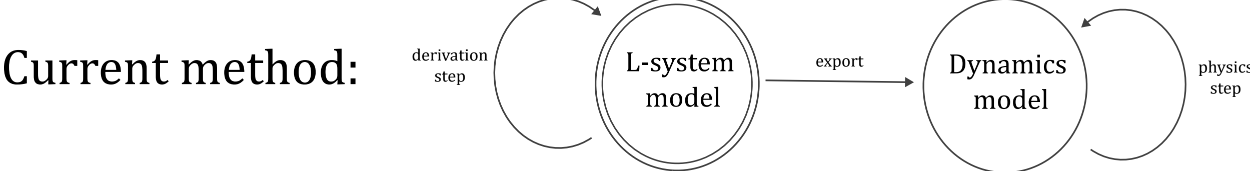


Figure 10.2: A state diagram illustrating the methodology employed in this thesis. Growth can influence dynamics, but dynamics cannot influence growth.

In order to address these limitations, we need to add information transfer from the dynamics model to the growth model (Figure 10.3). This extension would let the growth model know what’s going on in the dynamics world, and would let us capture certain currently unrepresentable behaviours such as plastic deformation, fracture, or even loss of branches. There is one remaining issue with this method, however, in that it requires two models to be defined: a growth model and a dynamics model, meaning that data is needlessly duplicated.

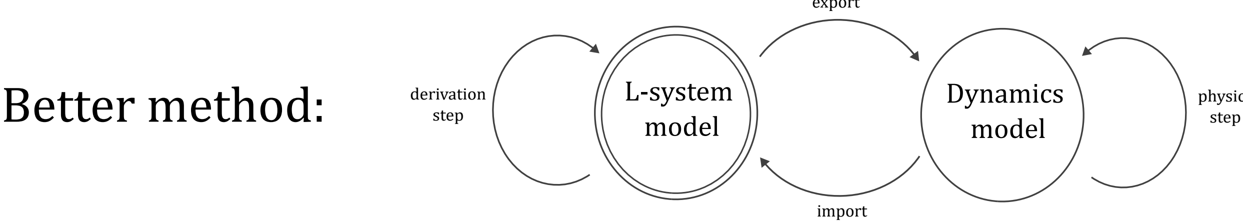


Figure 10.3: A more elegant solution would account for bi-directional flow between the growth model and dynamics model

This leads into the last solution - which I believe to be the most elegant - and that is to calculate the dynamics *within* the L-system programming language itself (Figure 10.4). The overarching idea here is that there are no fundamental differences between a growth model and a dynamics model; they are both representations of the *time evolution* of a plant, regardless of the time scale on which they operate. It just so happens that both growth and (Featherstone-style) dynamics are ideally representable by Lindenmayer systems.

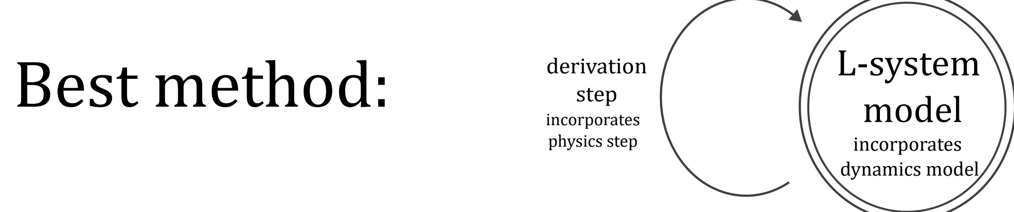


Figure 10.4: The most elegant solution would incorporate dynamics as a part of its growth model altogether.

Pursuing this methodology would necessitate the efficient implementation of Featherstone's algorithm within the L-system framework, which is not a simple task. Additionally, an elegant implementation of this methodology should presumably incorporate bi-directional communication between the L-systems and their environment. This would possibly require an extension to the concept of *Open L-systems* [52] to account for transferring abstract dynamical data between the plants and their environment.

Bibliography

[1] Harold Abelson and Andrea DiSessa. *Turtle geometry: The computer as a medium for exploring mathematics*. MIT press, 1986.

[2] Masaki Aono and Tosiyasu L. Kunii. Botanical tree image generation. *IEEE Computer Graphics and Applications*, 4(5):10–34, 1984.

[3] James Arvo, David Kirk, et al. Modeling plants with environment-sensitive automata. In *In Proceedings of Ausgraph'88*. Citeseer, 1988.

