Designing a Class Library for Interactive Simulation of Rigid Body Dynamics

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Barenbrug, Bart G.B.

Designing a class library for interactive simulation of rigid body dynamics / by Bart G.B. Barenbrug. - Eindhoven : Eindhoven University of Technology, 2000.
Proefschrift. - ISBN 90-386-0751-2
NUGI 855
Subject headings: animation techniques / computer simulation ; physics / software libraries / object oriented programming / computational geometry / numerical algorithms
CR Subject Classification (1998): I.3.5, D.2.2, D.1.5, I.3.7, I.6.8, F.2.2, G.1.0

Printed by University Press Facilities, Eindhoven
Cover design by Ben Mobach
Cover illustrations by Bart Barenbrug

Stan Ackermans Instituut, Centrum voor Technologisch Ontwerpen

© April 2000, Eindhoven University of Technology

Niets van deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt door middel van druk, fotokopie, microfilm of op welke andere wijze dan ook zonder voorafgaande schriftelijke toestemming van de auteur.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission from the copyright owner.

ISBN 90-3860-751-2

Designing a Class Library for Interactive Simulation of Rigid Body Dynamics

PROEFONTWERP

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op donderdag 27 april 2000 om 16.00 uur

door

Bart Gerard Bernard Barenbrug

geboren te Oud-Beijerland

Dit proefontwerp is goedgekeurd door de promotoren:

prof.dr.dipl.ing. D.K. Hammer
en
prof.dr. R.M.M. Mattheij

Copromotor:
dr.ir. C.W.A.M. van Overveld

Acknowledgments

Almost six years ago, I was looking for a subject in the field of computer graphics for a graduation project for my master's degree. Kees van Overveld suggested an assignment in the field of *dynamics* in computer animation, a subject that needed some explanation by Kees, because I had never heard of it. Little did I know that some six years later I would be writing a Ph.D. thesis on the topic. I've learned a lot in those six years, and for their help, support and insight during the time of the project that led to this thesis, I owe thanks to many people.

The work described in this thesis got started as a final project for the OOTI course. During that time, I had support from my fellow (now ex-)OOTIs and Corry and Marloes, as well as the members of the Technische Toepassingen group. The support of the latter only increased in the last two years when I was working in that group as an AIO. I had a very nice time both as an OOTI and as a TTer.

I also had a very pleasant time with the Madymo crash validation team at TNO, where Paul van Bemmelen, Ronald Schouten and Erwin Poeze and the rest of the team made me feel very welcome.

At the university, I owe thanks to Richard and Isabelle for TAO, along with the AIOs and programmers who were part of it. Gratitude also to Marja for keeping TT as vibrant a working environment as it was, to Tineke for her help during the final period of finishing this work, to Maarten for always being gamely, and to Alex, Paul and Elsa for the wonderful holidays.

But next to holidays, there was also the pleasure of work, which brings me to all the people who colaborated on the project itself: Arnold Reusken and Mischa Courtin who gave their input on some of the numerical problems. Huub and Elisabeth for their help with the intricacies of the GDP. Wim for helping reinvent the wheel, and Kees Huizing for his insights on the topic of component based software engineering. Gino for the discussions about interfacing collision detection and collision handling ideas, and Alex Telea for the insight in the general workings of simulation environments. Phap Nguyen and Cristian Pau for their work on the visualization of forces and looking into Java's EAI respectively. The members of the reading commitee for providing the comments that made this thesis that which it is. Most importantly, a big thank you to Kees van Overveld for all the slight nudges and big shoves (as needed) in the right direction, for the comments and suggestions for improving this thesis and the work which it describes, for making the project possible in the first place, and finally for introducing me to the wonderful topic of dynamics in computer animation six years ago.

Finally, more appreciation than can be put into words to my love Andrea for her support ever since we got together, and to my family, my parents Riet and Eric in particular, for all their support over a period of a lot longer than six years in the past, and hopefully many many more years in the future.

Contents

1 Introduction **1**

1.1 Previous work 2

1.1.1 Dynamics at the TUE 5

1.2 Organization 5

2 The environment and requirements **6**

2.1 Simulation/Animation environments 6

2.2 The assumptions about the host system 7

2.3 The requirements of Dynamo 8

3 The design **11**

3.1 Methodology 11

3.1.1 Views and notation 11

3.1.2 Software design phases 11

3.1.3 Object orientation 12

3.2 Forward dynamics 13

3.2.1 Physics 13

3.2.2 The forward dynamics classes 16

3.2.3 Integration of the motion equations 19

3.2.4 Forward dynamics classes revisited 23

3.2.5 Summary 27

3.3 Controllers 28

3.4 Inverse dynamics 32

3.4.1 Introduction 32

3.4.2 The interfaces of the constraints and the constraint manager 34

3.4.3 The inverse dynamics algorithm: overview 37

3.4.4 Algorithm details and incorporation in the classes 39

3.4.5 Determining $\dfrac{\partial \mathbf{C}}{\partial \mathbf{R}}$ 42
3.4.6 Solving ΔR from $-C = \frac{\partial C}{\partial R}\Delta R$ 50
3.4.7 Positional constraints issues 54
3.5 Additional features 56
3.5.1 Visualizing forces 56
3.5.2 Friction 58

4 Specializations of the constraints and the controllers **61**
4.1 Specific constraints 61
4.1.1 The empty constraint 61
4.1.2 The point-to-point constraint 62
4.1.3 The point-to-curve constraint 63
4.1.4 The point-to-surface constraint 66
4.1.5 The line-hinge constraint 67
4.1.6 The cylinder constraint 69
4.1.7 The connector constraint 70
4.1.8 The orientation constraint 71
4.1.9 The prism constraint 72
4.1.10 The plane constraint 73
4.1.11 The bar constraint 74
4.1.12 The multibar constraint 75
4.1.13 The wheel constraint 75
4.1.14 The collision constraint 79
4.2 Specific controllers 84
4.2.1 The spring and damper controller 84
4.2.2 The PID controller 84

5 Examples **87**
5.1 A simple example 87
5.2 The bicycle 89
5.3 The coupled pendula 91

5.4 The door spring 93
5.5 The roller coaster 94

6 The design process **101**
6.1 Introduction 101
6.2 Design phases 101
  6.2.1 Interface design 102
  6.2.2 Algorithm design 103
  6.2.3 Code design 104
6.3 Planning 104

7 Conclusions and future work **106**
7.1 Conclusions 106
7.2 Future work 107
  7.2.1 Extensions 107
  7.2.2 Internal changes 108

A Dyna's partial derivatives **109**
A.1 Introduction 109
A.2 Partial derivatives for central forces 110
  A.2.1 Determining $\frac{\partial p_{t+h}}{\partial F_t}$ analytically 110
  A.2.2 Determining $\frac{\partial \dot{p}_{t+h}}{\partial F_t}$ analytically 110
A.3 Partial derivatives for torques 111
  A.3.1 Determining $\frac{\partial p_{t+h}}{\partial M_t}$ analytically 111
  A.3.2 Determining $\frac{\partial \dot{p}_{t+h}}{\partial M_t}$ analytically 112
  A.3.3 Determining $\frac{\partial d_{t+h}}{\partial M_t}$ analytically 113
  A.3.4 Determining $\frac{\partial \dot{d}_{t+h}}{\partial M_t}$ analytically 114
A.4 Partial derivatives for (non-central) forces 114
  A.4.1 Determining $\frac{\partial p_{t+h}}{\partial f_t^{p'}}$ analytically 114
  A.4.2 Determining $\frac{\partial \dot{p}_{t+h}}{\partial f_t^{p'}}$ analytically 114

A.4.3 Determining $\frac{\partial d_{t+h}}{\partial f_t^{p'}}$ analytically 114

A.4.4 Determining $\frac{\partial \dot{d}_{t+h}}{\partial f_t^{p'}}$ analytically 115

A.5 Partial derivatives for impulses 115

A.5.1 Determining $\frac{\partial \dot{p}_{t+}}{\partial \iota_t}$ analytically 115

A.5.2 Determining $\frac{\partial \dot{d}_{t+}}{\partial \iota_t}$ analytically 116

A.5.3 Determining $\frac{\partial p_{t+h}}{\partial \iota_t}$ analytically 116

A.5.4 Determining $\frac{\partial d_{t+h}}{\partial \iota_t}$ analytically 118

A.5.5 Determining $\frac{\partial \dot{p}_{t+h}}{\partial \iota_t}$ analytically 118

A.5.6 Determining $\frac{\partial \dot{d}_{t+h}}{\partial \iota_t}$ analytically 119

B Discrete time motion integration **120**

C Velocity terms **124**

D The tensor of inertia and one dimensional dynas **126**

E List of symbols **127**

F OMT notation **128**

G Dynamo classes overview **129**

H Constraint satisfaction **130**

H.1 The constraint satisfaction problem 130

H.2 Domain specific issues 130

H.3 Iterative constraint correction 130

H.4 Constraint correction considerations 131

I Source of an example **132**

References **134**

Index **138**

Summary 140

Samenvatting 141

Curriculum Vitae 143

1 Introduction

The success of motion pictures such as *Jurassic Park*, *The Phantom Menace* and *A Bug's Life* show that computer animation is becoming more popular than ever. While the big movie studios such as *ILM*, *Disney* and *Pixar* can afford a lot of time, money and effort to get every motion right in these movies, in general it is very hard to make convincingly looking animation, especially animation supporting interactivity. Interactive animation is also becoming more and more commonplace in areas such as computer games, simulation and virtual reality applications.

One of the reasons that it is hard to generate convincingly looking animations is the suitability of the techniques that are often used to specify and calculate the motions of the moving objects in computer animations. Commonly used techniques are *keyframing* and *kinematics*. Using keyframing, an animator has to specify the exact positions and orientations of some of the objects he/she wants to animate for several moments in time, and the computer is used to calculate the positions and orientations of the objects in the in-between moments. Using kinematics, an animator directly specifies rotations and translations of the objects (possibly relative to each other) to specify their motions. Both these techniques require a skilled artist to make the motions look natural, but require relatively little computational effort.

Dynamics

With the continuously growing power of computers, the technique of using *dynamics* calculations has become feasible for application in interactive animation. So-called *forward dynamics* entails programming the animation program in a way that it can simulate the physics (as far as motions are concerned) that we see around us every day, thereby making the motions looking natural. In other words, dynamics calculations result in motions by calculating the effects of inertia and of forces.

Furthermore, it is possible to not only calculate the effects of forces, but also to derive many of the forces that occur in a given system, based on some notion of what their effect should be (*inverse dynamics*). Using so-called *constraints*, one can easily specify for example joints between moving objects (pin-joints, ball-joints and such), causing them to form more complex articulated bodies. Such joints partially limit the relative motions of the objects that are connected, and as a result the motions of the whole are far more complex, yet can still be calculated automatically via the occuring forces.

Software structure

To use the more complex dynamics algorithms within a software structure, attention has to be given to the way the algorithms are incorporated into the software. Of main importance is the manner in which the different parts of the software interface with each other. It should be possible to use and steer the dynamics subsystem without being bothered by many of the details of the underlying physics and mathematics in a manner which comes as natural as the resulting motions will look.

The main goal of the work presented in this thesis is to provide a software structure for performing and steering the dynamics calculations. Emphasis is not only on the calculations themselves, but also on how they are distributed over the components of a well structured piece of software. This software is made available in the form of a class library called *Dynamo* (the DYNAmics MOtion library, see [Baren 99]).

Software design

Determining a good software structure and interface is the main design challenge in the creation of Dynamo. Making the complex dynamics algorithms accessible, and harnessing the approximately 20.000 lines of source code of Dynamo in a structure that is understandable and extendible takes a lot of extra effort next to the algorithm design which would be the focus of a regular Ph.D. thesis. This is why this thesis has taken the form of a designer's Ph.D. thesis.

Beneficiaries

The beneficiaries of this work are first of all the users of Dynamo. Dynamo can be used in the field of computer animation, but because of the simulation approach taken, the application area of the software described in this thesis is widened from just application in the field of computer animation: since the motions are (close to) physically correct, it can also be used to analyse mechanical systems. For a more detailed description of the profile of a user of Dynamo, see Section 2.1.

Second of all, people interested in software design can benefit from this work as an example of how a relatively large system such as Dynamo can be developed in a way that uses the tools provided by object orientation to keep its structure clear, and its interface understandable. Part of this work has been sponsored by the TNO Road-Vehicles Research Institute, where the Madymo system is developed and exploited. Madymo provides (non-interactive) simulation of motions, and is aimed towards reliable simulation, so safety issues regarding road vehicles can be accurately studied. Madymo already incorporates dynamics calculations, so Dynamo itself is not so much of interest of TNO, but the software design aspects of the work described here are.

1.1 Previous work

For several decades, dynamics calculations have been applied in the field of simulation. Only recently, the computing power has been available to apply dynamics in interactive simulations. Several distinctions can be made with respect to software in the field of motion dynamics.

(i) Modular versus monolithic architectures

The existing systems differ considerably in architecture: in most systems, the dynamics software is embedded within a specific simulation or animation program (the dynamics within Madymo is such an example), and together with that program forms a rather monolithic structure, which can give problems with maintenance and extendibility. In recent years there has been a trend to try to develop software in a more modular fashion in the form of *components*. From that point of view, a piece of software which provides dynamics can be seen as a component ([Huiz 97]). This requires a more modular architecture not only of the dynamics software, but also of the software system into which it is embedded.

(ii) Rigid moving objects versus flexible moving objects

Most methods assume rigidness of the objects that have dynamic motion. But work has also been done on flexible models, for example in [Platt 88].

(iii) Constraints restricting the degrees of freedom of the moving objects

The way the degrees of freedom of the moving objects are represented is also a major difference between existing solutions. A collection of n moving rigid bodies has $6n$ degrees of freedom (a position and an orientation for each rigid body). Suppose m degrees of freedom are to be restricted by constraints.

The dynamics and inverse dynamics calculations can be combined by expressing the degrees of freedom of an object in well-chosen generalized coordinates (as was done for Madymo, see [Wijck 96]). In that way there are only $6n - m$ variables in the system to be calculated. This approach works well in situations where the constraints connect the rigid bodies in tree-like structures (loops are generally not supported in this approach), and in situations where the connections between the rigid bodies remain the same over the course of the animation. This approach is often used in robotics.

A more general approach is to represent the degrees of freedom of each object in the $6n$ cartesian coordinates. There are then two ways in which the constraint problem can be solved.

The first is to impose the constraints by directly taking the algebraic constraint equations into account when solving the forward dynamics problem ([Haug 89]). This way, a problem of $6n$ variables has to be solved under the condition that m constraints hold.

The second is to introduce m extra variables for the constraint forces which restrict the degrees of freedom of the moving objects. First, the m constraint variables are calculated, and the resulting forces are then input for the calculations that evaluate the $6n$ rigid body variables over time. This decouples the forward dynamics calculations from the inverse dynamics calculations, as the constraint forces are just the same to the forward dynamics subsystem as any other externally supplied forces.

Note that on the latter case the m variables correspond to the restricted degrees of freedom, while the variables in the very first approach correspond to the free degrees of freedom.

The decoupling of the forward and inverse dynamics calculations in the latter approach allows for the use of the same inverse dynamics methodology in case the forward dynamics subsystem is replaced by for example a subsystem which allows for flexible (non-rigid) objects.

(iv) Accuracy versus speed

For application in simulation, the main focus is on the accuracy of the dynamics calculations, since such simulations are for example used to gather knowledge about the safety of mechanical systems (Madymo). For interactive animation, accuracy can sometimes be sacrificed in exchange for speed. A fast response can in such applications be far more important than exactness, as long as the motions still "look natural". A major factor in this regard is the way constraints are satisfied. This can be done through application of constraint forces (which accurately model the underlying physics), or by 'faking it' by directly calculating displacements and rotations ([Gasc 94], an approach which is partly based on [Overv 91]). The connections (otherwise modeled by constraints) can be also be approximated by spring damper systems ([Kell 96]). For very approximative methods, [Chenn 98] describes how only the motions of objects which are visible can be calculated, using statistical models to estimate their motion when they reappear.

(v) Impulse-based versus force-based

Some approaches take motion discontinuities such as collisions as a starting point ([Faure 96], [Faure 99]) instead of (continuous) constraints. This can give rise ([Mirti 96]) to an impulse-based approach in which objects follow a ballistic trajectory in between impulse exchanges. The timing model indiced by this approach is not very suitable for interactive animations, because of the large variations in the sizes of the time steps taken, which does not correspond to the constant time between frames of an animation. Impulse-based approaches require extra constructs to support articulated rigid body systems, because the rigid bodies then do not move ballistically in between collisions.

(vi) Differentiating the constraints or not

Constraints give rise to algebraic differential equations (ADEs). These can be solved using numerical methods, or by differentiating them ([Baraf 92], [Baraf 96]) to arrive at easier-to-solve differential equations. The latter approach suffers the risk of numerical drift, which can cause problems for longer-running animations/simulations. Such drift then has to be compensated (see for example [Baum 72]).

For each of these distinctions, the proper choice for Dynamo will be presented in Section 2.3.

1.1.1 Dynamics at the TUE

At the TUE, research towards adding dynamics to the arsenal of known motion specification and calculation tools has started quite a while ago. Loosely based on the model described in [Barz 88] (which was proposed for assembling articulated bodies, and not animating them) the ideas described in [Overv 93a], [Overv 93b] and [Baren 94]) have been implemented in the animation program *Walt*. The successor of *Walt* is the *GDP*: the Generalized Display Processor, a system which uses the direct modification and direct manipulation paradigms to allow applications and/or users to interact in a 3D environment (see [Peet 95]). Some of the results pertaining to dynamics obtained from *Walt* have been introduced in the *GDP* (see [Krijg 94]), but by far not all, and more recent studies (see [Overv 95]) have shown that the algorithms used (relaxation, Euler integration, see [Baren 94]) should be replaced by better ones. The Walt implementation was also a monolithic approach where hardly any attention was paid to the interfaces, resulting in code that was not very extendable.

These were among the reasons for the work described in this thesis. It has been carried out in two parts: the first part comprised the creation of the basic design, and implementing this as part of the GDP system. This was done in the form of a final project for the OOTI course (the postgraduate programme for Software Technology at the TUE/SAI), and is described in [Baren 96]. In the second part, this work was expanded upon by separating the implementation from the GDP and providing it as a stand alone library (making the software usable for other programs), and augmenting it with new features and upgrading the internal algorithms for more versatility and robustness. The attention towards the design aspects is also evident in the cooperation with TNO.

1.2 Organization

The remainder of this thesis is structured in the following manner. In Chapter 2 of this thesis, animation/simulation systems in general are described, and the requirements for the software are presented. The design of the software is then described in Chapter 3. In that chapter, the forward dynamics subsystem is discussed first (Section 3.2), followed by an overview of the controllers which can be used to steer the dynamically moving geometries (Section 3.3). Then the inverse dynamics subsystem is introduced (Section 3.4), which allows the modeling of more complex articulated body systems, and which can be used to perform for example collision handling. In Chapter 4, the actual constraints and controllers are described, which are specific applications of the more general theory described in Chapter 3. Examples which show some uses of the library are then shown in Chapter 5. The design process is subsequently presented in Chapter 6, and conclusions and recommendations for future work are given in Chapter 7.

2 The environment and requirements

2.1 Simulation/Animation environments

In [Telea 99m] and [Telea 99j], three categories of users of animation/simulation tools are distinguished:

The end user An end user runs animations and simulations and interacts with them.

The application designer An application designer creates applications which an end user can run. He/she does not build every application from scratch, but uses predefined functionality: the application is assmbled from functional modules, so-called components. An application designer decides which components to use and how they interact with each other and with the user.

The component designer A component designer creates modules which provide functionality and services to the application designer. A component usually concentrates on a specific type of functionality, such as rendering, or user-interfacing, or finite element analysis.

Since components have to be combined, implicitly, there is a fourth category:

The framework designer A framework designer creates a system which can be used by an application designer to easily link the different components developed by the component designers. Examples of frameworks are the GDP, the Java virtual machine, AVS and Vission ([Telea 99m]).

AVS for example is a simulation framework (see [AVS]). In AVS, an application designer assembles applications by instantiating components and connecting their ports, creating a data-flow diagram which represents the application. In for example the GDP (provided by framework designer Eric Peeters, see [Peet 95]), the component designer provides the different class libraries (for rendering, user interfacing, kinematic motion calculation, and so on). The application designer writes Looks applications, using the classes provided by the component designer and the framework designer, and deciding how an end user (who runs the Looks application) can interact with the simulation.

The focus of this thesis is on the component design phase. A library is built which offers the functionality of dynamic motion calculations. As any component, the Dynamo library has application designers as immediate users, and end users as indirect users. It also means that Dynamo library assumes the presence of a framework (also called host system) which, through the use of other components, takes care of all aspects of the animations that are not directly related to the dynamic motion calculations (such as rendering the moving objects, and high level control). This enables the focus of this thesis to be solely on the dynamic motion calculations. The next section focuses on the host system and what the Dynamo component expects from it.

2.2 The assumptions about the host system

Dynamo aims to provide the motions for some of the moving objects in an animation. These objects will be called *dyna*s from now on (see Section 3.1.1 for general marks about nomenclature and notation). The motion of other objects can be calculated using different methods (kinematics, motion capture etc.). All moving objects are visualized by the host system, or one of the other components it employs. The host system therefore already has a representation for them, which, amongst others, captures their shapes. For each frame of the animation, the output of the dynamics calculations is a new set of values for position, linear velocity, orientation, and angular velocity for each dyna. These are properties that Dynamo keeps track of. A set of these properties (position, orientation, and their first time derivatives) is called a *motion state* from now on. This assumes rigid bodies: for non-rigid bodies the deformations of the objects must also be taken into account and should be returned as a result of the dynamics calculations.

The host system should provide the information to initialize the motion state of each of the dynas (i.e. a function such as reading geometry information from a file, and thereby reading the initial position and orientation, is the responsibility of the host system). For the objects that are not under control of Dynamo, but participating in one of Dynamo's constraints, the host system should provide the motion state for each frame to enable Dynamo to evaluate the constraints.