[4] David Baraff. Linear-time dynamics using lagrange multipliers. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 137–146, 1996.

[5] Jernej Barbič and Doug L James. Real-time subspace integration for st. venant-kirchhoff deformable models. *ACM transactions on graphics (TOG)*, 24(3):982–990, 2005.

[6] Jernej Barbič and Yili Zhao. Real-time large-deformation substructuring. *ACM transactions on graphics (TOG)*, 30(4):1–8, 2011.

[7] Jacob Beaudoin and John Keyser. Simulation levels of detail for plant motion. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 297–304, 2004.

[8] Ferdinand Pierre Beer, Elwood Russell Johnston, John T DeWolf, David Francis Mazurek, and Sanjeev Sanghi. *Mechanics of materials*, volume 1. mcgraw-Hill New York, 1992.

[9] Miklós Bergou, Max Wardetzky, Stephen Robinson, Basile Audoly, and Eitan Grinspun. Discrete elastic rods. In *ACM SIGGRAPH 2008 papers*, pages 1–12. 2008.

[10] Florence Bertails. Linear time super-helices. In *Computer graphics forum*, volume 28, pages 417–426. Wiley Online Library, 2009.

[11] Florence Bertails, Basile Audoly, Marie-Paule Cani, Bernard Querleux, Frédéric Leroy, and Jean-Luc Lévêque. Super-helices for predicting the dynamics of natural hair. *ACM Transactions on Graphics (TOG)*, 25(3):1180–1187, 2006.

[12] Blender Foundation. Blender, 2021. [Computer software].

[13] Jules Bloomenthal. Modeling the mighty maple. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, page 305–311, New York, NY, USA, 1985. Association for Computing Machinery.

[14] Jules Bloomenthal. Modeling the mighty maple. *SIGGRAPH Comput. Graph.*, 19(3):305–311, July 1985.

[15] Frederic Boudon, Przemyslaw Prusinkiewicz, Pavol Federl, Christophe Godin, and Radoslaw Karwowski. Interactive design of bonsai tree models. In *Computer Graphics Forum*, volume 22, pages 591–599. Wiley Online Library, 2003.

[16] Ayan Chaudhury and Christophe Godin. Geometry reconstruction of plants. In *Intelligent Image Analysis for Plant Phenotyping*, pages 119–142. CRC Press, 2020.

[17] Yung-Yu Chuang, Dan B Goldman, Ke Colin Zheng, Brian Curless, David H Salesin, and Richard Szeliski. Animating pictures with stochastic motion textures. In *ACM SIGGRAPH 2005 Papers*, pages 853–860. 2005.

[18] Tina LM Derzaph and Howard J Hamilton. Effects of wind on virtual plants in animation. *International Journal of Computer Games Technology*, 2013, 2013.

[19] Oliver Deussen and Bernd Lintermann. A modelling method and user interface for creating plants. In *Graphics interface*, volume 97, pages 189–198. Citeseer, 1997.

[20] Thomas Di Giacomo, Stéphane Capo, and François Faure. An interactive forest. In *Computer Animation and Simulation 2001*, pages 65–74. Springer, 2001.

[21] Roy Featherstone. The calculation of robot dynamics using articulated-body inertias. *The international journal of robotics research*, 2(1):13–30, 1983.

[22] Roy Featherstone. An empirical study of the joint space inertia matrix. *The International Journal of Robotics Research*, 23(9):859–871, 2004.

[23] Roy Featherstone. Plucker basis vectors. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 1892–1897. IEEE, 2006.

[24] Roy Featherstone. A beginner's guide to 6-d vectors (part 1). *IEEE robotics & automation magazine*, 17(3):83–94, 2010.

[25] Roy Featherstone. *Rigid body dynamics algorithms.* Springer, 2014.

[26] Roy Featherstone and David Orin. Robot dynamics: equations and algorithms. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 1, pages 826–834. IEEE, 2000.

[27] Pavol Federl and Przemyslaw Prusinkiewicz. Virtual laboratory: An interactive software environment for computer graphics. In *Computer graphics international*, volume 242, page 93x100. Citeseer, 1999.

[28] Martin L Felis. Rbdl: an efficient rigid-body dynamics library using recursive algorithms. *Autonomous Robots*, 41(2):495–511, 2017.

[29] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.

[30] Epic Games. Unreal engine 5. Computer software, 2021.

[31] Herbert Goldstein, Charles Poole, and John Safko. Classical mechanics, 2002.

[32] Barry J Goodno and James M Gere. *Mechanics of materials*. Cengage learning, 2020.

[33] Jianwei Guo, Shibiao Xu, Dong-Ming Yan, Zhanglin Cheng, Marc Jaeger, and Xiaopeng Zhang. Realistic procedural plant modeling from multiple view images. *IEEE transactions on visualization and computer graphics*, 26(2):1372–1384, 2018.

[34] Ralf Habel, Alexander Kusternig, and Michael Wimmer. Physically guided animation of trees. In *Computer Graphics Forum*, volume 28, pages 523–532. Wiley Online Library, 2009.

[35] Torsten Hädrich, Bedrich Benes, Oliver Deussen, and Sören Pirk. Interactive modeling and authoring of climbing plants. In *Computer Graphics Forum*, volume 36, pages 49–61. Wiley Online Library, 2017.

[36] Zygmunt Hejnowicz and Andreas Sievers. Tissue stresses in organs of herbaceous plants: I. poisson ratios of tissues and their role in determination of the stresses. *Journal of Experimental Botany*, pages 1035–1043, 1995.

[37] Hisao Honda. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology*, 31(2):331–338, 1971.

[38] Donald House and John C Keyser. *Foundations of Physically Based Modeling and Animation*. AK Peters/CRC Press, 2016.

[39] Doug L James, Christopher D Twigg, Andrew Cove, and Robert Y Wang. Mesh ensemble motion graphs: Data-driven mesh animation with constraints. *ACM Transactions on Graphics (TOG)*, 26(4):17–es, 2007.

[40] Catherine Jirasek, Przemyslaw Prusinkiewicz, and Bruno Moulia. Integrating biomechanics into developmental plant models expressed using l-systems. *Plant biomechanics*, pages 615–624, 2000.

[41] Radoslaw Karwowski and Przemyslaw Prusinkiewicz. The l-system-based plant-modeling environment l-studio 4.0. In *Proceedings of the 4th international workshop on functional-structural plant models*, pages 403–405. UMR AMAP Montpellier, France, 2004.

[42] Evangelos Kokkevis. Practical physics for articulated characters. In *Game Developers Conference*, volume 2004. Citeseer, 2004.

[43] Lucas Kovar, Michael Gleicher, and Frederic Pighin. Motion graphs, 2002.

[44] Tassilo Kugelstadt and Elmar Schömer. Position and orientation based cosserat rods. In *Symposium on Computer Animation*, pages 169–178, 2016.

[45] Jehee Lee, Jinxiang Chai, Paul SA Reitsma, Jessica K Hodgins, and Nancy S Pollard. Interactive control of avatars animated with human motion data. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 491–500, 2002.

[46] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

[47] Aristid Lindenmayer. Developmental systems without cellular interactions, their languages and grammars. *Journal of Theoretical Biology*, 30(3):455–484, 1971.

[48] Bernd Lintermann and Oliver Deussen. Interactive modeling of plants. *IEEE Computer Graphics and Applications*, 19(1):56–65, 1999.

[49] Steven Longay, Adam Runions, Frédéric Boudon, and Przemyslaw Prusinkiewicz. Treesketch: Interactive procedural modeling of trees on a tablet. In *SBIM@ Expressive*, pages 107–120. Citeseer, 2012.

[50] Miłosz Makowski, Torsten Hädrich, Jan Scheffczyk, Dominik L Michels, Sören Pirk, and Wojtek Pałubicki. Synthetic silviculture: multi-scale modeling of plant ecosystems. *ACM Transactions on Graphics (TOG)*, 38(4):1–14, 2019.

[51] Benoit B Mandelbrot. Fractal geometry: what is it, and what does it do? *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 423(1864):3–16, 1989.

[52] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410, 1996.

[53] Brian Vincent Mirtich. *Impulse-based dynamic simulation of rigid body systems*. University of California, Berkeley, 1996.

[54] Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Meshless deformations based on shape matching. *ACM transactions on graphics (TOG)*, 24(3):471–478, 2005.

[55] Karl J Niklas. *Plant biomechanics: an engineering approach to plant form and function*. University of Chicago press, 1992.

[56] Karl J Niklas. Plant height and the properties of some herbaceous stems. *Annals of Botany*, 75(2):133–142, 1995.

[57] Karl J Niklas and Hanns-Christof Spatz. *Plant physics*. University of Chicago Press, 2012.