Of all the other geometrical and topological data associated with a moving object, only the associated mass and mass distribution are of importance to the dynamics calculations (see Section 3.2.1). Providing this information (based on the geometrical details of the moving objects, for which Dynamo does not know the representations) is also the responsibility of the host system.

Setting the motion state and mass (distribution) for each dyna provides all the required initialization. Afterwards, the calculations performed by Dynamo should be interleaved with all the other tasks of the host system (such as rendering a frame, handling user input etc.). To support this, Dynamo provides a function which can be called by the host system to cause Dynamo to calculate the positions and orientations for the next frame. Calling this function at appropriate times is the responsibility of the host system.

Since Dynamo maintains constraints, the host system should take care that no actions that can violate these constraints are performed between the call to Dynamo and the rendering of the corresponding new frame. For example: moving a kinematically controlled object which is related to a dynamically controlled object by a constraint could invalidate that constraint. Any objects not under control of Dynamo, but participating in one of Dynamo's constraints, should already have the motion state for the new frame before Dynamo is called, so Dynamo can match the new motion states of the dynas to fulfill the constraints.

Simulated time is a property global to the whole animation system. The host system should decide how much the simulation time should be advanced between one frame and the next, and provide this information to Dynamo, before making the call that tells Dynamo to provide the motion states for the next frame. This will ensure that the time steps taken by Dynamo match the time steps taken by the objects that are not under Dynamo's control. Dynamo provides a method to specify this time step in its interface.

The following pseudo code summarizes the responsibilities and the behavior of the host system towards Dynamo:

```
for each dyna do
    set initial motion state and mass distribution
od;
while animating do
    perform host specific tasks (such as handling user input);
    set the time step for this frame;
    update all motion states not under Dynamo's control, but
      participating in Dynamo's constraints;
    call Dynamo;
    render the new frame
od
```

During Dynamo's development, the GDP (see [Peet 95]) was used as a host system, enabling the use of its framework and already available libraries for testing, and creating the demo examples described in Chapter 5.

2.3 The requirements of Dynamo

Dynamo has been developed in two stages. The first part was performed as a final project of the OOTI course, and in this part the basic design was made. In the second part, this design was expanded upon through the addition of several new features and refinement of existing features.

With respect to the criteria presented in Section 1.1 (previous work), the basic design developed in the first part was subject to the following requirements:

(i)a It should have a modular architecture: i.e. it should not be part of a monolithic structure.

(i)b The design should allow for flexibility and ease of extendibility, and its interface should provide easy access to its functionality.

(ii) Rigidness of the moving objects can be assumed, although easy extendibility to flexible geometries should be taken into account.

(iii) To support the the flexibility of being able to add and delete constraints on the fly and not limiting the constraint configuration to a tree structure, the representation of the degrees of freedom should be in cartesian coordinates (as opposed to well-chosen generalized coordinates; see the text under the third header in Section 1.1). This means that the rigid objects are all positioned in world coordinates, and not relative to one-another.

(iv)a The design should be suitable for interactive animations: i.e. speed is very important, and as such, accuracy may be sacrificed in exchange for more speed (if absolutely necessary).

(iv)b However, the constraints should be satisfied using explicitly calculated constraint forces. 'Faking it' such as described in Section 1.1 under the header of accuracy versus speed, would likely not allow for sufficient accuracy for application in more simulation oriented areas. Having explicitly calculated constraint forces allows the inverse dynamics subcomponent to make decisions based on the occurring constraint forces (which can be interpreted in a very natural way), and for the application designer to inspect them. The availability of *all* forces arising in a simulated system (including those resulting from constraints and controllers) allows for a greater insight in the dynamics of such a system.

(v) The dynamics system should not be purely impulse-driven, but fully support forces on the dynas to support articulated rigid body systems.

(vi) The algebraic differential equations arising from the constraints should be solved numerically to avoid drift in long animations (such as in virtual reality applications).

Furthermore, the basic design should also:

- incorporate techniques, where ever possible, that improve upon the versatility of the earlier TUE designs, to alleviate the problems identified with the work described in [Baren 94]:
    - accuracy is limited by the sole use of Euler integration (this relates to the requirement of flexibility: requirement (i)b)
    - the earlier design (or lack thereof) does not allow addition of new constraint types without the need for modifying all existing constraint types (this relates to the requirement of extendibility: requirement (i)b)
    - the use of relaxation (a method for solving constraints locally) does not support very long chains of objects (this relates to requirement (iii), since relaxation seriously limits supported constraint configurations because it can not handle long chains).
    - the use of constraints to enforce rigidity adds a large number of constraints, which are expensive to solve (this relates to requirement (iv)a: speed is an important issue for Dynamo).
- provide the interface that the host system needs to fulfill its requirements (see previous section).

To facilitate rapid prototyping, the dynamics functionality was incorporated directly into the GDP in the first phase (keeping a future separation in mind). Dynamo was uncoupled from the GDP, before the work on the actual extensions and improvements in the second phase could commence. Then the following extensions were required:

1. A user manual for the stand-alone library.
2. More examples of possible applications of the library, adding constraints (such as the wheel constraint for the bicycle example) as needed.
3. Improvement of the collision handling provided by the basic implementation (for the collision constraint that will be presented in Section 4.1.14, this entails the addition of the positional terms that prevent the positional aliasing when collisions are used in prolonged contact).
4. Addition of a way to model friction.
5. Addition of more curves and surface types for the point-to-curve and point-to-surface constraints.
6. Addition of handling near mass-less objects, such as ropes. These objects only transfer forces (the total force on them is always zero), but do not significantly contribute any mass and therefore can not be modeled by dynas, because the motion equations (see Section 3.2.1) can not be applied for (near) zero masses.
7. A mechanism for creating controllers. Controllers are a means to actively steer the dynas (whereas constraints are only reactive), see Section 3.3. A few controllers should be provided as examples. This should be done in such a way that it is easy to extend the library with more controllers later.
8. Increase the robustness of the inverse dynamics subsystem by adding handling of over-specified and under-specified constraint combinations.
9. Share the experience in software engineering techniques by providing consultancy to the Madymo group at TNO.

During the project these requirements were monitored in regular meetings to ensure that they were met to satisfaction.

3 The design

3.1 Methodology

3.1.1 Views and notation

In designing software that deals with simulating physics, three views can be distinguished:

Physics view In this view the real world physics is observed.

Modeling view In this view a (mathematical) model of the real world physics is described. Some details from the real world physics are abstracted from or simplified.

Software view In this view the model is implemented in a software structure.

For example, a physical object has mass. This mass can be modeled in a variable m which denotes the amount of mass of the object. In the software, this amount will likely be administrated in an attribute `mass` of a class which implements the model of the physical object.

These three views are very closely related and are referred to in an interwoven manner in the text of this thesis. As shown above, the physical view is denoted in regular font, the model view in slanted notation, and the software view in typewriter font. However, strictly adhering to this notation would give a very verbose text. For example: many modeling parameters map one-to-one to class attributes. Explicitly introducing these would be very cumbersome, and therefore the modeling parameters are used directly in some of the pseudo code and the text. So, for sake of brevity, it is sometimes left to the reader to distinguish between the different views, given the context in which they are used.

3.1.2 Software design phases

Three design phases are identified in the software design of Dynamo:

Interface design The interface should provide a uniform way for an application designer to specify the required motions and relations between the objects that are animated.

Algorithm design The algorithms have to be selected and (if necessary) newly designed, to provide the functionality specified by the interface.

Code design The code should be structured such that the different classes interface with each other in a manner that they collaborate to implement the algorithms. In this phase the interfaces that are internal to the library are designed.

More information on these phases, and an explanation of how these phases fit into the overall software lifecycle model are presented in Section 6.2.

3.1.3 Object orientation

The design process and methodology are discussed in more detail in Section 6, but one very important design decision needs to be mentioned here:

Design decision 1: The design of Dynamo is done in an object oriented fashion.

There are several reasons for this, some of which are:

- Object orientation allows for a common language in discussing and reasoning about the system in all of its views (even so much that the notational problem discussed at the end of Section 3.1.1 arises). The mechanical systems that are modeled by Dynamo lend themselves very well to modeling in an object oriented fashion, because the identification of the classes of physical objects such as dynas and constraints comes very natural. The physical 'interfaces' of these classes also map very easily to the software interfaces (for example: since pushing and pulling a dyna is the only way to make it move, a method such as `applyforce` comes naturally; see Section 3.2.2).

- Object oriented design allows for a clear separation between interface and implementation. This allows for a higher level of abstraction at the interface level, which aids in clearer (and often smaller) interfaces. The implementation behind those interfaces can be varied and improved, without affecting other components (changing and improving algorithms is a constant focus of attention in research-oriented design such as that of Dynamo).

- Object orientation supports several constructs (such as inheritance and object composition) to specify relations between subcomponents (classes). This allows a designer to provide a clear structure between the components of a library, as opposed to just providing a large collection of functions and procedures. Using this structure, abstractions can be carried through in the design in all its phases, so that for example the abstract notion of a constraint can be found from requirements to implementation. This helps in understanding, maintaining and extending the library. Of special interest is the inheritance mechanism, which can be used to specify abstract interfaces. In this way, a general interface specification such as “the constraint interface” can be defined, enabling a uniform interface for all constraints (ease of understanding and interfacing), and providing a “recipe” for extending the library with new constraint types (one simply has to implement the methods specified by the constraint interface to implement a new type of new constraint). Such use of inheritance allows for re-use of structure.

- Many frameworks are also based on object oriented design (mainly because of the same reasons that Dynamo is designed in an object oriented fashion), making it easier to incorporate Dynamo in these frameworks. See for example Section 5.5 for an example that uses the advantages of object orientation next to the functionality of Dynamo to model a more complex system.

For a more elaborate discussion on the merits of object oriented design, see Chapter 6.

The following sections address these design issues for the different components of Dynamo. First the forward dynamics subsystem is discussed. The controller and inverse dynamics subsystems then build on the forward dynamics functionality in calculating forces that can be input for the forward dynamics calculations.

3.2 Forward dynamics

This section deals with the design and implementation of the forward dynamics subsystem. This subsystem should allow motions of the moving objects in an animation to be computed in accordance to physics, i.e. based on inertia and applied forces. In order to understand the issues, we first need to take a look at the physical concepts that are being modeled. This is presented in Section 3.2.1 below. From the physics, we can derive the software interface through which this functionality can be made available. This is presented in Section 3.2.2. From the equations given by physics, we can also derive the mathematical schemes needed to implement the interface (Section 3.2.3). Once these schemes are known, we can incorporate them in a refinement step of the classes for which we earlier defined the interfaces (Section 3.2.4). Afterwards, a short summary of the preceding sections is given (Section 3.2.5).

So we first take a look at the physics view to derive the model.

3.2.1 Physics

The motion equations below are the results of a more detailed derivation presented in [Witt 77]. For a more comprehensive treatment of the physics see for example [Alonz 70].

Geometries and dynas

We start by introducing the moving objects which are present in the animations/simulations. We will call these the *geometries* (also called *rigid bodies* sometimes). For dynamics it is important to know the position and orientation of a geometry, and their derivatives: linear and angular velocity. This information is called the *motion state* of the geometry (as mentioned in Section 2.2). The position and orientation of a geometry define a local coordinate system. This induces the distinction between local coordinates and world coordinates. Local coordinates are very convenient to denote positions within a geometry (since the geometries are assumed to be rigid, these remain constant), but world coordinates are required when comparing for example positions between geometries (as is often done in constraints).

The position of the origin of a geometry will be denoted by vector z, and the orientation by orthonormal rotation matrix A. Conversion from local coordinates ρ to world coordinates p is done using:

$$p = z + A\,\rho$$

The linear velocity of the geometry is denoted by v, and the angular velocity by ω. The velocity in world coordinates $\dot{p}$ of a point of a dyna with local coordinates ρ can also easily be expressed:

$$\dot{p} = v + \omega \times A\,\rho$$

The motion of a rigid body is governed by its mass m (determining its linear inertia), and by its tensor of inertia J (determining its rotational inertia, see also Appendix D).

For the motions equations presented in the next section to be valid, there is an extra requirement on the position z of a dyna: the local coordinate system of the dyna should have its origin in its center of gravity (i.e. z needs to denote the centroid of the dyna).

It is also very convenient if the axes of the local coordinate system are aligned with the major axes of inertia of the dyna: this causes the tensor of inertia to be diagonal when expressed in the local coordinate system. We will require this property from initial orientation A.

For rigid objects, the tensor of inertia matrix J is a constant (and diagonal as we just assumed) when it is expressed in the local coordinate system of the object. It is variable when it is expressed in world coordinates: $J_w = AJA^T$. More on the tensor of inertia for the special case of one-dimensional objects can be found in Appendix D.

Inertia

When there are no external influences to a dyna, it moves under the effects of inertia only. For the position this means that $\dot{z} = v$. Naturally, the angular velocity ω and the change in orientation ($\dot{A}$) are also related. This relation can easily be expressed when orientation is represented by *Euler parameters* (see section 2.1.3 of [Witt 77]). An orientation which corresponds to a rotation with respect to the world coordinate system over an angle χ around axis u (where $||u|| = 1$) is represented by a unit-length four-dimensional vector q, where $q_0 = \cos(\frac{\chi}{2})$ and $q_{1..3} = u\sin(\frac{\chi}{2})$. Orientation matrix A can be expressed in terms of q as follows:

$$A = \begin{pmatrix} 2(q_0^2+q_1^2)-1 & 2(q_1q_2+q_0q_3) & 2(q_1q_3-q_0q_2) \\ 2(q_2q_1-q_0q_3) & 2(q_0^2+q_2^2)-1 & 2(q_2q_3+q_0q_1) \\ 2(q_3q_1+q_0q_2) & 2(q_3q_2-q_0q_1) & 2(q_0^2+q_3^2)-1 \end{pmatrix} \tag{3.1}$$

So it is sufficient to know q to be able to calculate A (and vice versa). The correspondence between ω and the change in orientation is more easily given in terms of the representation using Euler parameters (specifying the relation between ω and $\dot{q}$). Introducing the $\star$ operator), this relation is given by:

$$\dot{q} = \frac{1}{2}\begin{pmatrix} 0 & -\omega^T \\ \omega & -\tilde{\omega} \end{pmatrix} q \overset{def}{=} \omega \star q$$

where $\tilde{\omega}$ is the matrix encoding the cross product with ω:

$$\tilde{\omega} = \begin{pmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{pmatrix} \tag{3.2}$$

Forces and torques

When forces and/or torques are being applied to a dyna, this causes changes in its velocity (linear and/or angular). Central forces F (i.e. forces applied to the centroid of the dyna) affect the linear velocity, according to

$$m\dot{v} = F$$

Torques M affect the angular velocity, according to

$$J_w\dot{\omega} + \tilde{\omega}J_w\omega = M$$

A non-central force f (applied to point p) affects both the linear and angular velocity. Its effect on the linear velocity is equivalent to a central force $F = f$, while its effect on the angular velocity is equivalent to that of a torque[1] $M = (p - z) \times f = A\,\rho \times f$.

Together with the equations for inertia motion, the equations of motion of a dyna then are:

$$\begin{aligned} \dot{z} &= v && (3.3) \\ \dot{v} &= \frac{1}{m}F && (3.4) \\ \dot{q} &= \omega \star q && (3.5) \\ \dot{\omega} &= J_w^{-1}(M - \tilde{\omega}J_w\omega) && (3.6) \end{aligned}$$

Forward integration of these equations over time yields the motion of a dyna. This is discussed further in Section 3.2.3.

Impulse changes

For forces working over a very short time (like those occuring as a result of collisions), it is often easy to work directly with changes in impulse:

$$\iota(t) = \int_0^t f(\tau)d\tau$$

Using this approach, the exact values of the forces during the time interval from 0 to t need not be known, just their overall effect on the change of impulse ι. This is especially useful for very large forces working over a very short time interval. In the limit where the interval duration goes to zero and the forces to infinity, the impulse change remains finite. In this

---

[1] Note that the torque is derived from the (non-central) force: forces are fundamental (as used in for example Newton's *action=–reaction* principle), while torques are not. We will base a design decision (design decision 7) on this later.

case, the effects of non-impulsive forces and inertia vanish, so that the motion equations then become (for an impulsive force applied to the point with local coordinates ρ):

$$\begin{aligned} m\Delta v &= \iota \\ J_w \Delta\omega &= A\,\rho \times \iota \end{aligned}$$

or:

$$\Delta v = \frac{1}{m}\iota \tag{3.7}$$

$$\Delta\omega = J_w^{-1}(A\,\rho \times \iota) \tag{3.8}$$

3.2.2 The forward dynamics classes

The geometry class

In the previous section, the physics were presented that are to be modeled in the Dynamo software. This section will present the structure and the interface of the software which will provide easy access to this functionality.

As mentioned in Section 2.2, host systems for Dynamo will already have some notion of a geometry (for purposes of rendering etc.), while Dynamo will only administrate the properties it needs in companion objects:

Design decision 2: Dynamo will manage *companion objects* to the host system's geometries.

Callbacks are defined, which a user must implement in order to specify how the companion objects can interface with the geometries from the host system. Using these callbacks, a user can tell Dynamo how to retrieve and update the motion state of a geometry of the host system.

Design decision 3: Companion objects interface with their counterparts through callbacks.

For more information on the callbacks, see Section 2.2, and the user manual [Baren 99].

Dynamo's companion objects are therefore nothing more than a means for the other Dynamo classes to interface with the geometries of the host system: the companion objects administrates and provides access to the motion state(s) of the host system's geometries through an interface which is known to the other Dynamo classes. The callbacks are used to keep the motion states of the companion objects and the host system's geometries in correspondence.

Since time is discrete in animation systems (in an animation, the only points in time of interest to an end user are those for which a frame is rendered, since those are the only points in time he/she can observe), there are two motion states which are of interest when calculating the motions from one frame to the next. The first is the motion state for the latest frame (sometimes also called the current frame), which is the result of the motion calculations from last frame, and which functions as a starting point for the motions of the geometries between this frame and the next. The second is the motion state for the next frame, which is being calculated. We will denote the time stamp of the former with t, and the duration time between frames with h (so the time stamp of the next motion state is $t + h$). The companion objects should administrate both these motion states.

There is a clear distinction between two groups of geometries. On the one hand, there are the dynas for which the motions are calculated by Dynamo. On the other hand, there are also other geometries which are not under Dynamo's control (these can be kinematically controlled for example).

From the view of the host system, there are many different kinds of geometries, each of which have different motion behavior. All geometry types have in common that they administrate and provide access to their motion states. They differ in the way that their motion behavior is modeled, and thereby in the way new motion states are calculated. Only for dynas, new motion states are calculated using equations 3.3–3.8. Other types of geometry might use kinematics calculations for example. To be able to interface with every type of geometry present in the host system, the companion objects in Dynamo are companions to the host system's geometries: a `geometry` class is introduced in Dynamo (from now on, reference to "`geometry`" will refer to Dynamo's geometry class, and not the host system's geometries, unless specifically stated otherwise). Dynamo's companion objects to the host system's geometries belong to this `geometry` class.

The `geometry` class provides methods for inspecting and changing both motion states, and for conversions between local and world coordinates based on both motion states.

The dyna class

Dynamo's dynas add the dynamics-specific motion behavior to some of the geometries. Therefore:

> **Design decision 4:** Inheritence from the `geometry` class is used to introduce forward dynamics.

This allows for reuse of the motion states administrated by the `geometry` class, and the interface through which it can be accessed. It also allows reuse of the callbacks the user defined to communicate the motion states. The specialization of the `geometry` class is the `dyna` class.

The `dyna` class adds several attributes and methods with respect to its parent class to provide access to the dynamic motion functionality.

First of all, it adds attributes for administrating the required mass and tensor of inertia of the dyna, so the `dyna` class can evaluate the motion equations presented in Section 3.2.1.

Second, there are methods that can be called to apply forces, torques and impulse changes to a dyna:

```
applyforce(p:point; frombasis:geometry; f:vector):void;
    // Force 'f' is applied to point 'p', expressed in the coordinate
    // system of 'frombasis'
applycenterforce(f:vector):void;
    // f in world coordinates
applytorque(t:vector):void;
    // t in world coordinates
applyimpulse(p:point; frombasis:geometry; i:vector):void;
    // Impulse change 'i' is applied to point 'p', expressed
    // in the coordinate system of 'frombasis'
```

We will come back to the exact semantics of these methods in Section 3.2.3.

Third, it should be possible to trigger a dyna to advance a frame. This entails making sure that the next motion state is up to date with respect to the equations of motion presented in Section 3.2.1, and the forces, torques and impulse changes that have been applied to it. Then the frame advancement (effectively increasing time) is performed by shifting the next motion state into the current motion state. The corresponding method of the `dyna` class is called `next_frame`.

Regular (once a frame) triggering, interleaved with calls to the four '`apply`' methods introduced above, mimics the motion under influence of inertia and the applied forces, torques and impulse changes.

One other extension to the interface of the `dyna` class is made: the introduction of the notion of velocity damping, as a crude (but very effective) approximation of friction with the medium in which a dyna exists. Methods are introduced to set and retrieve the velocity damping factor (which lies between zero and one). This factor is applied to the velocity (both linear and angular) when shifting the next motion state to the current motion state at the end of each frame (taking the step size into account), thereby slowing the dyna down.

The dyna_system class

Several properties are global to all dynas. The step size h signifying the length of the time interval between one frame and the next is such a property. To improve the interface with an application designer, a class is introduced to administrate such global properties, and to provide a single entry-point for the dynamics calculations. The `dyna_system` class can maintain a list of all dynas, and trigger all of of them in response to a single call to a method "`dynamics()`" of its own. This way, an application designer does not have to trigger each `dyna` individually. After triggering the `dynas`, the `dyna_system` can also call the callbacks for each of the `dynas` to copy their newly calculated motion states to the host system, in the `dynamics` method (see the end of Section 3.2.4).