[58] MJ O'Dogherty. A review of the mechanical behaviour of straw when compressed to high densities. *Journal of Agricultural Engineering Research*, 44:241–265, 1989.

[59] Peter E. Oppenheimer. Real time design and animation of fractal plants and trees. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, page 55–64, New York, NY, USA, 1986. Association for Computing Machinery.

[60] Shin Ota, Machiko Tamura, Tadahiro Fujimoto, Kazunobu Muraoka, and Norishige Chiba. A hybrid method for real-time animation of trees swaying in wind fields. *The Visual Computer*, 20(10):613–623, 2004.

[61] Dinesh K Pai. Strands: Interactive simulation of thin solids using cosserat models. In *Computer graphics forum*, volume 21, pages 347–352. Wiley Online Library, 2002.

[62] Wojciech Palubicki, Kipp Horel, Steven Longay, Adam Runions, Brendan Lane, Radomír Měch, and Przemyslaw Prusinkiewicz. Self-organizing tree models for image synthesis. *ACM Transactions On Graphics (TOG)*, 28(3):1–10, 2009.

[63] Sören Pirk, Michał Jarzabek, Torsten Hädrich, Dominik L Michels, and Wojciech Palubicki. Interactive wood combustion for botanical tree models. *ACM Transactions on Graphics (TOG)*, 36(6):1–12, 2017.

[64] Sören Pirk, Till Niese, Torsten Hädrich, Bedrich Benes, and Oliver Deussen. Windy trees: computing stress response for developmental tree models. *ACM Transactions on Graphics (TOG)*, 33(6):1–11, 2014.

[65] Joanna L Power, AJ Bernheim Brush, Przemyslaw Prusinkiewicz, and David H Salesin.

Interactive arrangement of botanical l-system models. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 175–182, 1999.

[66] Przemyslaw Prusinkiewicz. Applications of l-systems to computer imagery. In *International Workshop on Graph Grammars and Their Application to Computer Science*, pages 534–548. Springer, 1986.

[67] Przemyslaw Prusinkiewicz. Graphical applications of l-systems. In *Proceedings of graphics interface*, volume 86, pages 247–253, 1986.

[68] Przemyslaw Prusinkiewicz, Yvette Erasmus, Brendan Lane, Lawrence D Harder, and Enrico Coen. Evolution and development of inflorescence architectures. *Science*, 316(5830):1452–1456, 2007.

[69] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

[70] Przemyslaw Prusinkiewicz, Lars Mündermann, Radoslaw Karwowski, and Brendan Lane. The use of positional information in the modeling of plants. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 289–300, New York, NY, USA, 2001. Association for Computing Machinery.

[71] Przemyslaw Prusinkiewicz, Lars Mündermann, Radoslaw Karwowski, and Brendan Lane. The use of positional information in the modeling of plants. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 289–300, 2001.

[72] Ed Quigley, Yue Yu, Jingwei Huang, Winnie Lin, and Ronald Fedkiw. Real-time interactive tree animation. *IEEE transactions on visualization and computer graphics*, 24(5):1717–1727, 2017.

[73] Alex Reche-Martinez, Ignacio Martin, and George Drettakis. Volumetric reconstruction and interactive rendering of trees from photographs. In *ACM SIGGRAPH 2004 Papers*, pages 720–727. 2004.

[74] William T Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Transactions On Graphics (TOG)*, 2(2):91–108, 1983.

[75] William T Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *ACM siggraph computer graphics*, 19(3):313–322, 1985.

[76] Adam Runions, Martin Fuhrer, Brendan Lane, Pavol Federl, Anne-Gaëlle Rolland-Lagan, and Przemyslaw Prusinkiewicz. Modeling and visualization of leaf venation patterns. In *ACM SIGGRAPH 2005 Papers*, pages 702–711. 2005.

[77] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling trees with a space colonization algorithm. *NPH*, 7:63–70, 2007.

[78] Tatsumi Sakaguchi. Botanical tree structure modeling based on real image set. In *ACM SIGGRAPH 98 Conference abstracts and applications*, page 272, 1998.

[79] Tatsumi Sakaguchi and Jun Ohya. Modeling and animation of botanical trees for interactive virtual environments. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 139–146, 1999.

[80] Arno Schödl, Richard Szeliski, David H Salesin, and Irfan Essa. Video textures. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 489–498, 2000.

[81] Michael A Sherman, Ajay Seth, and Scott L Delp. Simbody: multibody dynamics for biomedical research. *Procedia Iutam*, 2:241–261, 2011.