For the `dyna_system` to be able to keep track of all dynas, a `dyna` registers itself with the `dyna_system` in its constructor, and 'checks itself out' in its destructor. Similarly, each non-dyna `geometry` registers itself with the `dyna_system`, so the `dyna_system` can make sure that it calls the callbacks which ensure that the companion's motion state is updated when required (see the pseudo code in Section 3.3), relieving the user of keeping the companions up to date.

Having a central `dyna_system` class, enables easy introduction of other global properties, such as gravity. The `dyna_system` class will have an attribute for administrating this property (with methods for setting and retrieving it), and the `dyna_system` can apply gravity to all dynas (using their `applycenterforce` method) in its `dynamics` method, so that gravity will always be applied without any user interaction, once a user has specified the gravity vector.

Intermezzo

Methods such as the methods of the `dyna_system` that a `dyna` can use to register itself, and a method like the 'trigger' method of the `dyna` class are part of the interface of the class that they belong to. However, they are only for use within Dynamo itself. These are not methods that a user of Dynamo should use (or even be aware of). Many more such methods will be introduced further on in this thesis. One of the short-comings of C++ (in which Dynamo is implemented) is the lack of support for this 'library', or 'package', or 'module' notion, which would allow an easy way to specify which methods are exported outside the library, and which are meant for internal use only (the `friend` notion in C++ is too cumbersome for this purpose since it needs explicit naming of the classes within the library, which limits extendibility). This is a specific example of the problem identified in Section 4.3.1 of [Aksit 92]. In the following text, we will denote this difference as 'exported methods' and 'internal methods' (so these are two 'flavors' of public methods, next to private methods). In the user manual of Dynamo (see [Baren 99]), only the external methods are documented to deter a user from using the internal ones, but C++ does not provide a mechanism to prevent the user from actually using any internal methods.

3.2.3 Integration of the motion equations

In the previous two sections, the interfaces of the forward dynamics classes were presented. To be able to implement the functionality specified by these interfaces, the motion equations presented in Section 3.2.1 need to be integrated.

Integration step size

The step size h of the simulation is administrated via the `dyna_system` class. This step size can be adjusted by the user. Furthermore:

> **Design decision 5:** The step size is determined by the user alone: Dynamo does not modify the step size.

This decision is made because the time step is global to the whole animation. If Dynamo were to change the time step it uses, the dynas new motion state would be calculated for a different time stamp than the motion states of the geometries that are not under Dynamo's control. Changing the time step, and notifying the host system of this change, would require the host system to recalculate the motion states for the other geometries also, which it may not be able to do (it would require all other motion calculation components to support backtracking a frame).

Also, in real-time computer animation, the step size is often directly related to the simulated time between subsequent frames. If Dynamo were to change the step size on its own, this would result in a perceived slowing down or speeding up of the animation. Such time warping is not desirable.

Since integration errors depend on the step size, this also provides users with a mechanism for error control: users should use a smaller time step if they want more accurate results. Unfortunately, it also burdens the user with the responsibility to provide a step size small enough for the calculations to be stable. The user is aided in solving this problem by providing methods that report back for example the constraint error: these can indicate if the step size is too large.

Time warping can also result from changes in the amount of calculations that need to be performed to calculate the next motion state: if suddenly there is a large increase in the required calculations (when for example many collisions occur), there will also be a perceived slow-down. To try to counter this effect, the following design decision is made:

Design decision 6: Dynamo does not subdivide the time step specified by the user's step size

In other words, Dynamo does not try to compute a consistent motion state for all dynas for any moment between t and $t + h$. Again, such an intermediate motion state might require information about the non-dynamically controlled geometries, which might not be available from the components that calculate their motions.

This still does not guaranty a constant computation time per frame, but at least it precludes scenarios where the simulation comes to a grinding halt because for example many collisions cause the time interval between t and $t + h$ to be subdivided in many smaller intervals which each cost as much time to integrate as normally the whole interval would.

A user can compensate for the remaining time warping by setting the time step for each frame based on the difference between the simulated time and the time provided by a real-time clock (similar to the way Java handles the correspondence between real time and simulated time).

Design decision 6 poses some limitations to the ways Dynamo can handle for example collisions (see Section 4.1.14) and apply impulses (see design decision 10).

Constant forces and torques

To facilitate the integration of the motion equations presented in Section 3.2.1, equation 3.6 is rewritten to

$$\dot{\omega} = AJ^{-1}A^T(M - \tilde{\omega}AJA^T\omega) \tag{3.9}$$

(using $J_w = AJA^T$), so that the constant J can be used, and only the motion state variables are non-constant, next to the total force F and total torque M.

The dependency of F and M on time within the time span from one frame to the next depends on the semantics of the `applyforce`, `applycenterforce`, and `applytorque` methods. Because of the discrete modeling of time, we will take these semantics to be that a *constant* force or torque is applied to the dyna between the current frame and the next. But the term *constant* is still open to interpretation, as is shown below.

For a non-central force, the applied torque M is specified by $M = A\rho \times f$, as was seen in Section 3.2.1. This means that the torque exerted by f depends on the orientation of the object. As the torque is being applied, its rotational acceleration is added to already present inertial rotation of the dyna. So the orientation will likely change between t and $t+h$, causing in turn a change in the amount of torque that results from force f.

There are two ways to interpret the term *constant* when it comes to force vector f:

1. It can be kept constant in world coordinates. This means that the above mentioned effect takes place, and the resulting torque will vary over the time interval $[t..t+h]$.
2. It can be kept constant in local coordinates. This means that the force will "rotate along" with the dyna. This yields a constant torque between t and $t+h$, but as the direction of the force now varies in world coordinates, the linear effect of the force is no longer constant.

Likewise, we can choose to keep the application point of the force constant in local or in world coordinates (but keeping it constant in world coordinates does not make much sense mechanically).

Many forces that are encountered are reaction forces, which come in pairs: a force which is applied to one dyna, and a counter force which is applied to another dyna (satisfying Newton's *action=−reaction* principle). To ensure that such reaction force pairs remain each other's opposites during the entire integration interval (see the footnote on page 15), we choose to keep forces constant in world coordinates.

Design decision 7: When applying forces, the forces are kept constant in world coordinates, and the application point is kept constant in local coordinates, over the course of the integration interval.

Methods of integration

Design decision 7 results in a constant central force F between t and $t+h$, which means that equations 3.3 and 3.4 can be integrated analytically:

$$z_{t+h} = z_t + v_t h + \frac{1}{2}\frac{F}{m}h^2 \tag{3.10}$$

$$v_{t+h} = v_t + \frac{F}{m}h \tag{3.11}$$

For the orientation and angular velocity, numerical integration is required however. For a numerical integrator $\aleph$:

$$< q_{t+h}, \omega_{t+h} >= \aleph(< q_t, \omega_t >, M) \tag{3.12}$$

Several integrators are available, each with their own precision and computational cost (see Appendix B for some examples). To allow a user to balance the precision and computational cost in his/her application, we decide:

> **Design decision 8:** We will leave the choice of the motion integrator open to the user.

We will take a look at the effects of this decision on the software in the next section.

The more advanced integrators evaluate the differential equations (equations 3.5 and 3.9) for several time stamps between t and $t + h$. This is where the effect of the varying torque comes in.

We might choose not to account for this effect, by just calculating the torque corresponding to a given force once (using for example the orientation of the current motion state) and then using that torque for each evaluation of the differential equations. This makes forces that rotate along with the dyna for the duration of the frame indistinguishable from forces that do not. In this case, the error which is made is of the order of the discretization step. Accounting for this effect at each evaluation of the differential equations means that the error is of the order of the integrator which is used: if an integrator chooses to evaluate the differential equations for different time stamps in the integration interval, it will get more accurate results. In this way a user can get more accurate results by choosing a higher-order integrator. Such scalability can be advantageous when using Dynamo for simulation purposes, so the little overhead caused by the extra administration is taken for granted:

> **Design decision 9:** We re-evaluate the torque caused by a force at each evaluation of equation 3.9 to account for the change in orientation, and thereby gain more accuracy.

Before looking at how these decisions are incorporated in the `dyna` class, we still have to define the precise semantics of the `applyimpulse` method.

Impulse changes

In taking a time step from time t to time $t+h$, impulse (ex)changes could occur at any moment during the interval. However, allowing an impulse change to be applied at any moment in the interval would require sub-sampling the time interval (by maintaining a list of impulse-changes-to-be-applied, and doing the integration of regular forces in a piecewise fashion). This would computationally be very expensive, which is unacceptable in the time-critical motion integration, as was decided in design decision 6. It would also mean that the user's choice in motion integrator would give him/her no guaranty for the computational effort used in an integration step, which would mean a user would not be able to properly balance precision against computational speed. Therefore:

> **Design decision 10:** Impulse changes are applied at the start of the integration interval, not at arbitrary points in time during the interval.

The impulse changes are also assumed to be instantaneous, so that indeed the effects of inertia and non-impulsive forces can be neglected as the effects of the impulse changes are calculated, as denoted by equations 3.7 and 3.8.

3.2.4 Forward dynamics classes revisited

Now that we know how the motion equations will be integrated, we can revisit the forward dynamics classes and see how the notions discovered in the previous section influence these classes, and can help implementing the methods specified in the interfaces given in Section 3.2.2.

The motion state after impulse changes

Due to the possible application of impulse changes at the start of the integration interval, there are now three motion states of interest to a dyna while it is making the step from time t to $t+h$. The two motion states administrated by the `geometry` class, and also the motion state at time t after impulses are applied. The two motion states at time t will be distinguished by t^- and t^+. The former is the motion state before application of impulse changes (and is therefore the committed result of the motion calculations of the previous frame), and the latter is the motion state after application of impulse changes (which may change as more impulse changes are applied).

The `dyna` class is extended with an attribute for the third motion state, and the related methods, so that it can be queried and set just as the two other motion states using the methods of the `geometry` class. The methods inherited from the `geometry` class for setting the motion state at time t are re-implemented to set both the states at t^- and t^+.

For the motion equations, application of an impulse change results in:

$$\begin{aligned} v_{t^+} &= v_{t^-} + \Delta v_t \\ \omega_{t^+} &= \omega_{t^-} + \Delta\omega_t \end{aligned}$$

and since positions are not affected in an infinitely short time interval:

$$\begin{aligned} z_{t^+} &= z_{t^-} = z_t \\ A_{t^+} &= A_{t^-} = A_t \\ q_{t^+} &= q_{t^-} = q_t \end{aligned}$$

Time integration for the regular forces is now based on z_t, v_{t^+}, A_t, q_t and ω_{t^+}.

The motion integrator classes

To implement equation 3.12 we will have to use a numerical integrator to provide function $\aleph$, and design decision 8 specifies that the choice of the motion integrator is up to the user. The motion integration is therefore encapsulated in a separate class hierarchy: an abstract `m_integrator` class specifies the interface which a dyna can use to integrate its motion state, while inheritance is used to implement this interface in different ways specific to each motion integrator. This will also allow for easy addition of new motion integrator types.

The `integrate` method is the most important method of the `m_integrator` class. It returns the rotational part of the motion state for the next frame given the rotational part of the motion state for this frame and a reference to the dyna whose motion state is being integrated.

The dyna class offers support for the motions integrators through the addition of (internal) method "`ode`". This method implements the right-hand sides of equations 3.5 and 3.6 yielding for (the rotational part of) motion state y: $\dot{y} = ode(y)$ (the forces and torques that are required for this evaluation are attributes of the `dyna` and do not have to be given as parameters of the `ode` method).

The dual representation of the orientation (in both A and q) causes a small complication: after a new value for q has been calculated, A should be updated as well: the integrator should call the `A2q` method of the new motion state. To illustrate the use of this method, here are the `integrate` methods for Euler and for Runge Kutta 4 (more on the specific integrators and the discretization of the motion equations can be found in Appendix B):

```
euler::integrate(y,d): supvec {
  var ytemp: supvec;
    ytemp:=y + $h$ d.ode(y); ytemp.q2A();
    return ytemp;
}
```

Note that this method, despite the syntax used here, will be implemented in C++ (as the method below and most other code presented in this report).

```
rungekutta4::integrate(y,d): supvec {
  var k1, k2, k3, k4, ytemp : supvec;
     k1:=d.ode(y );      ytemp:=y+$\frac{h}{2}$k1;                       ytemp.q2A();
     k2:=d.ode(ytemp);   ytemp:=y+$\frac{h}{2}$k2;                       ytemp.q2A();
     k3:=d.ode(ytemp);   ytemp:=y+$h$k3;                                 ytemp.q2A();
     k4:=d.ode(ytemp);   ytemp:=y+$\frac{h}{6}$(k1 + 2 k2 + 2 k3 + k4);  ytemp.q2A();
     return ytemp;
}
```

The administration of the integrator is handled by the `dyna_system` class which was introduced earlier: it will have a reference to the integrator which is currently used, and there is a method through which the `dynas` can obtain this reference when they need a motion integrator.

`Dyna`'s attributes

Motion integration could be performed in each call to `applyforce`, `applytorque`, `applycenterforce`, and `applyimpulse`. However, since it is likely that more than one force, torque and/or impulse change is applied to a `dyna` before the next motion state is required, we add some administration to keep track of the forces and torques which have been applied so far, to minimize the number of times that the (computationally expensive) integration is performed. Integration is then only performed when the next motion state is required. This optimisation is possible because –under its interface– the `dyna` class is solely responsible for how it performs its functions.

Design decision 11: Motion integration is performed on demand, rather than each time a new force or torque is applied.

Two private boolean attributes `ShouldIntegrateTrans` and `ShouldIntegrateRot` are introduced. The former adminstrates the invalidity of the next motion state's position and linear velocity. The latter the invalidity of the orientation and angular velocity. This way, the results of an integration step can be reused until they are invalidated by application of forces, torques, impulse changes, or a shift to the next frame. A method `integrate` is introduced in the `dyna` class. This method is called each time when the next motion state it required, to make sure that that next motion state is up to date.

The total amount of central force (due to calls to `applycenterforce` and `applyforce`) and total amount of torque (due to calls to `applytorque`, but not to `applyforce`) are administrated by the `dyna`: it has vector-typed attributes `F` and `M` to do this. For the torque effect of forces that are applied through the `applyforce` method, a list `fList` of force/vertex pairs will have to be maintained since the particular orientation at which the torque is required is not known in advance, since this depends on the time stamp for which the integrator needs to evaluate the motion equations.

`Dyna`'s methods

The pseudo code below shows how the considerations presented in the previous sections are captured in the `dyna` class.

At the very end of each frame the newly calculated motion state is shifted into both current motion states (accounting for the velocity damping). Both motion states at t being equal signifies that no impulses have yet been applied during the ensuing frame. The `ShouldIntegrateTrans` and `ShouldIntegrateRot` attributes are set to `true` to indicate that there has been no integration step performed yet for the next frame:

```
dyna::next_frame() {
    integrate();
    $< z_{t^-}, v_{t^-}, q_{t^-}, A_{t^-}, \omega_{t^-} > := < z_{t+h}$,vd*$v_{t+h}, q_{t+h}, A_{t+h}$,vd*$\omega_{t+h} >$;
    $< z_{t^+}, v_{t^+}, q_{t^+}, A_{t^+}, \omega_{t^+} > := < z_{t^-}, v_{t^-}, q_{t^-}, A_{t^-}, \omega_{t^-} >$;
    F:=$\vec{0}$;   M:=$\vec{0}$;   fList.makeempty();
    ShouldIntegrateTrans:=true;   ShouldIntegrateRot:=true;
}
```

where `vd` is the velocity damping factor.

The four methods which can be used to apply the different kinds of forces to a dyna simply modify the adminstration of the forces, torques and/or impulses which have been applied this frame:

```
dyna::applycenterforce($f$) {
    F+=$f$;  ShouldIntegrateTrans:=true;
}

dyna::applytorque($M$) {
    M+=$M$;  ShouldIntegrateRot:=true;
}

dyna::applyforce($p$,$g$,$f$) {
  var $p_{local}$:point;
    $p_{local}$:=to_local($g$.to_world($p$));
    F+=$f$;  fList.add(<$p_{local}$,$f$>);
    ShouldIntegrateTrans:=true;   ShouldIntegrateRot:=true;
}

dyna::applyimpulse($p$,$g$,$\iota$) {
  var $p_{local}$:point;
    $p_{local}$:=to_local($g$.to_world($p$));
    $v_{t^+}$+=$\frac{1}{m}\iota$;   $\omega_{t^+}$+=$A_t J^{-1} A_t^T (A_t p_{local} \times \iota)$;
    ShouldIntegrateTrans:=true; ShouldIntegrateRot:=true;
}
```

As seen in the last method, impulse changes applied during a frame are directly accounted for in the motion state for t^+. This motion state then functions as a starting point for the algebraical and numerical integration to the motion state at $t + h$.

To really integrate the motion variables, the dyna class has the (internal) `integrate` method:

```
dyna::integrate() {
    if (ShouldIntegrateTrans) {
        $z_{t+h}$:=$z_{t^+} + v_{t^+}h + \frac{1}{2}\frac{F}{m}h^2$;   $v_{t+h}$:=$v_{t^+} + \frac{F}{m}h$;
        ShouldIntegrateTrans:=false;
    }
    if (ShouldIntegrateRot) {
        $< q_{t+h}, \omega_{t+h} >$:=dyna_system.get_integrator().integrate( $q_t$,$\omega_t$,$this$);
        ShouldIntegrateRot:=false;
    }
}
```

For the motion integrator to be able to integrate the rotational part of the motion state, the dyna provides the `ode` method (only to be called by the motion integrators):

```
dyna::ode($< q, \omega >$): supvec {
    return $< \omega \star q$ , $J_w^{-1}(M+$GetTorque$(q) - \tilde{\omega} J_w \omega) >$;
}
```

where GetTorque(q) is the method that returns the torque for non-central forces, based on orientation q and the value of `dyna`'s `fList` attribute.

The `next_frame` method of the `dyna` class is called by the `dyna_system` from its `dynamics()` method:

```
dyna_system::dynamics() {
    for each dyna d do
        d.applycenterforce(d.mass()*gravity);
        d.next_frame();
        "call the callback that copies the motion state of d to
         the corresponding geometry in the host system"
    od
}
```

3.2.5 Summary

This completes the description of the forward dynamics subsystem of Dynamo. The main classes in the subsystem are shown in Figure 3.1 below. Detailed descriptions of these (and Dynamo's other) classes can be found in the user/reference manual which is available at [Baren 99].

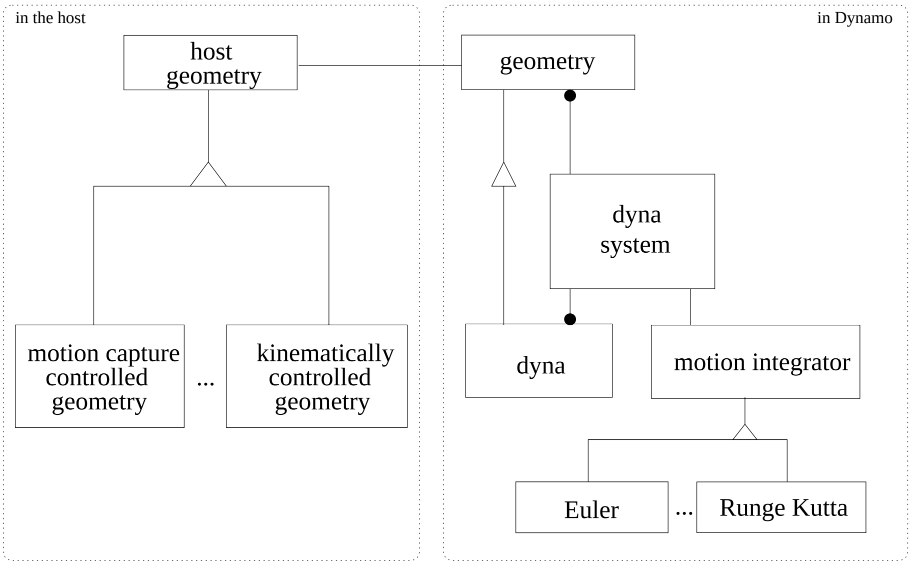


Figure 3.1: The major classes in the forward dynamics subsystem

Using the forward dynamics subsystem, we can have geometries in the host system which move under the influence of inertia and forces, torques and impulses that are applied to them. To accomplish this, a user of Dynamo calls the `dynamics` method of the `dyna_system` class each frame, interleaved with calls to the four `apply` methods of the `dyna` class to "push" the corresponding geometries in the host system, and influence their motion.

This provides geometries which move in a very natural way since their motions obey the laws of physics (within the accuracy of the motion integration process). It however does not provide a convenient way to exert control over the motions: steering a geometry only through forces which have to be provided explicitly is very cumbersome. The next section introduces controllers which can be used to facilitate such steering.

3.3 Controllers

The previous sections described how the forward dynamics problem can be solved. This resulted in the introduction of the `dyna` class, which models geometries that move under influence of forces, torques and impulses. Such dynas however are very passive: they need to be 'pushed' into motion. One way (which will be described in Section 3.4) is to connect such a dyna (using so-called *constraints*) to another geometry which does exhibit some active motion behavior (such a geometry could be kinematically controlled, or move according to some motion-captured data such as a the position of the computer's mouse).

Another way of making a dyna move is to have an entity that automatically calculates forces that are applied to it. Such an entity will be called a *controller*. A controller is often given a reference signal which specifies the desired motion. Its task is then to provide steering to the dyna such that it exhibits a behavior that corresponds to the reference signal (for more on the general theory of controllers see for example [Vegte 90]).

To this end, a controller measures properties of the dyna using a *sensor*. From the sensor-data, the controller calculates a steering signal. This steering signal is then transformed to the proper forces using an *actuator*. This is depicted in Figure 3.2 below. This figure shows a so-called closed-loop controller: the system to be controlled is monitored using a sensor. The difference between the measured data and the reference signal is used to calculate a steering signal, which is sent to an actuator. This actuator transforms this signal into forces,torques or impulses which are then applied to the system and –hopefully– will make the system behave more like prescribed by the reference signal.