[82] Troy Shinbrot, Celso Grebogi, Jack Wisdom, and James A Yorke. Chaos in a double pendulum. *American Journal of Physics*, 60(6):491–499, 1992.

[83] Mikio Shinya and Alain Fournier. Stochastic motion—motion under the influence of wind. In *Computer graphics forum*, volume 11, pages 119–128. Wiley Online Library, 1992.

[84] Ilya Shlyakhter, Max Rozenoer, Julie Dorsey, and Seth Teller. Reconstructing 3d tree models from instrumented photographs. *IEEE Computer Graphics and Applications*, 21(3):53–61, 2001.

[85] Jos Stam. Stochastic dynamics: Simulating the effects of turbulence on flexible structures. In *Computer Graphics Forum*, volume 16, pages C159–C164. Wiley Online Library, 1997.

[86] Ondrej Stava, Sören Pirk, Julian Kratt, Baoquan Chen, Radomír Měch, Oliver Deussen, and Bedrich Benes. Inverse procedural modelling of trees. In *Computer Graphics Forum*, volume 33, pages 118–131. Wiley Online Library, 2014.

[87] Meng Sun, Allan D Jepson, and Eugene Fiume. Video input driven animation (vida). In *Computer Vision, IEEE International Conference on*, volume 2, pages 96–96. IEEE Computer Society, 2003.

[88] Ping Tan, Tian Fang, Jianxiong Xiao, Peng Zhao, and Long Quan. Single image tree modeling. *ACM Transactions on Graphics (TOG)*, 27(5):1–7, 2008.

[89] Ping Tan, Gang Zeng, Jingdong Wang, Sing Bing Kang, and Long Quan. Image-based tree modeling. In *ACM SIGGRAPH 2007 papers*, pages 87–es. 2007.

[90] Christopher D Twigg and Zoran Kacic-Alesic. Point cloud glue: Constraining simulations using the procrustes transform. In *Symposium on Computer Animation*, pages 45–53, 2010.

[91] Stanislaw Ulam et al. On some mathematical problems connected with patterns of growth of figures. In *Proceedings of Symposia in Applied Mathematics*, volume 14, pages 215–224. Am. Math. Soc. Vol. 14, Providence, 1962.

[92] William Van Haevre, Fabian Di Fiore, and Frank Van Reeth. Physically-based driven tree animations. In *NPH*, pages 75–82, 2006.

[93] Bohan Wang, Yili Zhao, and Jernej Barbič. Botanical materials based on biomechanics. *ACM Transactions on Graphics (TOG)*, 36(4):1–13, 2017.

[94] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 119–128, 1995.

[95] Jason P Weber. Fast simulation of realistic trees. *IEEE Computer Graphics and Applications*, 28(3):67–75, 2008.

[96] Jamie Wither, Frédéric Boudon, M-P Cani, and Christophe Godin. Structure from silhouettes: a new paradigm for fast sketch-based design of trees. In *Computer Graphics Forum*, volume 28, pages 541–550. Wiley Online Library, 2009.

[97] Dong-Ming Yan, Julien Wintz, Bernard Mourrain, Wenping Wang, Frédéric Boudon, and Christophe Godin. Efficient and robust reconstruction of botanical branching structure from laser scanned points. In *2009 11th IEEE International Conference on Computer-Aided Design and Computer Graphics*, pages 572–575. IEEE, 2009.

[98] Long Zhang, Chengfang Song, Qifeng Tan, Wei Chen, and Qunsheng Peng. Quasi-physical simulation of large-scale dynamic forest scenes. In *Computer Graphics International Conference*, pages 735–742. Springer, 2006.

[99] Long Zhang, Yubo Zhang, Zhongding Jiang, Luying Li, Wei Chen, and Qunsheng

Peng. Precomputing data-driven tree animation. *Computer Animation and Virtual Worlds*, 18(4-5):371–382, 2007.

[100] Xiaopeng Zhang, Hongjun Li, Mingrui Dai, Wei Ma, and Long Quan. Data-driven synthetic modeling of trees. *IEEE transactions on visualization and computer graphics*, 20(9):1214–1226, 2014.

[101] Yili Zhao and Jernej Barbič. Interactive authoring of simulation-ready plants. *ACM Transactions on Graphics (TOG)*, 32(4):1–12, 2013.