This encapsulates one class of controllers. A simpler type such as an open loop controller is also useful in some cases. An open loop controller lacks the sensor depicted in Figure 3.2. Also much more complicated controllers can be envisioned, such as controllers that can control complete vehicles and steer them in such a way that such a vehicle will avoid obstacles (requiring long-term planning, and many more sensors and actuators). Such a more complex controller is still structured as in Figure 3.2, but in most cases the actuators are themselves again controllers. In this way, the steering value of the high-level controller functions as reference signal for a lower-level controller. So there is quite a diversity in controllers.

What all controllers have in common is that they need to update their steering signal(s) periodically (taking into account new sensor readings, if applicable): a controller needs to be

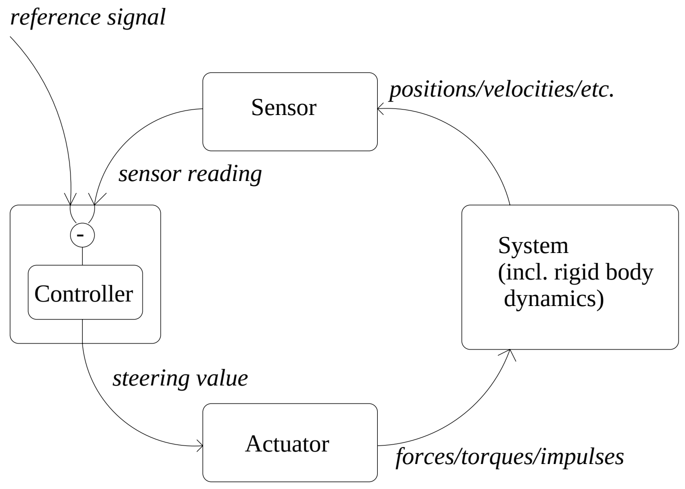


Figure 3.2: A controller controlling a system using a sensor and an actuator

triggered to calculate and apply its new steering value on a regular basis.

In the software, the `dyna_system` component is already triggered regularly to perform its calculations. It therefore makes sense to have the `dyna_system` trigger the software equivalents of the controllers: the objects from the `controller` class. This way, a host needn't do anything in addition to the call to `dynamics`, to make it appear that the controllers update autonomously.

To be able to loop over all controllers, the `dyna_system` object will need references to them. In a similar way that `geometry` and `dyna` objects register themselves with the `dyna_system`, `controller` objects do the same. To this end, the `controller` class is fitted with `activate` and `deactivate` methods. Having the registration of a `controller` not just in its constructor and destructor, means that a controller can temporarily be disabled, and later enabled, which can be useful in an animation.

Since a controller may have a sensor which monitors a non-dyna geometry, the `dyna_system` component also needs to keep the motion states of their companion objects up to date. The `dynamics` method now encompasses:

```
dyna_system::dynamics() {
    for each registered non-dyna geometry g do
        “call the callback that copies the motion state of g from
         the host system to its companion in Dynamo”
    od;
    for each controller c do  “trigger” c  od;
    for each dyna d do
        d.applycenterforce(d.mass()*gravity);
        d.next_frame();
        “call the callback that copies the motion state of d to
         the corresponding geometry in the host system”
    od
}
```

Specific examples of controllers are presented in Section 4.2.

Sensors

Closed loop controllers require a sensor to monitor the system under control. A sensor has to be able to take measurements from the system. So an object from the `sensor` class has a method `sense` which returns a new sensor reading. In the API given below, the return type of that method is `real`. It is conceivable that a sensor would measure properties with a more complicated structure, but usually a sensor performs some processing on its own and presents the controller with a single value. For example a distance sensor would measure the coordinates of two points in the system, but report only the distance between them. So for the sake of simplicity and efficiency, a single value is used. More sensors can be put in parallel if required of course. Here is the API of the `sensor` class:

```
class sensor {      // base class for sensors
  public
    real sense();   // perform a sensor reading and return the result
}
```

Some specific types of sensors are presented in Section 4.2.2.

Actuators

An actuator should translate a steering value (also assumed to be a single value, for similar reasons as sensors delivering a single value) into forces, torques and/or impulses. The engine and transmission of a car for example could be seen as an actuator which translates a single steering value (how far the gas pedal is pressed) to a torque on the wheel axis. Usually the steering value will be the magnitude of a force or torque or impulse of which the direction is already known, or can easily be determined by the actuator.

Often it is convenient if a limit can be put to the steering value. This models the maximum strength of an actuator. A `max_actuator` attribute is introduced against which a value that has to be applied is checked (provided the attribute has a nonzero value).

The resulting `actuator` API is:

```
class actuator {                      // base class for actuators
  public
    apply(real): void;                // apply a given steering value
    set_max_actuator(real): void;     // set the maximum strength
    get_max_actuator(): real;         // get the maximum strength
}
```

Examples of specific actuators are also presented later (in Section 4.2.2).

Using controllers to combine dynamics with other types of motion control

Controllers can enable combining dynamic motion control and other types of motion control more freely.

Take for example the motion of a trapeze artist in a circus. The swinging motion of the trapeze and the artist is clearly something to be modeled using dynamics. The artist drives the swinging motion by shifting his/her center of mass. This could be incorporated into the model by a control that varies the angle between the body of the artist and his legs. See Figure 3.3 below. Angle variations are a typical form of kinematic control. However, to be able to account for the shift in weight, both the upper and lower body need to be models by `dynas`. Neither can be modeled by a kinematically moved geometry.

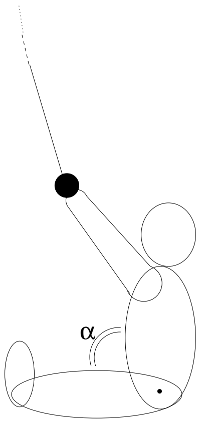


Figure 3.3: A course model of a trapeze artist

Therefore, direct application of kinematics is not possible. Instead, a controller can be used. It receives the kinematic angle as reference signal, and will provide the required torques to the upper and lower body. In this way a kinematic “signal” can be used to steer the motion of dynamically controlled geometries, resulting in a very kinematic-like motion (specification).

The reference signal used to mimic kinematic behavior in the example above can of course also be the result of inverse kinematic calculations, or another type of motion control, enabling the free combination/mimicking of those types of motion control as well.

Sometimes (for example in the GDP, see [Peet 95]), kinematic parameter changes, calculated by inverse kinematics routines, are not applied in full at once. This gives a sense of weight of the kinematic objects, since they seem to exhibit a sort of inertia in complying with the specified inverse kinematic goal. This slowness in response is easily modeled directly in the controller, since a controller usually also exhibits a “slow” response. A controller can not always exactly keep the sensor reading the same as the reference signal, but will require some time to compensate for disturbances in either the reference signal or the system that is under control. This is mainly because it takes time for the controller forces to take effect, and the controller can not always foresee all disturbances. Depending on the type of controller, this ‘slowness’ can be regulated through a controller parameter for dampening.

The slowness of a controller can be advantageous when using it to combine other types of motion control with dynamics. However, sometimes the control offered by a controller is not exact enough because of this slowness. In those cases, we will want to impose a very strict demand on the system. This can be done using the constraints discussed in the next section.

Summary

Figure 3.4 below shows an overview of the major classes in the controller subsystem of Dynamo.

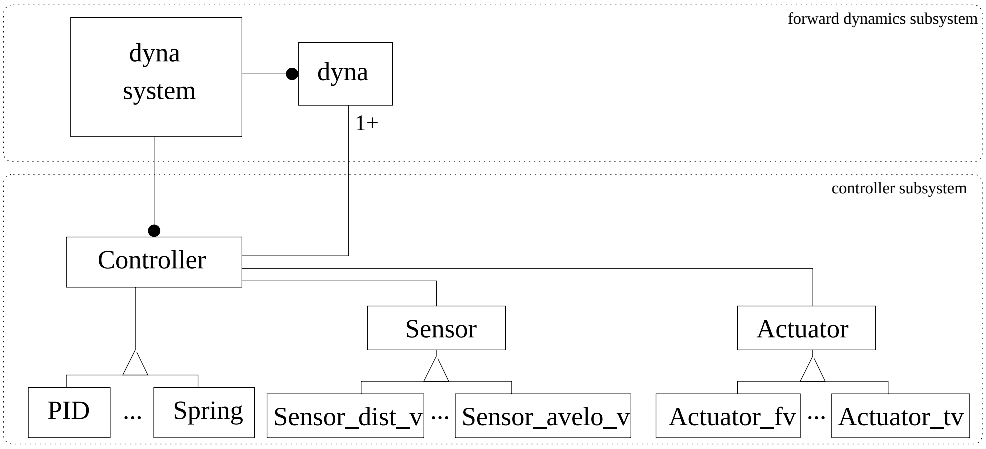


Figure 3.4: The major classes of Dynamo's controller subsystem

3.4 Inverse dynamics

3.4.1 Introduction

Using forward dynamics and controllers, one can obtain geometries that move according to the laws of physics in a controlled way. Much more interesting and useful motion behavior can however be obtained by not just using rigid geometries, but by using articulated rigid body systems: systems of rigid geometries that are connected to each other using all kinds of hinges and joints. Such hinges and joints could be modeled by designing controllers which try to keep connected geometries together, but then the "slowness" of controllers becomes a real burden. Very often such connections have to be very rigid: the relative motion of the connected geometries has to be constrained strictly. Such connections are also usually permanent, and most of the time do not need and external control signal like the controller in Figure 3.2.

This leads to a different approach: the approach using constraints. A constraint is a rule that the system has to abide by. It can be specified once, and then the system will have to keep making sure that it is not violated. The fact that it has to be specified only once makes constraints very easy to use, and facilitate the use of Dynamo considerably.

A lot of research in the field of constraint programming has been done and can be drawn upon. Appendix H gives an overview of the inverse dynamics problem in terms of constraint programming, and explains how it is solved in the terminology often used in the area of constraint satisfaction.

Constraints have to be enforced. Since the constraints at hand here all deal with the properties based on the motion state of a geometry, it can enforce its goal by steering those motion properties by applying forces, torques and/or impulses to the objects. Constraint satisfaction is therefore often also called inverse dynamics in this context, because a constraint calculates the forces, torques and/or impulses based on their desired effect.

Constraints have to be valid over a longer period of time. This yields a lot of similar constraint problems: one for each frame of the simulation. We can exploit this temporal coherence by not trying to solve for the constraint forces each frame from the start, but rather by formulating the problem as a problem of constraint correction. In the small amount of time between frames, a constraint will not have been violated much, and the situation will not be very different from the situation in previous frames. Therefore the forces calculated for the previous frame will probably be a good starting point in solving for the forces for this frame. Currently, linear extrapolation from the previous two values is employed in Dynamo.

Design decision 12: Constraints are satisfied using constraint correction

A constraint performs these calculations much like a controller does. Figure 3.5 shows how a constraint is constructed (it can be compared to Figure 3.2 on page 29 in which the controller structure is depicted). In order to make sure that the constraint is met at all times, it has to know the exact effect of the constraint forces. Therefore a constraint is constructed as a controller which can look one frame into the future. Whereas a controller goes through the loop *system→sensor→controller→actuator→system* once a frame, a constraint can repeatedly go through that loop and keep adjusting the steering value until a value is found such that

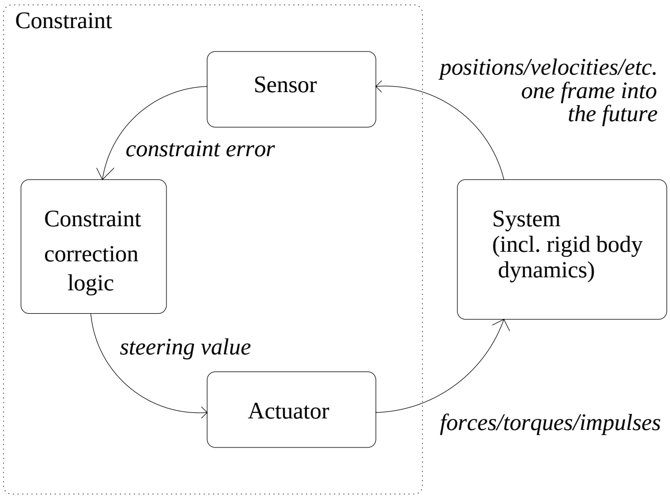

Constraint
Sensor
positions/velocities/etc. one frame into the future
constraint error
Constraint correction logic
System (incl. rigid body dynamics)
steering value
Actuator
forces/torques/impulses

Figure 3.5: A constraint imposed on a system

the constraint is met in the next frame. This makes a constraint much more computationally expensive than a controller.

The actuator and sensor of a constraint are often much akin. That is why it is chosen to not keep those separate from the constraint's control logic (or reuse the ones from the controllers), but to combine them (along with the constraint logic) into one class. This way, they can share each other's data and calculation results (often a direction used for and computed by the sensor is used by the actuator, and only has to be calculated once in this way).

> **Design decision 13:** For efficiency reasons, the sensor(s) and actuator(s) of a constraint are integrated into the constraint class.

When looking at constraints that specify connections (all types of hinges), a constraint acts between two geometries. It can restrict anywhere from zero to the full six degrees of relative motion freedom between the two geometries. A maximum of three of these are positional, while the (at most three) others deal with orientation. The following table shows some examples of specific constraints (these and more will be presented in detail in Section 4.1) as a function of the number and type of their restrictions.

	0 positional	1 positional	2 positional	3 positional
0 rotational	empty	point-to-surface	point-to-curve	point-to-point
1 rotational				
2 rotational		plane	cylinder	line-hinge
3 rotational	orientation		prism	connector

More types of constraints are of course possible: non-connection constraints such as the collision constraint, and constraints which act on more (or less) than two geometries such as the rope constraint. And there can be more types of a constraint which restrict the same degrees of motions: the point-to-point constraint is an example of a constraint which restricts three positional degrees of freedom, but so is the wheel constraint. And a constraint can also restrict the derivative of a position or orientation, either instead of the position and/or orientation, or in combination with them (non-holonomic constraints, see [Caban 68]). Before looking at these specific constraints however, we will concentrate on the general constraint correction problem that has to be solved.

3.4.2 The interfaces of the constraints and the constraint manager

Below, we will define the general interface of a constraint. Specific constraints can then adhere to this interface by inheriting from the general constraint class that specifies this interface.

> **Design decision 14:** An abstract constraint interface is defined to enable easy extension of the system with new constraints by specialization.

A constraint specifies some restriction on the (relative) motion states of dynas. It can be advantageous to be able to (temporarily) deactivate a constraint. We therefore introduce an `activate` and `deactivate` method. Another method `active` can be used to query the constraint to see if it is active or not.

Another useful concept is that of the *stiffness* of a constraint. The stiffness is a factor between zero and one which can be used to soften the constraint. This factor is applied to the changes of the forces that the constraint correction calculates. Constraint correction (see design decision 12) has the following form:

```
while simulating do    // animation loop
    while constraints at $t + h$ not sufficiently met do    // constraint correction loop
        Use the constraint error to calculate adjustments in steering values;
        Apply these adjustments
    od;
    Advance to the next frame
od
```

When using a *stiffness* factor, this factor is applied to the adjustment in steering values before it is applied, thereby slowing down the convergence of the constraint correction. In combination with a limited number of iterations for the correction, this causes constraints to exhibit a less rigid behavior.

A stiffness factor is also useful to increase the stability of the constraint correction when using correction methods such as relaxation (as presented in [Baren 94]). In the latter case, the stiffness factor is used to only partially correct a constraint each step, reducing the risk of overshoot. Overshoot can occur when the constraint is not exactly met after application of a force that was based on an imprecise estimate. Overshoot can impede the convergence of the constraint correction loop. If the estimation is really far off, overshoot can cause the constraint error to actually get worse. The stiffness factor can reduce too large a constraint force that causes overshoot to a constraint force that doesn't.

Next to the activeness and stiffness, specializations of the constraint class may have additional properties such as a maximum force and/or torque that can be exerted by such a constraint (at the price of breaking, see for example Section 4.1.2). But the external interface of the general constraint class looks like:

```
class constraint {              // class from which the different kinds of constraints
                                // are derived.
  public
    init(): void;               // initialize and activate the constraint.
                                // Depending on the type of constraint more
                                // information in the form of parameters to
                                // this method may be required.
    active(): boolean;          // is the constraint active at the moment
    activate(): void;           // activate the constraint
    deactivate(): void;         // de-activate the constraint
    stiffness(): real;          // get the stiffness of the constraint
    stiffness(s:real): void;    // set the stiffness of the constraint
}
```

This external interface will be extended with an internal interface which is used to perform the actual constraint correction. As discussed in [Kell 99], object orientation and constraint management require two interfaces for the objects that are being constrained: in the case of Dynamo, these are the external and internal interfaces.

Just as there is a global system object to manage all dynas in the forward dynamics subsystem, we introduce a class for one global object which will manage the constraints. This is the `constraint_manager` class. Like the dyna system object, this is a special one-of-a-kind[2] object that for each new frame checks and restores the constraints.

Constraints register themselves with the constraint manager in the same way as dynas register themselves with the dyna system (with the addition that constraints can remove and re-register themselves during their life-time to implement their `activate` and `deactivate` methods).

The constraint manager has a method `solve_constraints` which functions as an entry-point to the inverse dynamics system, in the same way that the `dynamics` method of the dyna system provides access to the full dynamics functionality. The `solve_constraints` method should be called each frame, and should ensure that the constraint correction is applied to all active constraints (i.e. those that are registered with the constraint manager at that moment). The dyna system can execute this call in its `dynamics` method, so the method can be part of the internal interface of the constraint manager class. This leads to a new version of the implementation of the `dynamics` method:

```
dyna_system::dynamics() {
    for each registered non-dyna geometry g do
        “call the callback that copies the motion state of g from
         the host system to its companion in Dynamo”
    od;
    for each controller c do  “trigger” c  od;
    for each dyna d do d.applycenterforce(d.mass()*gravity);
    constraint_manager.solve_constraints();
    for each dyna d do
        d.next_frame();
        “call the callback that copies the motion state of d to
         the corresponding geometry in the host system”
    od
}
```

The actual inverse dynamics algorithms will require the external interface of the constraint manager to be extended to offer the user control of the parameters of the constraint correction algorithm, but most of the additional methods which will be introduced below for the constraints, the constraint manager, and the dynas belong to Dynamo’s internal interfaces.

---

[2] more than one constraint manager could lead to problems since the sets of objects that are governed by the constraints of different constraint managers *have* to be disjoint. When a user tries to create a second constraint manager, he/she will be returned a reference to the first. The dyna system object is treated in a similar fashion.

Figure 3.6 below shows the major classes in the inverse dynamics subsystem of Dynamo (the specializations of the constraints are presented in Section 4.1). This overview is combined with the earlier overviews (for the forward dynamics and controller subsystems, Figures 3.1 and 3.4 in Appendix G).

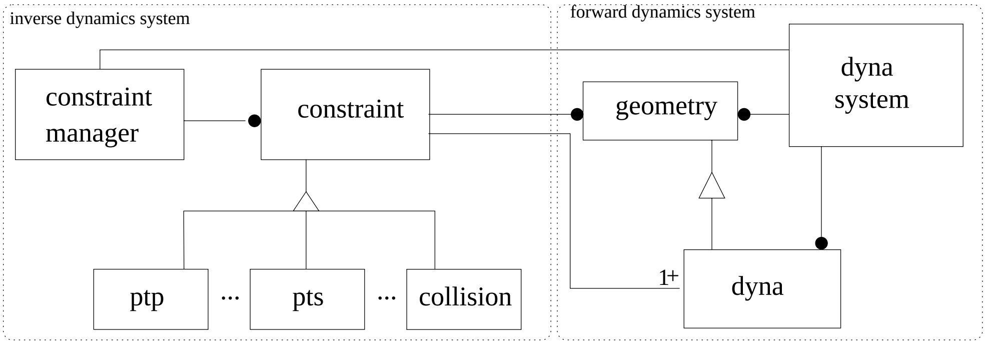


Figure 3.6: The major classes of Dynamo's inverse dynamics subsystem

3.4.3 The inverse dynamics algorithm: overview

Solving for the required constraint forces is a global process: the forces exerted by one constraint can influence the motion state of a dyna, and thereby influence the constraint error of another constraint. One way of handling this dependence is to look at the constraint equation of each constraint in isolation, and iteratively account for the dependencies between them, as was done in [Baren 94] using the relaxation algorithm. This approach however is very time-consuming for large chains of connected dynas since the constraint forces have to be propagated along the chain, and it therefore takes a high number of iterations to solve the problem.

Instead, we choose to combine all constraint equations into one system of equations, and solve that system, solving for all the constraint forces at once. This is done by the constraint manager.

Design decision 15: All constraint equations are combined to enable solving for all constraint forces at once.

A constraint uses its "sensor" to measure its constraint error. This is modeled by the constraint's *constraint function*, which returns the constraint error as a function of the properties of the geometries from which it takes measurements. The constraint function is zero when the constraint is met, and otherwise shows the magnitude and the direction of the error. The constraint correction algorithm should calculate new steering values which reduce the constraint error. We introduce the name *restriction* for such a steering value. These restriction values are then transformed to the forces, torques and/or impulses which restrict the dyna's motions using a constraint's *application function*. Knowing the application function of a constraint, the unknowns of the constraint correction problem are the restriction values.

Take for example the point-to-surface constraint discussed in full in Section 4.1.4. This constraint specifies that a point belonging to one geometry should always lie on a surface which is part of another geometry (see Figure 3.7 below). The point however can move freely in the surface. This free motion means that the point-to-surface constraint can only exert forces perpendicular to the surface. So the point-to-surface constraint is one-dimensional. There is one restriction value, which specifies the magnitude of a constraint force in the direction of the surface normal at the current position of the point in the surface. This force should pull the point and the surface together. The constraint error measures the (signed) distance from the point to the surface along the surface normal. The constraint error function of the constraint is therefore the function that returns this distance as a function of the 3D position of the point, the corresponding 3D position on the surface, and the surface normal at that point. The application function of the constraint translates the restriction value back to the 3D force vector using the surface normal.