[102] Olgierd Cecil Zienkiewicz, Robert Leroy Taylor, Perumal Nithiarasu, and JZ Zhu. *The finite element method*, volume 3. McGraw-hill London, 1977.

Appendix A

Equations of motion for the double physical pendulum

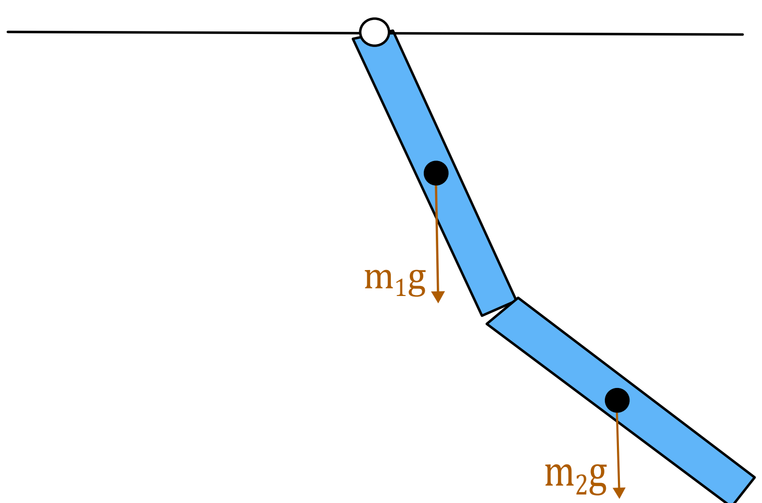


Figure A.1: A 2D double physical pendulum

The *double physical pendulum* consists of a link of two rigid bodies free to oscillate about a 2D plane (figure A.2). The first body is attached to the ceiling via a revolute joint, and the second body is attached to the first body also via a revolute joint. The equations of motion of this system can be represented by two functions f_1 and f_2, and they will both depend on all the instantaneous positions and velocities of the system:

$$\ddot{\theta}_1 = f_1(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2)$$
$$\ddot{\theta}_2 = f_2(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) \tag{A.1}$$

.

Equations of motion

We shall use the Euler-Lagrange equations in order to find f_1 and f_2 and solve for $\ddot{\theta}_1$ and $\ddot{\theta}_2$:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}_i}\right) = \frac{\partial \mathcal{L}}{\partial q_i}, \tag{A.2}$$

where the q_i are the generalized coordinate of the system (in our case, θ_1 and θ_2), and $\mathcal{L}$ is the Lagrangian of the system. The Lagrangian is equal to the total kinetic energy T minus the total potential energy V of the system,

$$\mathcal{L} = T - V. \tag{A.3}$$

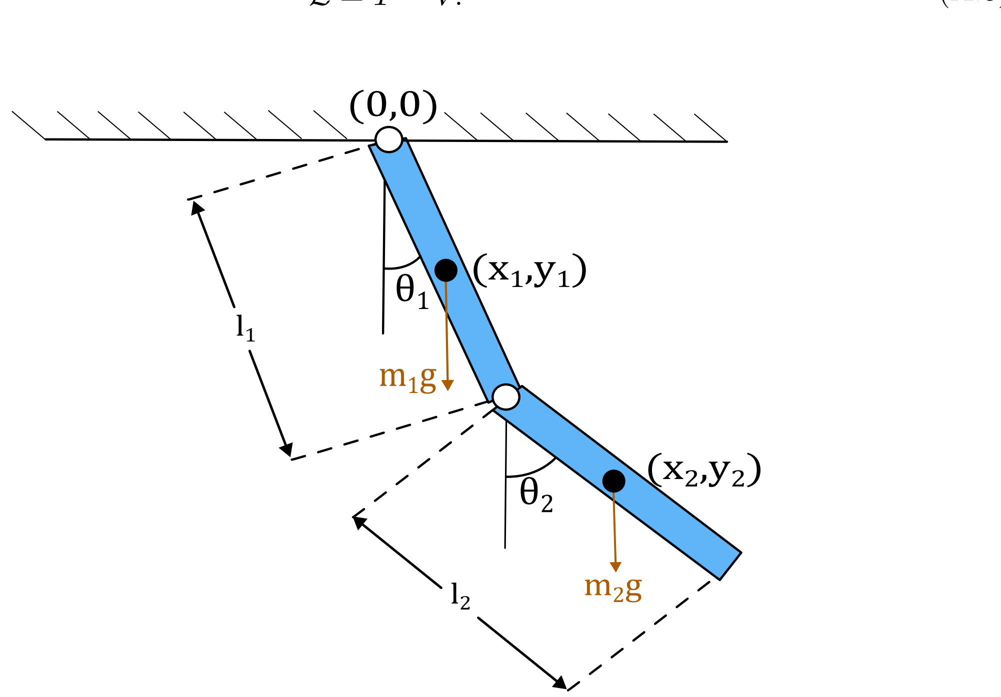


Figure A.2: Free body diagram of a double physical pendulum

We proceed by defining the Cartesian coordinate system of our mechanical system as shown