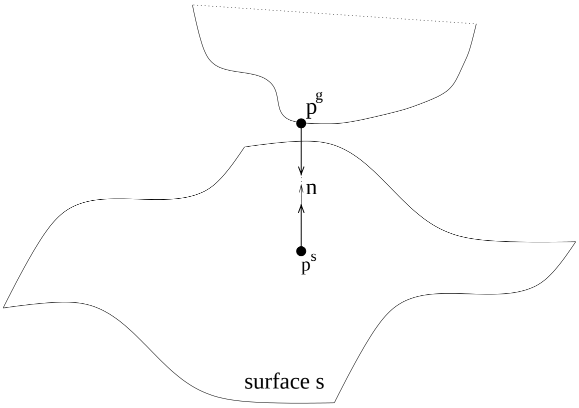


Figure 3.7: The point-to-surface constraint

The constraint manager combines all constraint errors in one large vector C. The goal of the constraint correction is to enforce $C = \vec{0}$. We can use Newton iteration (for a general explanation of Newton iteration see [Henr 64]; for the use in this case, see below) to accomplish this. To this end, the dependence of the constraint error on the restriction values R is linearized:

$$\Delta C = \frac{\partial C}{\partial R} \Delta R \tag{3.13}$$

Jacobian $\frac{\partial C}{\partial R}$ signifies in first order how the constraint error changes as the restriction values change. The user is responsible for simulating with a small enough step size that the higher order terms do not cause divergence (see design decision 5). In case of divergence, Dynamo simply does not apply the set of reaction forces that makes matters worse. Through the application function, such a change in restriction values results in changes in the constraint forces, which result –through the forward dynamics integration– in changes to the properties that are “measured” in the constraint error function, thereby changing the constraint error.

Assuming that we know the matrix $\frac{\partial C}{\partial R}$ (how it can be determined will be presented in Section 3.4.5), equation 3.13 allows us to calculate an approximation of the required change in restrictions. Since we have the constraint error functions, we can measure the current constraint error C. We want to nullify this constraint error, so we want to accomplish a change of constraint error $\Delta C = -C$. Using equation 3.13, we can calculate an approximation for the required change in restriction values by solving ΔR from:

$$-C = \frac{\partial C}{\partial R} \Delta R \tag{3.14}$$

Since this provides a solution for the linearized problem, we need to iteratively improve the R vector (using the scheme described above) to account for any non-linear dependencies between C and R (this is the Newton iteration process).

Two criteria are used to decide when to stop the constraint correction iteration: the iteration terminates when either a maximum number of iteration steps is reached or when the magnitude of the constraint error (measured globally over all constraints) is below a given threshold. To allow the user to control these algorithm parameters, methods are added to the external interface of the constraint manager to set and retrieve the maximum number of iterations and the maximum error. Setting the maximum error to a negative value enforces exactly the maximum number of iterations. Setting the maximum number of iterations very high should allow the algorithm to converge far enough for the final error to be smaller than the maximum error.

3.4.4 Algorithm details and incorporation in the classes

The constraint manager does all its calculations in terms of restriction vectors and error vectors. The error vector C is an aggregation of the individual error vectors of each constraint, and likewise the restriction vector R consists of the restriction values which are used to steer each individual constraint. A constraint has a *dimension*. This is the number of degrees of freedom the constraint restricts (as for example shown in table 3.4.1 on page 34). It equals the number of coordinates in the restriction vector of the constraint, since at least as many are needed to enforce restriction of this many degrees of freedom, while more would be redundant. It also equals the number of coordinates of the constraint error vector of the constraint. Again: at least as many are needed to provide enough information to calculate the restriction vectors, and more would be redundant. The sum of dimensions of each constraint is the dimension of the C and R vectors.

The internal interface between the constraint manager and the constraints is therefore in terms of these error and restriction vectors. This is depicted in Figure 3.8.

Each constraint has methods `applyrestrictions` and `applyrestrictionchanges`. These take a restriction vector and use the constraint's application function to convert the restriction values to the actual constraint forces. These are then applied to the dynas using the `applyforce`, `applycenterforce`, `applytorque` and/or `applyimpulse` methods.

A constraint also has a method `get_error` which returns the error vector of the constraint. This vector is calculated using the constraint's error function. The error is a function of the motion state of the geometries which are constrained by the constraint. To compute the

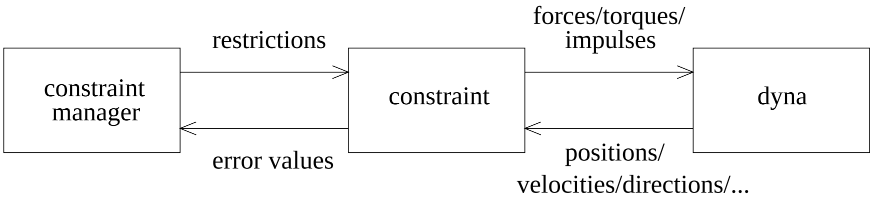


Figure 3.8: Inverse dynamics interfaces

constraint error, a constraint uses its ‘sensor’ function by calling the methods of the geometries which reveal its motion state (methods such as `get_next_position` and `get_next_velocity`).

In the constraint correction algorithm, matrix $\frac{\partial C}{\partial R}$ is used to estimate required changes in the restriction values. In each step of the iteration a change in restriction value is calculated based on the current $\frac{\partial C}{\partial R}$ and the current constraint error. So this iteration is a form of Newton-iteration. This means that a matrix $\frac{\widetilde{\partial C}}{\partial R}$ which looks sufficiently like $\frac{\partial C}{\partial R}$ will also cause fast convergence of the algorithm (making it pseudo-Newton). If we can save a lot of computational effort by making approximations in determining $\frac{\partial C}{\partial R}$ (determining $\frac{\widetilde{\partial C}}{\partial R}$ in effect), this might outweigh the cost of additional iterations because of slower convergence of the algorithm.

One application of this principle is the following. True Newton iteration requires re-evaluation of the derivative ($\frac{\partial C}{\partial R}$ in this case) after each iteration step: since R is modified, we are in a different point of the function that relates C and R. But since computing $\frac{\partial C}{\partial R}$ is quite costly, we choose to only calculate $\frac{\partial C}{\partial R}$ once a frame, and use that value as an approximation for all R-values encountered during that frame, on the basis that those will lie close to each other. This does require that $\frac{\partial C}{\partial R}$ is not zero in the neighborhood where $C = 0$, but this is the case for most constraints[3].

Therefore:

Design decision 16: Jacobian $\frac{\partial C}{\partial R}$ is only calculated once a frame.

It is even possible to only re-determine the matrix every few frames instead of every frame (unless of course sudden things happen like collisions, but in that case $\frac{\partial C}{\partial R}$ has to be recalculated anyway since the extra collision constraints mean a change in dimension of $\frac{\partial C}{\partial R}$). This has been implemented, but did not lead to any significant improvement (in most cases where the constraint forces vary slowly, a larger step size is a more effective means to increase the simulation speed).

The design decision is incorperated in the following algorithm (which is a refinement of the constraint correction loop shown on page 35):

---

[3] $\frac{\partial C}{\partial R}$ being zero might indicate a change in sign in $\frac{\partial C}{\partial R}$, which would mean that constraint error C might not properly indicate in which direction R would have to be changed. Take for example a constraint that is modified to return the (coordinate wise) square of its original constraint error. Such a constraint would always return a positive constraint error, so for the same $\frac{\partial C}{\partial R}$, the corresponding ΔR would always be in the same direction, even when –due to some overshoot– the value of R would pass the value for which $C = 0$.

```
constraint_manager::satisfy_constraints() {
    apply restrictions $R$ (to all constraints);
    calculate constraint error $C$ (from all constraints);
    calculate $\frac{\partial C}{\partial R}$;
    while $((|C| > \epsilon) \wedge$ (number of iterations$<$max_iter)) do
        solve $\Delta R$ from $-C = \frac{\partial C}{\partial R}\Delta R$;
        apply restriction changes $\Delta R$ (to all constraints);
        calculate constraint error $C$ (from all constraints);
    od
}
```

During this (pseudo-)Newton iteration, we have to account for the stiffness factor of each constraint (introduced in Section 3.4.2) which allows a user to soften a constraint. The stiffness value was introduced as a factor by which the calculated reaction forces were multiplied before they were applied to the dynas. In the approach proposed here, changing a reaction force after all reaction forces have been calculated undermines design decision 15, since diminishing one reaction force not only influences the constraint related to this force, but also other dependent constraints.

So we account for the stiffness in another manner. The goal of the stiffness factor is to only partially correct the constraint for this frame. So instead of trying to accomplish a ΔC of $-C$, we only try to correct for $\Delta C = -\textit{stiffness}\, C$. In order to accomplish this, we can modify the constraint equation of a constraint to report the smaller "error" *stiffness* C to the constraint manager during the pseudo-Newton iteration instead of C, causing the constraint manager to calculate forces to only correct this partial error.

The stiffness factor is mostly useful for the initialization of constraints which are not met entirely initially: a low stiffness value can in that case be used to gently converge to a state where the constraint is met (constraints can be used in this manner for auto-assembly of articulated bodies, as in [Barz 88]). In such a case, large forces are present which cause large changes in C and R, which weakens for example the basis for using pseudo-Newton instead of Newton iteration. The stiffness factor can be used to compensate. Since in each iteration step only factor *stiffness* of the remaining constraint error is corrected, the error will decrease roughly as a negative exponential.

For the implementation of the constraint correction we will need some auxiliary classes which implement the matrices and vectors (which can be of arbitrary dimension, but which dimension is known in advance). Some operations for handling (assigning and extracting) submatrices and subvectors will be useful in these classes, and of course methods for solving equations like $-C = \frac{\partial C}{\partial R}\Delta R$ are required. More details on the matter of solving this equation can be found in Section 3.4.6.

Finally, the constraints need an internal method which is called by the constraint manager at the very start of the inverse dynamics stage, in which the constraints determine and apply the first estimates for the restriction values. These first estimates stem from constant or linear extrapolation from the previous frame's values (the pseudo code for the `satisfy_constraints` method above uses constant extrapolation).

The remaining problem is determining $\frac{\partial C}{\partial R}$ at the start of each frame.

3.4.5 Determining $\frac{\partial \mathbf{C}}{\partial \mathbf{R}}$

At the start of each frame, the constraint manager needs to calculate the $\frac{\partial C}{\partial R}$ matrix which relates changes in the restriction vector to changes in the constraint errors. In order to do this, the constraint manager needs support from the constraints who influence this relation through their constraint error and application functions. Support is also needed from the dynas which need to predict the changes in the motion state resulting from changes in reaction forces. Two approaches are possible to obtain $\frac{\partial C}{\partial R}$:

1. An empirical approach: the constraint manager can observe the effects of changes to the restriction values by applying test changes in the value for each restriction of each constraint (which results in the application of test forces to the relevant dynas). Observing the effects of these tests on the constraint errors yields the relation between changes to the restriction values and the constraint errors.

   In this approach, the constraint manager maintains a black-box view of the constraints and their effects through the forward dynamics and tries to determine the relationship between ΔC and ΔR by simply trying a few restriction value changes and observing the effects on the constraint error.

2. An analytical approach: the equations which relate the constraint error to the restriction values are known: a change in restriction value is transformed to a force using a constraint's application function. This force is then transformed to a change in motion state using motion equations 3.3–3.8. This change in motion state is then transformed to a change in constraint error using the constraint error function. (Linear approximations of) the derivatives of these equations can be determined, and combined to arrive at $\frac{\partial C}{\partial R}$.

   In this approach the constraint solver has a white-box view of the constraint correction process and tries to determine how changes propagate through the constraints and the dynas (through the loop depicted in Figure 3.5).

Both approaches are discussed below.

Determining $\frac{\partial C}{\partial R}$ empirically

Using test forces to obtain $\frac{\partial C}{\partial R}$ requires little extra of the constraints next to the already existing `applyrestrictions` and `get_error` methods which can be used to apply the test forces and observe their effects. Since we do not want the effects of the test forces to show up in the next frame, a means is required to undo the effects of the test forces on the motion state of the dynas and on the internal state of the constraints: to this end the `constraint` and `dyna` classes are fitted with methods `begintest` and `endtest` which respectively save and restore the relevant parts of the state. Before testing, the `constraint_manager` calls the `begintest` method of each `constraint` (which causes the constraint to not only save its own state, but also call the `begintest` method of all `dynas` that it influences). After application of each test restriction and inspection of the changes in constraint errors, the `endtest` method of the corresponding `constraint` is called (which calls the `endtest` method of all corresponding `dynas`).

The $\frac{\partial C}{\partial R}$ matrix is constructed column by column: the ith column relates the effect of a change in reaction value i to all the resulting changes in constraint error. So in order to determine a column of $\frac{\partial C}{\partial R}$, the constraint manager can apply a test restriction vector which is only non-zero in the corresponding coordinate (we will from now on assume such a coordinate is set to 1), and then use the `get_error` method of all constraints to get the new error, which can be compared to the constraint error from before the test restriction vector was applied.

More accuracy is obtained if test forces $\frac{1}{2}r_t$ and $-\frac{1}{2}r_t$ are applied, and the difference in constraint error between the results of those changes are observed: $\dot{C}(R)$ is better approximated by $(C(R + \frac{1}{2}r_t) - C(R - \frac{1}{2}r_t))$ than by $C(R + r_t) - C(R)$, since the latter actually provides an approximation for $\dot{C}(R + \frac{1}{2}r_t)$, and it depends entirely on the scale of the reaction forces in the simulated system how close this derivative resembles the $\dot{C}(R)$. Whether the extra call to `applyrestrictions` (with resulting integration of the motion equations of several dynas) warrants the extra gain in precision remains to be seen from experiments (this issue is resolved in the last paragraph of Appendix B).

Applying test forces is quite expensive: for each constraint a number of test forces equaling the dimension of the constraint have to be applied to all the dynas involved in the constraint. Fortunately, several optimizations are possible:

When determining the effects of reaction forces on constraints, it can be seen that a given reaction force only affects the constraints which have a dyna in common with the constraint the reaction force stems from. This is illustrated in Figure 3.9 where four objects (A, B, C and D) are connected by three point-to-point constraints (constraint i connecting p and q, constraint j connecting r and s, and constraint k connecting t and u).

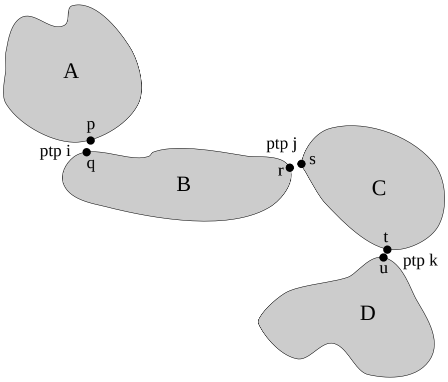


Figure 3.9: The influence of reaction forces on point-to-point constraints

In this case, reaction force f_i (resulting from restriction R_i) affects both constraint i and j, but not k: it causes movement of objects A and B only, so that only constraints pertaining to those objects are affected.

So we do not need to call the `get_error` method of all constraints, but only of the constraints that share at least one dyna with the constraint that the test restriction was applied to. This optimization has not been implemented since re-integration for the affected dynas' motion state is only done once anyway (see design decision 11), so the only savings would be in the evaluation of the constraint errors.

Another optimization might be to observe the effects of application of restriction values that have to be applied anyway, instead of using separate test restriction vectors. This does mean some extra processing since we lose the property that our test restriction vectors are orthogonal: for the matrix T_r of which the columns are the test restrictions, and the matrix T_c of which the columns are the resulting constraint changes, $\frac{\partial C}{\partial R}$ can be calculated from $T_c = \frac{\partial C}{\partial R} T_r$. Note that the required inverting of T_r requires that the test restriction vectors are independent, which might not always be the case for restriction values that are applied anyway.

This means that for an $n \times n$ matrix $\frac{\partial C}{\partial R}$ at least n motion integration steps are required to obtain the columns of T_c. The constraint correction algorithm (shown in the pseudo code on page 41) performs only one more motion integration step (as evidenced by the number of constraint error calculations) than the algorithm's number of iterations (which is usually small compared to n). Therefore, any gain from this optimization is probably offset by the required inversion of T_r, so, this “optimization” is not implemented. A possible optimization which might be useful would be if the observation of the effects of restriction vectors that are applied anyway were to be combined with replacing the (pseudo) Newton iteration with Broyden's method (see [Press 92]), but this is a subject for future research.

Determining $\frac{\partial C}{\partial R}$ analytically: decomposition

When we want to calculate $\frac{\partial C}{\partial R}$ (more specifically: $\frac{\partial C_{t+h}}{\partial R_t}$; subscripts denote the time stamps) analytically, we can observe that it is comprised of submatrices $\frac{\partial C^i_{t+h}}{\partial R^j_t}$ which each relate the change in constraint error of constraint i to changes in the restriction vector of constraint j. The constraint manager can traverse all constraint combinations and separately determine the $\frac{\partial C^i_{t+h}}{\partial R^j_t}$ submatrices, and then combine these to form $\frac{\partial C_{t+h}}{\partial R_t}$.

To determine such a submatrix $\frac{\partial C^i_{t+h}}{\partial R^j_t}$, we can look closer to how C^i_{t+h} depends on R^j_t: C^i_{t+h} is a function of the properties of the dynas that are under the influence of constraint i. These properties can be affected by the constraint forces which are exerted by constraint j. These constraint forces stem from restriction values R^j_t. Derivative $\frac{\partial C^i_{t+h}}{\partial R^j_t}$ is then found by determining the dependences in each of these intermediate steps and combining those.

To start the decomposition of $\frac{\partial C^i_{t+h}}{\partial R^j_t}$, we first take a look at the constraint error function C^i. For all time stamps τ, the constraint error is a function of the motion state properties of the dynas who's motion the constraint restricts. Such a property can be any of the properties which can be observed from a dyna through its methods (such as positions or velocities of points of the dyna, or the direction in world coordinates of a vector given in local coordinates, or any other property based on the dyna's motion state). We will denote the k-th property that constraint i depends on, by $\wp^{ik}_\tau$. This property depends on the motion state of geometry $\mathit{geom}(i,k)$ (geom is a constant function which tells to which geometry the k-th property of constraint i belongs).

Linearizing the relationship between C_τ^i and the $\wp_\tau^{ik}$ yields:

$$\Delta C_\tau^i = \sum_k \frac{\partial C_\tau^i}{\partial \wp_\tau^{ik}} \Delta \wp_\tau^{ik} \tag{3.15}$$

From 3.15, the calculation of the $\frac{\partial C_{t+h}^i}{\partial R_t^j}$ matrix can be split as follows:

$$\frac{\partial C_{t+h}^i}{\partial R_t^j} = \sum_k \frac{\partial C_{t+h}^i}{\partial \wp_{t+h}^{ik}} \frac{\partial \wp_{t+h}^{ik}}{\partial R_t^j} \tag{3.16}$$

which denotes that we determine the relationship between the constraint error of constraint i and the properties of the geometries, and we also determine the relationship between the properties of the geometries and the restrictions of constraint j. Combining these relationships yields the relationship between the constraint error of constraint i and the restrictions of constraint j.

An example: the point-to-surface constraint (see Section 4.1.4) has constraint equation $C = (p^g - p^s, n)$, so $\frac{\partial C}{\partial p^g} = n^T$ and $\frac{\partial C}{\partial p^s} = -n^T$. For such a constraint $\frac{\partial C_{t+h}^i}{\partial R_t^j} = n^T \frac{\partial p_{t+h}^g}{\partial R_t^j} - n^T \frac{\partial p_{t+h}^s}{\partial R_t^j}$

This accounts for the constraint error functions. and leaves us with $\frac{\partial \wp_{t+h}^{ik}}{\partial R_t^j}$. To further decompose this derivative, we can look at the application functions of the constraint, in particular to the application function of constraint j. We can observe the following:

Constraint j gives rise to forces, torques and/or impulses that are applied to the dyna(s) it controls. These steer the dynas in a way that its constraint equation is satisfied. To this end, it can use the `applyforce`, `applycenterforce`, `applytorque`, and `applyimpulse` methods of the dynas. These forces, torques and impulses are derived from the restriction vector of the constraint using the constraint's application function(s). Let Γ^{jl} denote the l-th force, torque or impulse of constraint j. Its dependence on R_τ^j can be linearized:

$$\Delta \Gamma_\tau^{jl} = \frac{\partial \Gamma_\tau^{jl}}{\partial R_\tau^j} \Delta R_\tau^j \tag{3.17}$$

Force, torque or impulse Γ_τ^{jl} is applied to dyna $dyn(j, l)$ (dyn is a constant function similar to the *geom* function introduced above, which tells to which dyna the l-th property of constraint j belongs).

Since restriction values are usually magnitudes of forces, torques or impulses, the columns of the $\frac{\partial \Gamma_\tau^{jl}}{\partial R_\tau^j}$ matrices are often the direction vectors for these forces (as often the $\frac{\partial C_\tau^i}{\partial \wp_\tau^{ik}}$ matrices from the constraint error functions consist of rows of the direction vectors along which the constraint error is measured).

Using equations 3.17, we decompose $\frac{\partial \wp^{ik}_{t+h}}{\partial R^j_t}$ as:

$$\frac{\partial \wp^{ik}_{t+h}}{\partial R^j_t} = \sum_l \frac{\partial \wp^{ik}_{t+h}}{\partial \Gamma^{jl}_t} \frac{\partial \Gamma^{jl}_t}{\partial R^j_\tau} \tag{3.18}$$

which denotes that we calculate the relation between changes in the property $\wp^{ik}_{t+h}$ and restriction vector R^j_t by combining the relations between $\wp^{ik}_{t+h}$ and the forces that are applied to the dynas which constraint i governs, and the relations between the forces that are applied to the dynas which constraint i governs and R^j_t.

An example: the point-to-surface constraint (see Section 4.1.4) has one reaction force which is directed along the surface normal, and is applied to p^g and (negated) to p^s. So in this case Γ^j_t denotes a force and $\Gamma^j_t = nR^j_t$ in this case yielding $\frac{\partial \wp^{ik}_{t+h}}{\partial R^j_t} = \frac{\partial \wp^{ik}_{t+h}}{\partial \Gamma^j_t} n$.

Combining the results from equations 3.16 and 3.18 yields:

$$\frac{\partial C^i_{t+h}}{\partial R^j_t} = \sum_k \sum_l \frac{\partial C^i_{t+h}}{\partial \wp^{ik}_{t+h}} \frac{\partial \wp^{ik}_{t+h}}{\partial \Gamma^{jl}_t} \frac{\partial \Gamma^{jl}_t}{\partial R^j_t} \tag{3.19}$$

Matrices $\frac{\partial C^i_{t+h}}{\partial \wp^{ik}_{t+h}}$ depend only on constraint i, and matrices $\frac{\partial \Gamma^{jl}_t}{\partial R^j_t}$ are related to constraint j only (as described above). Matrices $\frac{\partial \wp^{ik}_{t+h}}{\partial \Gamma^{jl}_t}$ tell how properties $\wp^{ik}_{t+h}$ depend on forces/torques/impulses Γ^{jl}_t. This matrix is only non-zero when $geom(i,k) = dyn(j,l)$ (as was described in the context of Figure 3.9), and is a linearization of the forward dynamics calculations:

$$\Delta \wp^{ik}_{t+h} = \frac{\partial \wp^{ik}_{t+h}}{\partial \Gamma^{jl}_t} \Delta \Gamma^{jl}_t \tag{3.20}$$

Property $\wp^{ik}_{t+h}$ depends on the motion state variables of $geom(i,k)$, and these in turn change when a force, torque or impulse is applied to $dyn(j,l)$, through the analytical integration of its position and linear velocity, and the numerical integration $\aleph$ of its orientation and angular velocity. A $\frac{\partial \wp^{ik}_{t+h}}{\partial \Gamma^{jl}_t}$ matrix linearizes these dependencies and depends only on the state of dyna $dyn(j,l)$. The exact formulae for this are presented in Appendix A, but the idea is the same as the decomposition presented above: decompose the partial derivatives as sums and products of other partial derivatives which are easier to determine. For example: the position of a point in a dyna depends on both the position of the dyna and on its orientation, so these motion state variables can be looked at separately and their dependencies combined later.

Determining $\frac{\partial C}{\partial R}$ analytically: incorporation in the classes

It is now clear that determining $\frac{\partial C^i}{\partial R^j}$ involves constraints i and j and all the dynas shared between these constraints. In the software, a $\frac{\partial C^i_{t+h}}{\partial \wp^{ik}_{t+h}}$ matrix can be calculated by `constraint` i, a $\frac{\partial \Gamma^{jl}_t}{\partial R^j_t}$ matrix by `constraint` j, and a $\frac{\partial \wp^{ik}_{t+h}}{\partial \Gamma^{jl}_t}$ matrix by the appropriate `dyna`. To determine how these classes interface with each other and the constraint manager, to combine their individual results, we observe the following:

Equation 3.19 can be written as:

$$\frac{\partial C^i_{t+h}}{\partial R^j_t} = \sum_l \left(\sum_k \frac{\partial C^i_{t+h}}{\partial \wp^{ik}} \frac{\partial \wp^{ik}}{\partial \Gamma^{jl}_t} \right) \frac{\partial \Gamma^{jl}_t}{\partial R^j_t} \tag{3.21}$$

or as

$$\frac{\partial C^i_{t+h}}{\partial R^j_t} = \sum_k \frac{\partial C^i_{t+h}}{\partial \wp^{ik}} \left(\sum_l \frac{\partial \wp^{ik}}{\partial \Gamma^{jl}_t} \frac{\partial \Gamma^{jl}_t}{\partial R^j_t} \right) \tag{3.22}$$

The first alternative shows how `constraint` j can provide the `constraint_manager` with the full $\frac{\partial C^i_{t+h}}{\partial R^j_t}$ matrix if `constraint` i can provide $\frac{\partial C^i_{t+h}}{\partial \Gamma^{jl}_t}$. `Constraint` i can calculate such a matrix using its own $\frac{\partial C^i_{t+h}}{\partial \wp^{ik}}$ matrices and calls to the proper `dyna`-methods that provide $\frac{\partial \wp^{ik}}{\partial \Gamma^{jl}_t}$. This is depicted in Figure 3.10 below. In this case the interface between `constraints` i and j would be in terms of the four possible Γ types: forces, central forces, torques and impulses.

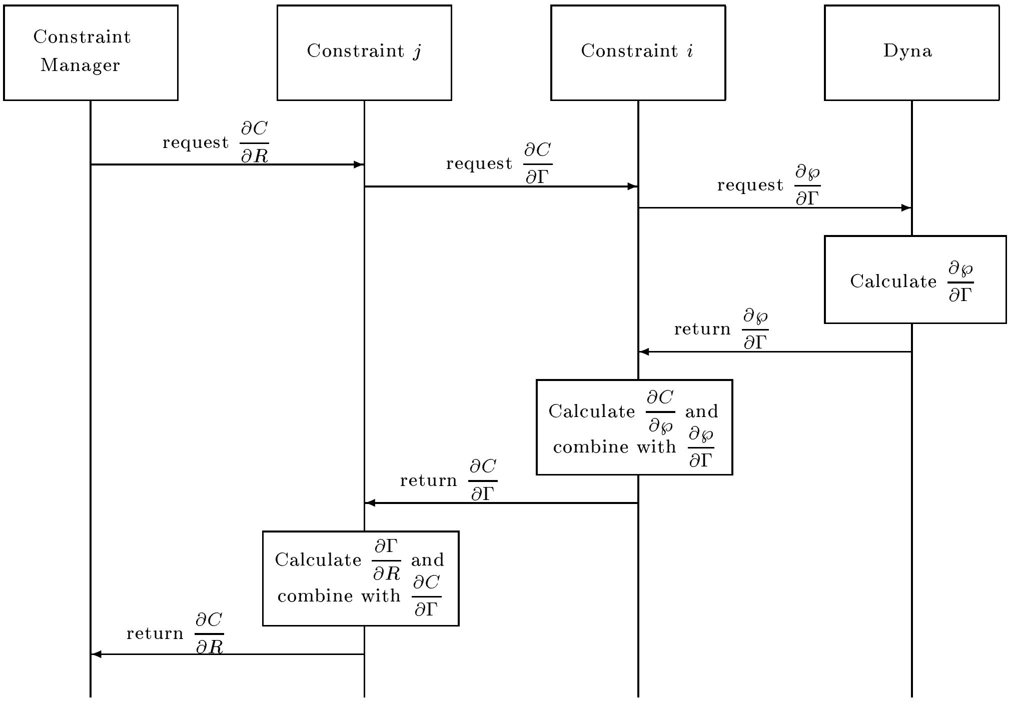

Constraint Manager
Constraint j
Constraint i
Dyna
request $\frac{\partial C}{\partial R}$
request $\frac{\partial C}{\partial \Gamma}$
request $\frac{\partial \wp}{\partial \Gamma}$
Calculate $\frac{\partial \wp}{\partial \Gamma}$
return $\frac{\partial \wp}{\partial \Gamma}$
Calculate $\frac{\partial C}{\partial \wp}$ and combine with $\frac{\partial \wp}{\partial \Gamma}$
return $\frac{\partial C}{\partial \Gamma}$
Calculate $\frac{\partial \Gamma}{\partial R}$ and combine with $\frac{\partial C}{\partial \Gamma}$
return $\frac{\partial C}{\partial R}$

Figure 3.10: Class interaction in determining $\frac{\partial C}{\partial R}$

The second alternative shows how the `constraint_manager` can ask `constraint` i for the complete $\frac{\partial C^i_{t+h}}{\partial R^j_t}$ matrix. `Constraint` i can calculate this matrix by multiplying its $\frac{\partial C^i_{t+h}}{\partial \wp^{ik}}$ matrices with the $\frac{\partial \wp^{ik}_{t+h}}{\partial R^j_t}$ matrices which have to be supplied by `constraint` j. `Constraint` j can calculate such matrices with the help of the `dyna` which provides $\frac{\partial \wp^{ik}}{\partial \Gamma^{jl}_t}$. This choice would mean that `constraint` j would have to have a method for calculating $\frac{\partial \wp^{ik}_{t+h}}{\partial R^j_t}$ for every possible kind of $\wp$ (properties of a dyna in which constraints are expressed).

The first alternative provides a smaller interface between constraints, and therefore less methods to implement. This interface is complete in the sense that it is unlikely that there will ever be another type of Γ^{jl}_t, while on the other hand it is very well possible that the motion state variables will be combined in new ways to properties $\wp^{ik}$. Choosing the first alternative ensures that existing constraint types will not have to be extended with new methods when a new property is added.

Because of these reasons, we decide:

> **Design decision 17:** Calculation of $\frac{\partial C}{\partial R}$ is decomposed according to equation 3.21 as opposed to equation 3.22.

As a result, each constraint is fitted with method (internal to Dynamo) `dCdR(constraint):matrix` (to be used by the `constraint_manager`), and methods (also internal to Dynamo) `dCdf(dyna,point):matrix`, `dCdF(dyna):matrix`, `dCdM(dyna):matrix`, and `dCdi(dyna,point):matrix` (for use between `constraints`). The `constraint_manager` calls method `dCdR` of `constraint` j, with `constraint` i as parameter. In this method, `constraint` j accounts for its own $\frac{\partial \Gamma^{jl}_t}{\partial R^j_t}$ matrices in combination with the results of calls to the appropriate `dCd`Γ methods (`dCdf`, `dCdF`, `dCdM` and/or `dCdi`) of `constraint` i. In each of these four methods, `constraint` i can immediately see if the force, torque, or impulse is applied to any `dyna` on which its constraint equation is based. For any such `dyna`, the appropriate method which calculates $\frac{\partial \wp^{ik}}{\partial \Gamma^{jl}_t}$ is then called, and `constraint` i's $\frac{\partial C^i_{t+h}}{\partial \Gamma^{jl}_t}$ is multiplied with the result.

In order to support the scheme suggested above, the `dyna` class will have to provide methods for all $\wp/\Gamma$ combinations. Derivations for the formulae for all of these combinations are presented in Appendix A.

Analytical $\frac{\partial C}{\partial R}$ determination can be sped up by letting the `constraint_manager` maintain a list of all `constraint` pairs which yield non-zero $\frac{\partial C^i}{\partial R^j}$ submatrices (i.e. the pairs of `constraints` which have at least one `dyna` in common). This list needs only be updated when constraints are added or removed, and in other cases having this list means that a lot of calls to the `constraint::dCdR` method can be avoided: usually the number of non-zero submatrices is linear in the number of constraints, and this compares favorably to the quadratic number of submatrices in $\frac{\partial C}{\partial R}$.

Another optimization can be derived from the observation that several `constraints` may invoke the inverse dynamics methods of a `dyna` object during the determination of $\frac{\partial C}{\partial R}$. There are a few expressions which are only dependent of the motion state variables, and not on the specific positions where a force or impulse is applied to, nor on specific properties (local coordinates of a point in the dyna etc.) which are combined with the motion state variables to arrive at a property $\wp^{ik}$. These expressions (see Appendix A) are $\frac{\partial A_{it+h}}{\partial \dot{\omega}_t}$, $\frac{\partial \dot{\omega}_t}{\partial M_t}$, and $(\tilde{\omega}_t \frac{\partial A_{it+h}}{\partial \dot{\omega}_t} - \tilde{A}_{it} h)$, and they only have to be calculated once and can then repeatedly be re-used between multiple calls to the inverse dynamics methods[4]. This significantly speeds up the process of calculating $\frac{\partial C}{\partial R}$. This optimization is very easy to implement within the object oriented structure of the software because the dyna class can easily store all the cached information (which is based on the local data of the dyna) in additional private attributes, and can thereby completely hide the optimization from the users of the class (since its interface is unaffected by this optimization).

Comparison between the two methods to calculate $\frac{\partial C}{\partial R}$

The two methods to obtain the $\frac{\partial C}{\partial R}$ matrix each have their own advantages. When only determining $\frac{\partial C}{\partial R}$ empirically, it is less work to add new constraint types since the only extra methods required are `begintest` and `endtest`, which are very easy to implement. Support for analytical $\frac{\partial C}{\partial R}$ determination requires implementation of the `dCdR`, `dCdf`, `dCdF`, `dCdM` and `dCdi` methods for each new type of constraint, which is a little more work. Most work in the analytical $\frac{\partial C}{\partial R}$ determination is however done by the methods of the `dyna` class, which only have to be implemented once (and have already been implemented for the cases where the properties of the dynas that are used in the constraints are positions and directions and their derivatives). The partial derivatives that have to be calculated by the `constraints` (for the constraint error and application functions) are usually very simple to determine and implement.

The caching that is used in the analytical $\frac{\partial C}{\partial R}$ determination looks like an advantage, but empirical $\frac{\partial C}{\partial R}$ determination uses a similar caching since after integrating a test force once, all dependencies on the corresponding restriction can be checked without the need of further integration (due to the administration described directly after design decision 11): a complete column of $\frac{\partial C}{\partial R}$ can be calculated at the cost of the integration corresponding to a change in one restriction value. This can be an advantage for the empirical method if the $\frac{\partial C}{\partial R}$ matrix is large and contains few zeros (i.e. the interconnectivity between the constraints is high).

The main advantages in terms of required calculational effort for the analytical alternative are that the analytical alternative does its calculations in terms of constraints, and not in terms of restriction values. Since each constraint contributes between one and six restriction values (see the table on page 34) this is advantageous for the analytical method. Another gain in speed for the analytical alternative over the empirical alternative is achieved by exploiting the symmetrical and anti-symmetrical structure of the matrices in the implementation of the support methods in the dyna class (see Appendix A).

Indeed, tests show that the analytical alternative is substantially faster. The best speedup

---

[4] Note that it is not beneficial to cache the three products $\frac{\partial A_{it+h}}{\partial M_t}$ since we would then have to calculate the sum in equation A.6 with full matrices each time, instead of with the anti-symmetrical matrices $\frac{\partial A_{it+h}}{\partial \dot{\omega}_t}$ from equation A.18

is obtained for a configuration with one constraint of the largest dimension available (at the time of these tests that was the point-to-point constraint of dimension three). In that configuration the speedup[5] of the analytical alternative over the empirical alternative is 10. A single point-to-curve constraint has dimension two and therefore offers a lower speedup of 7.5, while the one-dimensional point-to-surface constraint offers the even lower speedup of 5.0. A larger configuration consisting of three point-to-point constraints (connecting four dynas into a chain, yielding a 9×9 matrix where only two of the 3×3 submatrices are always zero) yields a speedup of almost six, and another larger configuration consisting of two pairs of point-to-curve constraints (yielding a $\frac{\partial C}{\partial R}$ matrix of 8×8 consisting of two 4×4 submatrices of non-zero elements and two 4×4 zero submatrices) yields a speedup of 4.5.

The analytical method also better supports parallelizing the computation of $\frac{\partial C}{\partial R}$ (should that ever be implemented): test forces cannot be applied in parallel since then their effects would mask each other. Caching aside, the submatrices of $\frac{\partial C}{\partial R}$ can all be calculated in parallel.

A drawback to the analytical alternative is that in differentiating constraint equations some effects are not taken into account: the shifting of the curve and surface parameters for the point-to-curve and point-to-surface constraints (see Sections 4.1.3 and 4.1.4) are examples of this. This hardly ever leads to real problems, and the slower convergence of using the less accurate $\frac{\partial C}{\partial R}$ is offset by the better performance of the analytical method.

The empirical approach is very useful for prototyping a new constraint type: it can first be tested with the empirical $\frac{\partial C}{\partial R}$ determination, and only later –when the constraint performs as it should– can the analytical support methods be implemented. This makes it easier to extend Dynamo (with newer constraint types).

Given these considerations and results, the analytical approach is preferred over the empirical one, but since it costs almost nothing extra to support the empirical method as well this method is included also, so a user can choose to use this approach (analytical $\frac{\partial C}{\partial R}$ determination is the default method used by the `constraint_manager` however). Methods are added to the external interface of the `constraint_manager` using which the user can switch between the two approaches.

3.4.6 Solving ΔR from $-C = \frac{\partial C}{\partial R}\Delta R$

Each frame, the constraints are iteratively corrected by solving ΔR from $-C = \frac{\partial C}{\partial R}\Delta R$ to arrive at adjustment ΔR for the restriction values based on the current constraint error C. For this, we need a suitable solver. In choosing this solver, we can consider the following:

1. The $\frac{\partial C}{\partial R}$ matrix is kept constant during all iterations of one frame (design decision 15). So the solver usually has to successively solve ΔR for different ΔC, but the same $\frac{\partial C}{\partial R}$. Speedup can be expected for solvers that can use this "asymmetry" by for example decomposing $\frac{\partial C}{\partial R}$ once and can then being able to solve for different ΔC very fast.

2. The $\frac{\partial C}{\partial R}$ matrix can be sparse. Especially for larger systems, using this sparseness can

---

[5] The speedups mentioned here are the speedups for the calculation of $\frac{\partial C}{\partial R}$ only. Since this calculation is only part of the total calculation, the overall speedup will be lower. Measurements showing performance evaluations of the overall process are presented in [Court 96]

lead to considerable speedup. In general the problem is of order $O(n^2)$ for an $n \times n$ $\frac{\partial C}{\partial R}$ matrix, but most constraint configurations give rise to $O(n)$ non-zero elements, so exploiting this feature reduces the time complexity of the solving process.

3. The solving process is part of an iterative process to take non-linearities into account. This means that very precise solutions are not required since the lesser significant digits in the answer of the linearized $\frac{\partial C}{\partial R}\Delta R = -C$ are probably not going to be of interest due to possible ignoring higher order terms.

The first consideration leads to decomposition solver methods such as LU decomposition. The $\frac{\partial C}{\partial R}$ matrix will only have to be decomposed once a frame, and during each successive solve step, only the much cheaper backward substitution process is required. LU decomposition can also be done more efficiently for bandwidth-limited matrices, so if the sparseness of $\frac{\partial C}{\partial R}$ can be made to take the form of $\frac{\partial C}{\partial R}$ having a small bandwidth b, the sparseness can be used as well (reducing the time complexity from $O(n^2)$ to $O(bn)$).

It is indeed possible to calculate an ordering of the constraints so that the corresponding $\frac{\partial C}{\partial R}$ matrix has minimal bandwidth. The algorithms that calculate an ordering that yields exactly the minimum bandwidth are of rather high complexity however (see [Tsang 93]), but an ordering that yields an approximate minimum bandwidth is easily found: such an ordering is the CutHillMcKee ordering presented in [Saad 95]. This ordering is the result of a specific breadth-first search traversal of the constraint graph, and is as such dependent on the starting node for the search. As a starting node, we use a node with the minimum number of neighbors in the graph. The constraints only have to be reordered when the structure of the constraint graph changes (when constraints are added or deleted). Having the `constraint_manager` order the constraints frees the Dynamo user from having to pay attention to the effect that the interconnectivity of the constraints and their ordering have on the performance. Dynamo will (approximately) optimize this performance.

A disadvantage of LU decomposition is that it does not handle overspecified and underspecified constraint configurations, since those lead to singular matrices. So it would be prudent to have an alternative that can handle such configurations. The third consideration above points to iterative solvers. An iterative solve method that only references $\frac{\partial C}{\partial R}$ through multiplication (and can therefore easily be optimized for sparseness, and has no a priori problems with singular matrices) is the conjugate gradient method. This method maintains the remaining constraint error $r = \frac{\partial C}{\partial R}\Delta R - C$, making $\frac{r}{C}$ an excellent value to base its stopping criterium on: the iteration within the conjugate gradient method can stop if $\frac{r}{C}$ falls below a given tolerance level. Tests have shown that a tolerance level of order as large as $0.01 \sim 0.1$ is good enough in the light of consideration 3 above.

Conjugate gradient solving can be applied in cases where there are one or more solutions to the constraint problem. So it is useful in cases where a unique solution exists, and for underconstrained systems. There is still a problem when conflicting constraints are encountered. A solver method which can also handle cases where no solutions exist is the SVD (singular value decomposition) method (which will give a solution which yields minimal error in least squares sense). SVD does take advantage of the asymmetry mentioned in the first consideration, but it can not be optimized for sparse matrices like the LU decomposition method can (in [Zlat 91] it is mentioned that the matrices which make up the SVD decomposition of a sparse matrix need not be sparse themselves). In general, this method is the slowest of the three methods described here.

Although the SVD method will always find a solution that minimizes the error, there is no guaranty that using this solver will not lead to divergent solutions to the original constraint problem: the solver is only used to solve the linearized version of the problem, so if the solution to this linearized version is critically balanced, very large reaction forces may result, which can not be integrated properly by the dynas, or at least have very large higher order effects rendering the linearization a very poor approximation.

Tests with these methods show that the LU decomposition method is clearly the faster method. Conjugate gradient usually runs at about 50%-80% of the speed of LU decomposition, and singular value decomposition is by nature already much slower, and can not use any banded structure of $\frac{\partial C}{\partial R}$. The conjugate gradient method however is more stable than LU decomposition when constraint configurations become (close to) being overspecified or underspecified, and the singular value decomposition is even more stable.

So there is no clear general best choice between the methods: there is always a compromise between efficiency and stability. Therefore the choice should be made at run time when the quality of the solutions can be judged for the problem at hand.

Design decision 18: All three solver methods are provided, and the `constraint_manager` switches between them as needed.

Switching between the methods is done by the `constraint_manager` (except in the case when LU decomposition detects a singular matrix: in that case the `matrix` class itself can decide to switch to conjugate gradient solving) since the `constraint_manager` can easily monitor what the effect of the solutions is on the constraint error, by comparing constraint errors in subsequent iterations of the global solution process. By default LU decomposition is used. Whenever a singular matrix is encountered or the solution obtained by this method only increases the constraint error, the solve method is switched to conjugate gradient. Whenever the results of conjugate gradient show divergence in the constraint error, the switch to SVD is made. It is less clear when to switch back to the faster (but less stable) LU decomposition. So we can just try it after a sufficient number (currently 10) of frames have been solved successfully with the current solver. Should the switch to the faster method have been unwarranted, the resulting divergence will make sure of a switch to the more stable (but slower) methods through the mechanism described above. This process is visualized in Figure 3.11 below.