in figure A.2. Note that we define θ_1 and θ_2 to be the angle from the vertical, however, in our Featherstone's solver, θ_i is the angular displacement from our parent's heading vector. The reason for this is only to make the derivation of the Lagrangian a bit cleaner on paper.

Next, we need to find T and V. T is the total kinetic energy of the system and consists of the linear and angular kinetic energies of both bodies,

$$T = \frac{1}{2}m_1(\dot{x}_1^2 + \dot{y}_1^2) + \frac{1}{2}m_2(\dot{x}_2^2 + \dot{y}_2^2) + \frac{1}{2}I_1\dot{\theta}_1^2 + \frac{1}{2}I_2\dot{\theta}_2^2, \tag{A.4}$$

whereas the potential energy V contains the gravitational potentials of each body,

$$V = m_1 g y_1 + m_2 g y_2. \tag{A.5}$$

We proceed with a change of variable from the x_i's to θ_i's:

$$\begin{aligned} x_1 &= \frac{l_1}{2}\sin\theta_1, \\ y_1 &= -\frac{l_1}{2}\cos\theta_1, \\ x_2 &= l_1\sin\theta_1 + \frac{l_2}{2}\sin\theta_2, \\ y_2 &= -l_1 cos\theta_1 - \frac{l_2}{2}\cos\theta_2. \end{aligned} \tag{A.6}$$

Now we take the time derivative to get the $\dot{x}_i$'s in terms of the θ_i's and $\dot{\theta}_i$'s:

$$\begin{aligned} \dot{x}_1 &= \frac{l_1}{2}\dot{\theta}_1\cos\theta_1, \\ \dot{y}_1 &= \frac{l_1}{2}\dot{\theta}_1\sin\theta_1, \\ \dot{x_2} &= l_1\dot{\theta}_1\cos\theta_1 + \frac{l_2}{2}\dot{\theta}_2\cos\theta_2, \\ \dot{y_2} &= l_1\dot{\theta}_1\sin\theta_1 + \frac{l_2}{2}\dot{\theta}_2\sin\theta_2. \end{aligned} \tag{A.7}$$

We have successfully changed all variables from x_i's to θ_i's. Let's substitute them back into T:

$$
\begin{aligned}
T &= \frac{1}{2}m_1(\dot{x}_1^2 + \dot{y}_1^2) + \frac{1}{2}m_2(\dot{x}_2^2 + \dot{y}_2^2) + \frac{1}{2}I_1\dot{\theta}_1^2 + \frac{1}{2}I_2\dot{\theta}_2^2 \\
&= \frac{1}{2}m_1[\frac{1}{4}l_1^2\dot{\theta}_1^2(\cos^2\theta_1 + \sin^2\theta_1)] \\
&+ \frac{1}{2}m_2[l_1^2\dot{\theta}_1^2(\cos^2\theta_1 + \sin^2\theta_1) + \frac{1}{4}l_2^2\dot{\theta}_2^2(\cos^2\theta_1 + \sin^2\theta_1) + l_1 l_2\dot{\theta}_1\dot{\theta}_2(\cos\theta_1\cos\theta_2 + \sin\theta_1\sin\theta_2)] \\
&+ \frac{1}{2}J_1\dot{\theta}_1^2 + \frac{1}{2}J_2\dot{\theta}_2^2 \\
&= \frac{1}{2}(\frac{1}{4}m_1 l_1^2 + \frac{1}{12}m_1 l_1^2 + m_2 l_1^2)\dot{\theta}_1^2 + \frac{1}{2}(\frac{1}{4}m_2 l_2^2 + \frac{1}{12}m_2 l_2^2 + m_2 l_2^2)\dot{\theta}_2^2 + \frac{1}{2}m_2 l_1 l_2\dot{\theta}_1\dot{\theta}_2\cos(\theta_1 - \theta_2)
\end{aligned}
\tag{A.8}
$$