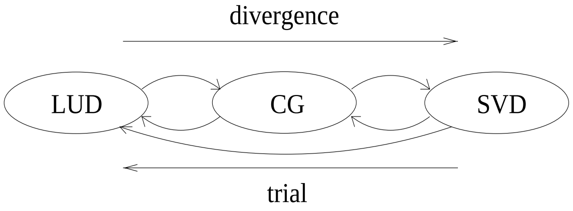


Figure 3.11: Switching between solve methods

The use of the conjugate gradient method imposes a restriction on the matrix: $\frac{\partial C}{\partial R}$ should be diagonally dominant (since conjugate gradient uses the diagonal of $\frac{\partial C}{\partial R}$ to form a preconditioner matrix, see [Press 92]) to increase numerical stability. This also benefits LU decomposition, since pivoting is not required in the case of diagonal dominance. The $\frac{\partial C}{\partial R}$ matrix is built up from submatrices which each indicate the influence of the restriction changes of one constraint on another's constraint error, so the submatrices on the diagonal correspond to the influence of a constraint's restriction values on its own constraint error.

To ensure diagonal dominance, we have to make sure that each of such submatrices are diagonally dominant: the i-th restriction value of a constraint should have as strong a correspondence as possible to the i-th constraint error value of that same type of constraint. This can usually be obtained by expression the constraint errors and restriction vectors in the same coordinate system, and by a proper relative ordering of the restriction values with respect to the error values. For for example an orthogonality constraint where the error consists of the inner product of two vectors, the corresponding restriction value should translate to a torque around a vector that is perpendicular to both vectors in the orthogonality constraint. This is seen in Section 4.1.8 in the correspondence between the coordinates of C in equation 4.1 and the columns in the matrix in equation 4.3 (it is also seen in case of the plane constraint in Section 4.1.10).

To allow the constraint manager to switch the solve method used by the matrix class which is used to represent $\frac{\partial C}{\partial R}$, the interface of this matrix class is extended with the following methods (next to the private methods that implement the actual solver algorithms):

1. `set_solve_method`: This method is to be used whenever the structure of $\frac{\partial C}{\partial R}$ has changed: based on which solver method is chosen, the method can either decide to use bandwidth-limited or regular LU decomposition (based on the bandwidth of the matrix), or sparse or regular conjugate gradient (based on the number of zeros in the matrix).

2. `prepare_for_solving`: This method has to be called whenever new values (within the above mentioned structure) have been provided for the matrix (i.e. once every frame usually). The method then prepares the matrix for solving, which means doing the LU decomposition if that was the method indicated in `set_solve_method`, or performing the singular value decomposition if that was the chosen solve method.

3. `solve`: This method does the actual solving: it uses backward substitution with the LU or singular value decomposition, or conjugate gradient if so indicated in `set_solve_method`.

The constraint manager is extended with a method that sorts the constraints to obtain a (near) minimal bandwidth (as described above). This method is called right before $\frac{\partial C}{\partial R}$'s `set_solve_method` is called.

To allow a user to also control which solver is used, the method `set_min_solve_method` is introduced: it indicates the solver which the constraint manager tries to switch back to after having solved a number of frames by a slow solver. So setting this to conjugate gradient, will cause the constraint manager never to consider LU decomposition, and setting this to singular value decomposition will ensure that only this solver is used. By default it is set to LU decomposition so that the fastest method can be selected. But if users know that there are overspecified constraints, they can indicate the conjugate gradient method to tell the constraint manager to avoid the overhead of trying (and failing) to use LU decomposition every once in a while.

3.4.7 Positional constraints issues

The discretization of the motion equations can give stability problems for constraints that are solely expressed in positional information of the dynas (such as the point-to-point and orientation constraints, see Sections 4.1.2 and 4.1.8). Because the constraints are only enforced at the timestamps t, $t+h$, $t+2h$ etcetera, the intermediate values for the constraint error can be non-zero if no restrictions to the derivative of the constraint are made. A high-frequency component (with a frequency corresponding to twice the step size h) in the reaction force does not show in the constraint error at the sampling time stamps, because the constraint error nicely crosses zero at those times. This is shown in Figure 3.12.

The point-to-point constraint (see Section 4.1.2) with its constraint equation $p_d - p_g = 0$ (with p_d the position of a vertex of a dyna and p_g the position of a vertex of a dyna or geometry) is such a constraint. In most cases, such a high-frequency component simply does not arise, or is harmless, but in some cases it's amplitude increases up to the point where the effective reaction force becomes insignificant with respect to this component, causing numerical instability.

Figure 3.12 below shows what can happen. At time t a constraint is met: $C_t = 0$. But it can still be the case that $\dot{C}_t \neq 0$. For the point-to-point constraint for example, this would mean that the two points coincide, but that they do have a significant relative velocity. This means that if nothing were done, the constraint would not be met the next frame. So a reaction force is calculated. This force will be of such magnitude that $|\dot{C}|$ is first decreased over the first half of the integration interval between t and $t+h$. Halfway the integration interval $\dot{C}$ will be zero, but C will not. The same reaction force will however keep changing $\dot{C}$, so that it will now grow opposite its original direction. This will reverse the relative motion between the two points of the point-to-point constraint, in effect reversing the growth in $|C|$. At time $t+h$ the constraint error will once again be (close to) zero, but its derivative will now be opposite to its value at the previous time stamp. In the next frame, a reaction force opposite to the one for this frame will have to be applied to once again reverse $\dot{C}$ to keep C at zero.

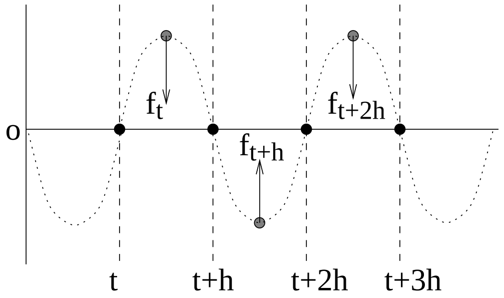


Figure 3.12: Constraint error in oscillation (but zero at all sample times)

If the constraint error depicted in Figure 3.12 is the distance between two point masses to which no external forces are applied, the piecewise constant constraint forces f_T will cause relative trajectories of the two point masses (between successive time stamps) that are exactly parabolae, because of the constant relative acceleration (between two successive time stamps). This oscillation is not dampened in any way, so numerical errors can lead this oscillation to either grow or decrease at will.

In a test-configuration consisting of a chain of ten cubes with the upper cube fixed in space (all connections via point-to-point constraints), and gravity causing the chain to swing (with one swing taking about four hundred frames), the component of the reaction force in the direction of the chain was measured. Figure 3.13 shows that component for the first 68 frames of the animation.

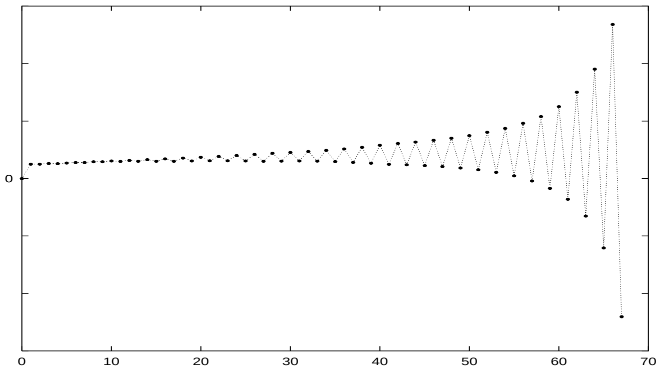


Figure 3.13: Oscillations in the reaction force magnitude for the first 68 frames of a solely positional constraint configuration

The high-frequency component is clearly visible and its amplitude increases exponentially in time. Indeed, the chain becomes instable shortly after the first seventy frames because the amplitude of the extra oscillation becomes so large that the magnitude of the average reaction force becomes insignificant with respect to the magnitude of the oscillations.

A solution to this problem is to also impose a restriction on the relative velocity of the vertices concerned, because the alternating relative velocity causes the large forces. Ideally, the continuous constraint $p_d - p_g = 0$ implies $\dot{p}_d - \dot{p}_g = 0$ (so that no non-holonomic constraints are required), but since time is discrete this is not the case, as was shown before.

The positional and velocity constraints should then be combined (since we only have one reaction force to regulate both), yielding for point-to-point the new constraint equation $p_d - p_g + \frac{h}{2}(\dot{p}_d - \dot{p}_g) = 0$. See Appendix C for the derivation of factor $\frac{h}{2}$. A constraint equation of this form is a little "softer" than the original equation, since it allows $p_d - p_g$ to be non-zero if there is a relative velocity that will soon bring p_d and p_g closer to each other. This can be unwanted. To reduce this effect, it is enough to only apply the extra velocity terms in the constraint error once in a while (say, every n frames).

These extra velocity terms show a duality with Baumgarte's approach to compensate for drift when constraining the first time derivative of a positional constraint. There, positional terms are added to compensate, much like velocity terms are added here to compensate for problems that arise with position-only constraints.

Figure 3.14 shows the result of a test with the modified constraint equation. The high-frequency oscillation is now gone, and only the normal slow increase of the varying constraint force (due to the gaining momentum and increasingly vertical orientation of the chain of cubes) remains.

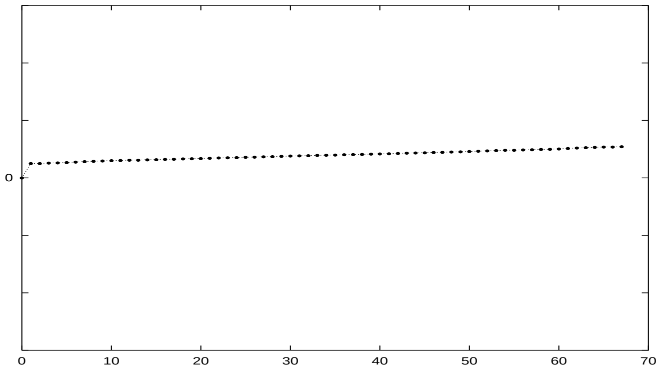


Figure 3.14: No unwanted oscillations if velocity terms are incorporated in the constraints

Other constraints suffering from the same problem can be modified in a similar way. The point-to-surface constraint (see Section 4.1.4) for example is modified from $(n, p^g - p^s) = 0$ to $(n, p^g - p^s + \frac{h}{2}(\dot{p}^g - \dot{p}^s)) = 0$.

The modification also helps to dampen large initial constraint forces in configurations where constraints are not entirely met initially.

3.5 Additional features

3.5.1 Visualizing forces

Since the main purpose of Dynamo is to calculate and handle forces (using the name “forces” here to indicate all “actuator types”: force, torques and impulses) for the purpose of animation, it is often convenient if these forces can be visualized in an easy manner. This is especially true for the reaction forces of constraints and controller forces. Studying these can give insight in the inner workings of a simulated system (see Figure 5.7 on page 92 for an example of an animation in which the constraint forces are visualized). This section deals with an extension of the Dynamo to facilitate the rendering of forces.

Since the actual rendering of forces deals with making the data structures which represent the forces (creating an arrow-shaped topology for a given force vector for example), and visualizing them on any cameras that are present, it is mainly the responsibility for the host system. But the dynamics library can provide an easy-to-use interface through which the information can be obtained about the forces that need to be visualized.

To this end we introduce two classes: the `DL_force_drawable` class and the `DL_force_drawer` class. Objects from the `DL_force_drawable` class are entities which have one or more forces to be visualized. The `force_drawer` manages these objects and takes care of the actual rendering:

the host system should provide an implementation for this class.

In Dynamo, the `DL_force_drawable` class functions as a base class for for example the `constraint` and `controller` classes and defines the interface using which the `force_drawer` can query the `force_drawables` about what forces to visualize. The `DL_force_drawer` class provides an interface for the `force_drawables` to register themselves so they can be managed. A `force_drawable` then has the following methods in its interface:

```
show_forces():void;
hide_forces():void;
get_fd_info(var nrf:integer, var tf:integer):void;
get_force_info(i:integer, var at:actuator_type,
               var d:DL_dyna*, var p:point, var f:vector);
```

The first two methods are used to (un)register the `force_drawable` with the `force_drawer`. The latter two are used only by the `force_drawer` to inquire about the forces that need to be visualized. The third method returns two integers indicating that this `force_drawable` has `tf` forces to visualize of which the first `nrf` are 'forces', while the remaining ones are reaction 'forces' (this distinction is made since the 'forces' of constraints and controllers usually come in pairs, and a user might not want to show both, since that information is basically redundant and the extra forces can cloud the display). The `force_drawer` can use the fourth method to inquire about each individual 'force' `i`. The method returns the type of force (an enumeration type indicating `none`, `force`, `torque` or `impulse`), its application point (the point with local coordinates `p` in `dyna d`), and finally the force vector itself. The latter two methods are implemented by each actual `force_drawable` within Dynamo.

Like the other "manager" objects in the system (the `dyna_system` and the `constraint_manager`), there is only one `force_drawer`. This uniqueness is accomplished by combining the `force_drawer` with the already-required component that implements the `dyna_system` callbacks (for exchanging motion state information between the host system and the companions in Dynamo). This way the `force_drawer` in the host system can be accessed through the `dyna_system` component in Dynamo.

The remainder of this section describes the global design of the `force_drawer` class as it can be used in the host system. This design is incorporated in the embedding of the Dynamo in the GDP (but since the implementation is GDP-specific, it can not directly be used in other host systems). Figure 3.15 shows the class relationships (the OMT notation is explained in Appendix F).

For each `force_drawable` which is registered, the `force_drawer` creates a companion object which is added to a list. By traversing the list of these `fd_elems`, the `force_drawer` can manage all the `force_drawables`. Each `fd_elem` object maintains a reference to its corresponding `force_drawable`, and takes care that for each force of that `force_drawable`, a suitable `fd_topo` element is created. Such an `fd_topo` element maintains the actual topological data structures which are used by the renderer to visualize the `fd_topo` element. The different types of 'forces' are rendered differently by each specific type of `fd_topo`, while the difference between forces and reaction forces is expressed in the color of the topology.

The `force_drawer` maintains the default values for each `topology`: the colors corresponding

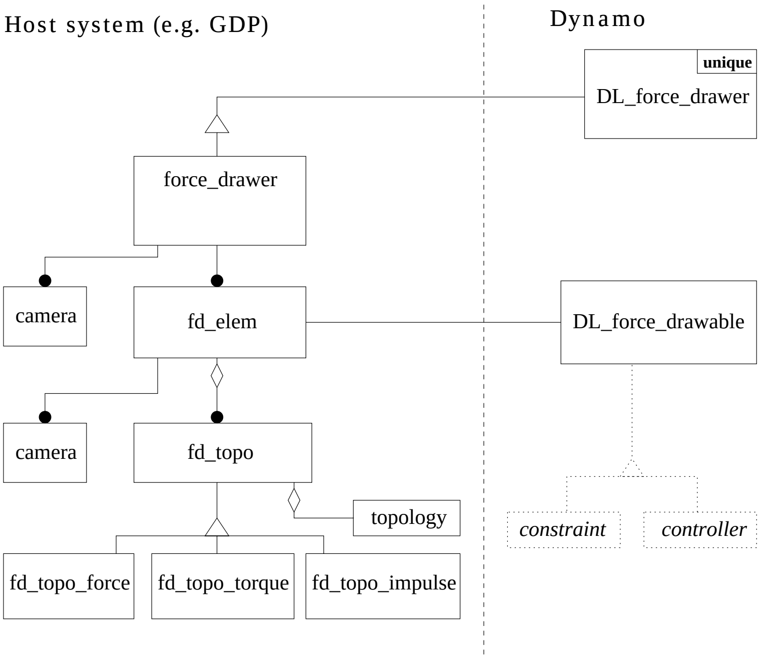


Figure 3.15: The classes used for drawing forces

to forces and reaction forces, a list of `cameras` on which the `topologies` are shown, whether to show forces and/or reaction forces, and the scaling factors using which the different types of actuators can be magnified to better show them. These default settings can be changed on a global level, and are given as parameters to the `fd_elems` upon creation where they can afterwards be changed individually.

Extensions to the system include methods in the `dyna_system` and `constraint_manager` to be able to show or hide the forces of all the `controllers` and `constraints` at once. A simple specialization of the `force_drawable` interface can also be added, in the form of a subclass of `force_drawable` which serves to store the `force_drawable` information required by the `force_drawer` to visualize a force. An object of this subclass can be used as a parameter of a new method of the `force_drawer` which will show the given force for one frame (relieving the user of such a method from administrating the storage of the required data). Such a method can for example be used by the `collision` constraint (see Section 4.1.14) which does not have the possibility to store the data itself for an entire frame because of its short existence (a collision constraint can be deleted before forces are visualised).

3.5.2 Friction

Introduction

In section 3.2, velocity damping was introduced as a very crude approximation of friction. In many situations, this may be accurate enough, but in other cases a more precise modeling of friction can be required. This section introduces a possible way of adding such a more refined friction model to the Dynamo. Due to time constraints, this model has however not been implemented, and more effort should be put towards shaping the theory presented here into a proper code design, so that friction can easily be added to all the contact constraints.

The Coulomb model of friction

Whereas velocity damping acts on the center of mass of a geometry, the Coulomb model (see pages 71-73 of [Boro 66]) assumes that the friction force f_F acts at a contact point between two geometries. In this contact point there is also a constraint force f_c which presses the two objects against each other. This constraint force induces a contact plane: the plane through the contact point, and perpendicular to the constraint force. The Coulomb model has two friction parameters μ_s and μ_d (the coefficients of static and dynamic friction respectively, where $\mu_s > \mu_d$ for known materials), corresponding to two types of friction:

Static friction where $|f_F| < \mu_s |f_c|$, and the friction force is large enough to prevent relative motion between the two geometries at the contact point.

Dynamic friction where $|f_F| = \mu_d |f_c|$ and the friction force can not prevent that the two object slide with respect to each other. The friction force is then directed opposite the sliding direction (in the contact plane).

Using the Coulomb model with the constraints

A friction constraint (which could act as a companion to a contact constraint) will have two “modes”, one modeling static friction, and the other modeling dynamic friction.

Dynamic friction is easily applied: given the contact force and the relative sliding direction, both the direction and the magnitude of the friction force are known, so the friction force can easily be calculated and applied. Since the friction force is directed opposite to the sliding motion, it will counter sliding velocity. There should be a switch to static friction whenever the friction force would be so big that the sliding velocity is reversed (easily determined by checking if the inner product between the sliding velocity vectors at successive time steps is equal or smaller than zero).

In static friction mode, the direction and magnitude of the friction force are unknown. It is however known that the effect of the force is zero sliding velocity. So this type of friction can be modeled by a sort of point-to-point constraint (which replaces the original contact constraint). This constraint is active as long as the inequality $|f_F| < \mu_s |f_c|$ holds. If the magnitude of the friction force would become too large, a switch to the dynamic friction mode should be made. In the case of rotational friction (in for instance a line hinge), the static friction constraint would take the shape of an orientation constraint.

The overall structure for modeling friction using the constraint mechanism is clear. There are however a lot of details that differ from contact constraint to contact constraint. So the incorporation of friction into the existing class hierarchy requires some additional work, and it might be the case that friction will have to be implemented separately for each of the contact constraints. It might however still be possible to specify a uniform “friction constraint” interface that allows handling some common elements the same way for all contact constraints. One of those common elements might be the administration of the friction constants. This constant can either be stored in the friction constraint, or determined from the two rigid bodies that have a contact. In the latter case, the geometry interface would be extended with the “material property” of the friction constants.

4 Specializations of the constraints and the controllers

In the previous chapter, the general design was presented. The subsystem for forward dynamics allows the use of geometries which move according to inertia and forces that are applied to them. Controllers can be used to apply forces to these dynas to make them move in a manner described by a reference signal that is provided to the controller. Constraints can be used to specify relations between the motions of such geometries that should always hold.

Classes were presented which specify abstract interfaces for constraints and controllers. Specific types of constraints and controllers should implement these interfaces in order to be useful in the schemes (presented in the previous chapter) for calculating controller and constraint forces. This chapter presents specializations for various types of constraints and controllers that can be used within these schemes.

4.1 Specific constraints

The general scheme for constraints requires for each specialization that a mapping from the restriction-values to actual reaction forces and torques is made (usually the restriction values are the magnitudes of forces and torques of which the direction is fixed in a local coordinate system of the constraint). This mapping is used in both in the `applyrestrictions` and `dCdR` method (the latter is for support of analytical determination of the $\frac{\partial C}{\partial R}$ matrix by the constraint manager; see Section 3.4.5).

Also, the constraint error vector has to be expressed in terms of the attributes (positions, velocities etc.) of the objects (dynas and geometries) that the constraint acts on. This information is then used for the implementation of the `get_error` and the `dCdΓ` methods.

For support of the empirical method of the constraint method to calculate the $\frac{\partial C}{\partial R}$ matrix, methods `begintest` and `endtest` should be implemented.

Furthermore, the exact parameters used to initialize (and therefore specify) the constraint should be decided upon. The initialization should also check that at least one of the objects that the constraint acts upon is a dyna.

Optionally extra features of the constraint (like maximum reaction forces etc.) can be decided upon. This section explores all these items for several specializations of the constraint class.

4.1.1 The empty constraint

The empty constraint is zero-dimensional, so no mapping between restrictions and reaction forces, or mapping between error vectors and dyna-attributes, or initialization parameters are required. This means that the signature of the empty constraint exactly equals the signature of the general constraint class. So it is possible to implement the otherwise abstract `constraint` class as the empty constraint. The constraint itself is not very useful, and should not be used, but implementing the constraint methods can be beneficial to the specializations.

The general constraint class can already declare the attributes in which the restriction vectors will be stored, and implement the `begintest` and `endtest` methods to save and restore these attributes. These latter only need to overriden by a specialization if it has extra attributes that need to be saved and restored, and even then the methods of the general constraint class can be used, so that only extra code has to be written for the extra attributes.