Note that the three quantities in blue are constants, so we will give them the following variable names:

$$
\begin{aligned}
J_a &= \frac{1}{3}m_1 l_1^2 + m_2 l_1^2, \\
J_b &= \frac{1}{3}m_2 l_2^2, \text{ and} \\
J_x &= \frac{1}{2}m_2 l_1 l_2.
\end{aligned}
\tag{A.9}
$$

J_a is the moment of inertia of the first body about the joint glued to the ceiling, with an extra term due to the inertia of the body attached to it. J_b is the moment of inertia of the second body about its own proximal joint, and J_x is a cross moment of inertia term that takes into account the rotational motion of the bodies with respect to each other. The final form for T is thus

$$
T = \frac{1}{2}J_a\dot{\theta}_1^2 + \frac{1}{2}J_b\dot{\theta}_2^2 + J_x\dot{\theta}_1\dot{\theta}_2\cos(\theta_1 - \theta_2). \tag{A.10}
$$

We proceed by also applying the change of variables to V:

$$
\begin{aligned}
V &= m_1 g y_1 + m_2 g y_2 \\
&= -\frac{1}{2}m_1 g l_1\cos\theta_1 - m_2 g(l_1\cos\theta_1 + \frac{1}{2}\cos\theta_2) \\
&= -(\frac{1}{2}m_1 + m_2)g l_1\cos\theta_1 - \frac{1}{2}m_2 g l_2\cos\theta_2.
\end{aligned}
\tag{A.11}
$$

The terms highlighted in blue are again constants, and we can name them μ_1 and μ_2 such that we're left with the following simplification:

$$\begin{aligned} V &= -\mu_1 \cos\theta_1 - \mu_2 \cos\theta_2, \text{ where} \\ \mu_1 &= (\frac{1}{2}m_1 + m_2)gl_1 \text{ and} \\ \mu_2 &= \frac{1}{2}m_2 g l_2. \end{aligned} \tag{A.12}$$

Our final equations for T (equation A.10) and V (equation A.12) may now be substituted into the Lagrangian:

$$\begin{aligned} \mathcal{L} &= T - V \\ &= \frac{1}{2}J_a\dot{\theta}_1^2 + \frac{1}{2}J_b\dot{\theta}_2^2 + J_x\dot{\theta}_1\dot{\theta}_2\cos(\theta_1 - \theta_2) + \mu_1\cos\theta_1 + \mu_2\cos\theta_2. \end{aligned} \tag{A.13}$$

The Lagrangian has now been re-stated in terms of its generalized coordinates, θ_1 and θ_2, which means that we may proceed with the Euler-Lagrange equations.

$$\frac{d}{dt}\left(\frac{\partial\mathcal{L}}{\partial\dot{\theta}_i}\right) - \frac{\partial\mathcal{L}}{\partial q_\theta} = 0. \tag{A.14}$$

Plugging in θ_1 into the above equation, we get the following equation of motion after simplification:

$$J_a\ddot{\theta}_1 + J_x\cos(\theta_1 - \theta_2)\ddot{\theta}_2 + J_x\sin(\theta_1 - \theta_2)\dot{\theta}_2^2 + \mu_1\sin\theta_1 = 0, \tag{A.15}$$

and similarly for θ_2:

$$J_b\ddot{\theta}_2 + J_x\cos(\theta_1 - \theta_2)\ddot{\theta}_1 + J_x\sin(\theta_1 - \theta_2)\dot{\theta}_1^2 + \mu_2\sin\theta_2 = 0. \tag{A.16}$$

We now isolate for $\ddot{\theta}_1$ and $\ddot{\theta}_2$:

$$
\begin{aligned}
\ddot{\theta}_1 &= \frac{J_x \cos(\theta_1 - \theta_2)\ddot{\theta}_2 + J_x \sin(\theta_1 - \theta_2)\dot{\theta}_2^2 + \mu_1 \sin\theta_1}{-J_a}, \\
\ddot{\theta}_2 &= \frac{J_x \cos(\theta_1 - \theta_2)\ddot{\theta}_1 + J_x \sin(\theta_1 - \theta_2)\dot{\theta}_1^2 + \mu_2 \sin\theta_2}{-J_b}.
\end{aligned}
\tag{A.17}
$$

Both $\ddot{\theta}_1$ and $\ddot{\theta}_1$ still appear in both equations, but it can be solved by substituting one equation into the other, and isolating for the respective $\ddot{\theta}_i$ term, yielding the desired equations.