In a similar way, the extrapolation of the restriction vectors to arrive at the first estimate for the next frame can be implemented in the general constraint class so that this functionality is available to all constraint specializations.

The general constraint class can also offer services that are required by several specific constraints. For example: both the point-to-curve and the line-hinge constraints maintain a local coordinate system. Each frame, one of the base vectors of such a coordinate system is rotated and the other two have to be rotated along with it to maintain orthogonality (see the context of the footnote on page 65). This can be implemented in a private method of the general constraint class, so that all the descendants can use it.

4.1.2 The point-to-point constraint

A point-to-point constraint states that the position of a given point of a geometry should coincide with the position of another given point of another geometry. It thereby induces a ball-joint as shown in Figure 4.1. It has dimension three: one arbitrary-directed reaction force applied to both objects (in opposite directions) in this shared point is available to do the constraint correction.

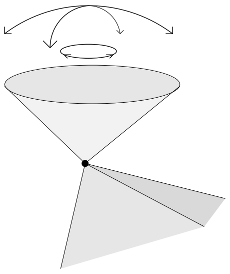


Figure 4.1: The point-to-point constraint yields a ball joint

Its constraint error is the difference vector between the positions of the two points concerned: its constraint equation is $C = p_d - p_g$ where p_d and p_g are the positions of the two points. Due to the issues discussed in Section 3.4.7, this constraint equation has to be modified to $C = p_d - p_g + \frac{h}{2}(\dot{p}_d - \dot{p}_g)$. The derivatives required for the `dCdR` methods can be derived from this equation: $\frac{\partial C}{\partial p_d} = 1$, $\frac{\partial C}{\partial p_g} = -1$, $\frac{\partial C}{\partial \dot{p}_d} = \frac{h}{2}$, and $\frac{\partial C}{\partial \dot{p}_g} = -\frac{h}{2}$

The mapping from the three translational restrictions to the reaction force is straightforward:

the three restrictions values are exactly the three coordinates of the reaction force (the constraint uses world coordinates as its local coordinate system). This means that $\frac{\partial f^{p_d}}{\partial R} = 1$ and $\frac{\partial f^{p_g}}{\partial R} = -1$ (since *action=–reaction* demands that $f^{p_g} = -f^{p_d}$)

The initialization method needs the specification of the two points in the form of a dyna and a position within that dyna, and a geometry (possibly a dyna) and a position within that geometry.

As an option, a maximum force magnitude attribute can be added. In previous work (see [Baren 94]), two possible courses of action were defined for when this maximum force magnitude is exceeded: the constraint either deactivates itself ("breaking the connection"), or only the maximum magnitude is applied. This last option is not compatible with the constraint satisfaction algorithms of Dynamo: the restriction values given to a `constraint` by the `constraint_manager` are part of a global solution to the global constraint problem. Changing the forces that are based on those values could invalidate other constraints (indeed: tests show that such behavior may even lead to instability). So in Dynamo only the first option is supported: if the maximum force magnitude is exceeded, the constraint will deactivate itself.

Checking against the maximum force magnitude is only done at the end of each frame: removing a constraint during the pseudo-Newton iteration would mean that the $\frac{\partial C}{\partial R}$ matrix would have to be changed during the iteration, which would be rather expensive in terms of computational effort. Also: the intermediate restriction values in the iteration on which such an intermediate deactivation would be based have no physical meaning: only the final restriction vector at the end of the iteration does.

A variant of the point-to-point constraint is a constraint which only states that the relative velocity of two vertices has to remain zero. This constraint makes sure that the difference in position between the two vertices remains constant. This constraint is called the *vtv* constraint (as opposed to the common *ptp* abbreviation of point-to-point). If the vertices initially have the same position, the `vtv` constraint is equivalent to a `ptp` constraint, barring numerical drift. Such numerical drift is often compensated for by methods such as those developed by Baumgarte ([Baum 72]) in other approaches to inverse dynamics which always differentiate the constraints.

4.1.3 The point-to-curve constraint

The point-to-curve constraint specifies that a given vertex p^g of a geometry should always lie on a given curve c (a continuous, differentiable function from $\Re$ to $\Re^3$). The vertex can move freely along the curve, but reaction forces perpendicular to the curve keep it from leaving the curve.

The free motion of the point along the curve means that the point-to-curve constraint should not exert any forces in the direction along the curve. Therefore, the point-to-curve constraint is two-dimensional. There is a reaction force that always lies in the plane perpendicular to the curve at the current position of the vertex. Two vectors x and y (see Figure 4.2 below) in this plane can be chosen to form an orthogonal basis together with the current first derivative d of the curve. There is a curve parameter s which administrates the current position on the curve, inducing point $p^c = c(s)$ on the curve. The two restrictions of the constraint are the magnitudes of the reaction forces in the directions x and y yielding:

$$f^{p^g} = \begin{pmatrix} \vdots & \vdots \\ x & y \\ \vdots & \vdots \end{pmatrix} R \quad \text{and} \quad f^{p^c} = -f^{p^g}$$

from which we can derive the derivatives for the `dCdR` method (the matrix in the equation above is a matrix which has vectors x and y as its two columns).

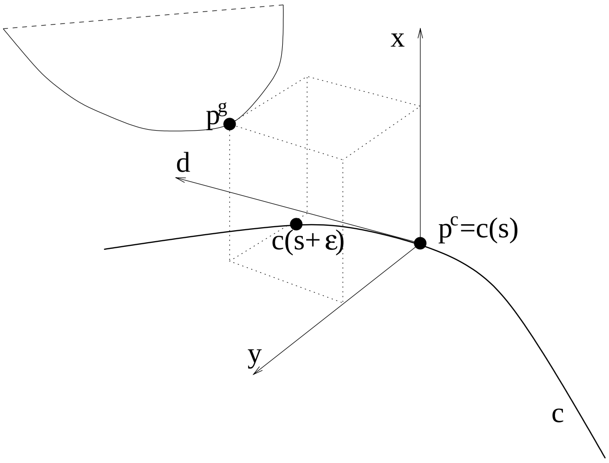


Figure 4.2: The point-to-curve constraint

The constraint error can be determined from the difference vector between p^g and p^c. As shown in Figure 4.2, this difference vector can be decomposed in the directions d, x and y. The components in the x and y directions are the components of the constraint error vector. So for the constraint equation we have:

$$C = \begin{pmatrix} \cdots x \cdots \\ \cdots y \cdots \end{pmatrix} (p^g - p^c)$$

In this formula, the matrix is the matrix that has vectors x and y as its two rows.

Due to the issues discussed in Section 3.4.7, this constraint equation has to be modified to:

$$C = \begin{pmatrix} \cdots x \cdots \\ \cdots y \cdots \end{pmatrix} (p^g - p^c + \frac{h}{2}(\dot{p}^g - \dot{p}^c))$$

From this equation, the $\dfrac{\partial C^i}{\partial \wp^{ik}}$ needed to implement the `dCdΓ` methods can easily be determined (for $\wp^{i0} = p^g$, $\wp^{i1} = p^c$, $\wp^{i2} = \dot{p}^g$, and $\wp^{i3} = \dot{p}^c$).

The component in the direction of d of the difference vector between p^g and p^c is used to determine the motion along the curve of p^c. In first order, we have $c(s + \epsilon) = c(s) + \epsilon\dot{c}(s)$. Since $d = \dot{c}(s)$, the component in the difference vector between p^g and p^c in the direction of d determines an appropriate value for an ϵ which can be added to s to move p^c along the curve. To account for higher order terms, the adjustment of s should be done iteratively.

The `get_error` method of the `ptc` class implements this adjustment procedure, since it already has to calculate the difference vector between p^g and p^c for its error calculation, and because it is called from within the loop of the `satisfy_constraints` method (see the pseudo code on page 41) which accounts for non-linearities.

In the methods `begintest` and `endtest` we now also have to save and restore the current curve parameter since determining the effect of a test restriction vector can change it.

As s is adjusted and the vertex moves along the curve, the derivative $\dot{c}(s)$ changes, causing a possible change of direction in d. To keep the local base orthogonal, vectors x and y will have to be rotated accordingly. This can be done at the start of each frame[1].

At initialization, the constraint requires a specification of the vertex (given by a geometry and the local coordinates of the vertex within that geometry) and the curve. Also, the initial position on the curve s_0 is required to initialize the p^c. To relieve a user from having to specify this initial curve parameter explicitly, an extra requirement can be added to the curve class (see below) specifying that it has a method which returns a value for the curve parameter which corresponds to a point of the curve that has minimal distance to a given vertex.

Currently, the constraint deactivates itself if the curve-parameter s moves out of the domain of the curve-function, but other reactions to such an event (e.g. changing the constraint to a point-to-point constraint which keeps the point to the end of the curve) might later be implemented.

In order to be able to implement the point-to-curve constraint for a variety of curves, a general abstract `curve` class is provided using which the `ptc` class can communicate with a variety of `curve` specializations that implement for example lines, Bezier curves etc. It has signature:

```
class curve {                 // virtual class from which to derive curves for the
  public                      // point-to-curve constraint
    geo():geometry;           // returns the geometry that this curve belongs to
                              // (i.e. the geometry that –if it is a dyna– any
                              // forces resulting from ptc constraints will be
                              // applied to).
    pos(s:real):point;        // returns c(s) expressed in the
                              // local coordinate system of geo
    deriv(s:real):point;      // returns ċ(s) expressed in the
                              // local coordinate system of geo
    indomain(s:real):boolean; // indicates if s is in the domain of the curve
    closeto(p:point):real;    // returns a curve parameter which corresponds to
                              // a point on the curve with minimal distance to p
    init(g:geometry):void;    // init with geometry g: a specific curve will
                              // probably have more parameters here
}
```

---

[1] if a is $\dot{c}(s)$ from the previous frame, and b is $\dot{c}(s)$ in this frame, we need a function that maps vector x to a rotated version Rx, where the rotation is one that satisfies $Ra = b$. Many such rotations are possible: we choose the one around $a \times b$. Vector x can then be expressed in basis $\{a, a \times b, a \times (a \times b)\}$. We know this basis rotates to $\{b, a \times b, b \times (a \times b)\}$ yielding $Rx = (x,a)b + (x, a \times b)a \times b + (x, a \times (a \times b))b \times (a \times b)$ for all x, assuming unit length for a and b

Note that it is not possible to change the `geometry` after the curve is initialized. Since the curve is given relative to this geometry, changing this attribute could suddenly move the curve radically which is not wanted since this would corrupt any active point-to-curve constraints. The curve becomes a special extension of the geometry (in case of for example Bezier curves, it is very probable that vertices of the geometry are used as control points).

Specializations of this `curve` class for lines, line-segments, circles, B-spline segments, B-splines, C-spline segments, and C-splines have been implemented, and more can be added in the future.

4.1.4 The point-to-surface constraint

Similar to the point-to-curve constraint, the point-to-surface constraint specifies that a point p^g should always lie on a surface s (a function from $\Re^2$ to $\Re^3$), where the point can freely move within the surface. Here, a reaction force perpendicular to the surface keeps the point from leaving it.

We now need two surface parameters u and v to administrate the current position $p^s = s(u,v)$ on the surface, and we now have two derivatives d_u and d_v (see Figure 4.3) along which motion in the surface can take place.

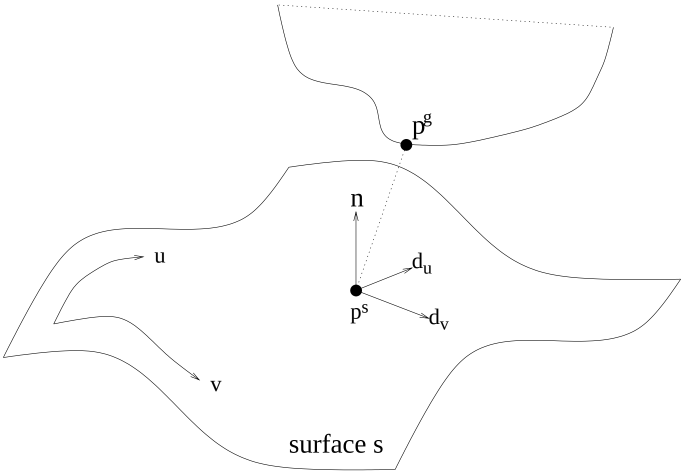


Figure 4.3: The point-to-surface constraint

The reaction force is directed along the local normal $n = d_u \times d_v$ of the surface. So we have $f^{p^g} = nR$ and $f^{p^s} = -nR$ to convert the one restriction value to the reaction forces (to be used in the implementation of `applyrestrictions` and `dCdR`).

For the constraint error, we have $C = (n, p^g - p^s)$ as the constraint equation (for use in `get_error` and `dCdΓ`), which we have to modify to $(n, p^g - p^s) + \frac{h}{2}(\dot{p}^g - \dot{p}^s))$ due to the issues discussed in Section 3.4.7.

Components of $p^g - p^s$ in the directions of d_u and d_v can be used to change the surface parameters u and v to reflect any motion in the surface.

As with the point-to-curve constraint, the surface parameters will have to be saved and restored in the `begintest` and `endtest` methods. If these parameters get out of the domain of the surface function, the constraint deactivates itself.

In order to be able to implement the point-to-surface constraint for a variety of surfaces, a general abstract `surface` class is provided from which specific surfaces (which for example implement planes, Bezier patches etc) can inherit. It has signature:

```
class surface {                        // virtual class from which to derive surfaces for the
  public                               // point-to-surface constraint
    geo():geometry;                    // get the geometry that this surface belongs to
                                       // (i.e. the geometry that –if it is a dyna– any
                                       // forces resulting from pts constraints will be
                                       // applied to).
    pos(u,v:real):point;               // returns $s(u,v)$ expressed in the local
                                       // coordinate system of geo
    deriv0(u,v:real):vector;           // returns $\frac{\partial s(u,v)}{\partial u}$ expressed in the local
                                       // coordinate system of geo
    deriv1(u,v:real): vector;          // returns $\frac{\partial s(u,v)}{\partial t}$ expressed in the local
                                       // coordinate system of geo
    indomain(u,v:real): boolean;       // indicates if (s,t) is in the
                                       // domain of the curve
    closeto(p:point):real×real;        // returns surface parameters which
                                       // corresponds to a point on the surface
                                       // with minimal distance to p
    init(g:geometry):void;             // init with geometry g: a specific surface
                                       // will probably have more parameters here
}
```

Again it is not possible to change the `geometry` attribute after the `surface` is initialized, as the `surface` is a special extension of the geometry (in case of for example Bezier patches, it is very probable that vertices of the `geometry` are used as control points).

Specializations for planes and ellipsoids have been implemented so far, but this surface library should be extended with more (complex) surfaces to make the `pts` constraint more useful in practice.

4.1.5 The line-hinge constraint

A line-hinge constraint specifies that two objects are connected through a line. The only relative motion possible is rotation along the hinge-line (see Figure 4.4 below). Two points (p^0 and p^1; each expressed in the local coordinates of both objects) on the hinge-line have to be chosen to which the reaction forces are applied[2]. In p^1, no force component in the direction of l is required, since a force directed along the line hinge applied in p^1 has exactly

---

[2] Note that we can also use *one* point to which a point-to-point-like reaction force is applied, in conjunction with a torque (without a component in the direction of the hinge-line). This turns out to be more expensive, so the alternative described in this section is used instead

the same translational and rotational effects as that same force applied in p^0 (and since the constraint restricts five degrees of freedom, we can only have five restriction values, and not the six that two full point-to-point constraints would yield). In order to direct the reaction force in p^1 such that it has no components in the hinge-direction, the line-hinge uses a local coordinate system for the directions of the reaction forces. The base vectors of this local coordinate system are the hinge-direction l and two (mutually orthogonal) directions x and y which are orthogonal to l (x, y and l are normalized so that restriction values are exactly the magnitudes of the reaction forces), as depicted in Figure 4.4.

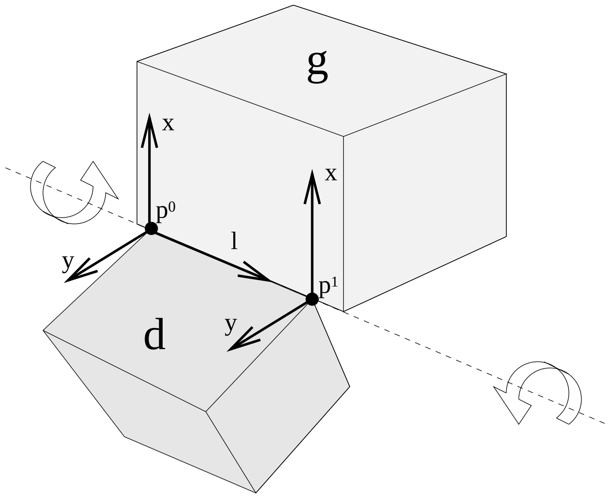


Figure 4.4: The reaction forces in a line-hinge connecting d and g

The line-hinge constraint has a force f^0 (acting on p^0 in both geometries) and a force f^1 (acting on p^1 in both geometries) at its disposal to enforce zero constraint error. For a restriction vector R, reaction forces f^0 and f^1 (applied to d, with their negations applied to g) are given by:

$$f^0 = xR_0 + yR_1 + lR_2 \quad \text{and} \quad f^1 = xR_3 + yR_4$$

or:

$$f^0 = \begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ x & y & l & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} R \quad \text{and} \quad f^1 = \begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & x & y \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} R$$

The constraint error is measured along the same directions and measures the difference in the positions of p^0 in d and in g (and likewise for p^1). More formally:

$$C = \begin{pmatrix} \cdots x \cdots \\ \cdots y \cdots \\ \cdots l \cdots \\ \cdots 0 \cdots \\ \cdots 0 \cdots \end{pmatrix} (p_d^0 - p_g^0) + \begin{pmatrix} \cdots 0 \cdots \\ \cdots 0 \cdots \\ \cdots 0 \cdots \\ \cdots x \cdots \\ \cdots y \cdots \end{pmatrix} (p_d^1 - p_g^1)$$

which –due to the issues discussed in Section 3.4.7– has to be modified to:

$$C = \begin{pmatrix} \cdots x \cdots \\ \cdots y \cdots \\ \cdots l \cdots \\ \cdots 0 \cdots \\ \cdots 0 \cdots \end{pmatrix} (p_d^0 - p_g^0 + \frac{h}{2}(\dot{p}_d^0 + \dot{p}_g^0)) + \begin{pmatrix} \cdots 0 \cdots \\ \cdots 0 \cdots \\ \cdots 0 \cdots \\ \cdots x \cdots \\ \cdots y \cdots \end{pmatrix} (p_d^1 - p_g^1 + \frac{h}{2}(\dot{p}_d^1 + \dot{p}_g^1))$$

The direction l has to be re-determined each frame as the difference between p^1 and p^0, and similar to the local coordinate systems of the point-to-curve constraint, x and y can then be rotated appropriately[3]. For initialization of the constraint, the two geometries (d and g) are required, and the local coordinates of p^0 and p^1 (in both those geometries). Like the point-to-point constraint, the line-hinge can be fitted with a `maxforce` attribute using which a user can specify a threshold that causes the hinge to "break" if it is exceeded. The magnitude of the reaction force is measured as the length of the five-dimensional restriction vector.

4.1.6 The cylinder constraint

A cylinder constraint allows rotation over a given axis, just like the line-hinge. The cylinder constraint however also allows for relative translation along that axis, as shown in Figure 4.5. It could be modeled by two point-to-curve constraints (with the curve being a line coinciding with the cylinder axis). A more efficient solution however is to use a line-hinge constraint of which the reaction force and constraint error components along the hinge-line have been removed. In that way we obtain the four-dimensional constraint with constraint equation:

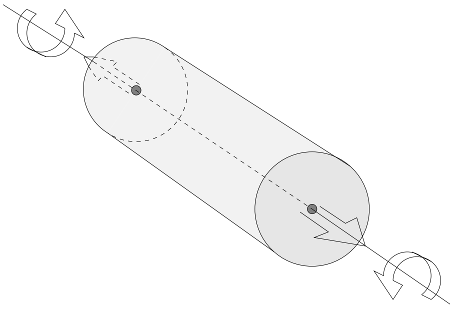


Figure 4.5: Relative motion for the cylinder constraint

$$C = \begin{pmatrix} \cdots x \cdots \\ \cdots y \cdots \\ \cdots 0 \cdots \\ \cdots 0 \cdots \end{pmatrix} (p_d^0 - p_g^0 + \frac{h}{2}(\dot{p}_d^0 + \dot{p}_g^0)) + \begin{pmatrix} \cdots 0 \cdots \\ \cdots 0 \cdots \\ \cdots x \cdots \\ \cdots y \cdots \end{pmatrix} (p_d^1 - p_g^1 + \frac{h}{2}(\dot{p}_d^1 + \dot{p}_g^1))$$

---

[3] When determining l, normally it does not matter if $p_d^1 - p_d^0$ is taken or $p_g^1 - p_g^0$ since they are used as directions for forces at time stamp t (at which all constraints are valid due to the constraint correction at the previous time stamp).

and with reaction forces:

$$f^{p^0} = \begin{pmatrix} \vdots & \vdots & \vdots & \vdots \\ x & y & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} R \quad \text{and} \quad f^{p^1} = \begin{pmatrix} \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & x & y \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} R$$

which is otherwise the same as the line-hinge.

4.1.7 The connector constraint

The connector constraint specifies that two geometries have no freedom left in their relative motion. The advantage of using a constraint to accomplish this, compared to just modeling the two geometries as one, is that the connection can be temporary in this way and it can break under too great a stress using a `maxforce` attribute as before. Also: it can be used as an aid to bring objects together (when initializing the constraint in such a way that the constraint is initially not met, and that the required relative position and orientation are reached when the constraint is met).

The line-hinge constraint presented earlier can be generalized to a connector constraint by adding a third point as depicted in Figure 4.6. This however would give rise to three points in each of the geometries in the constraint equation, and to three different reaction forces, i.e. we would need eighteen terms for the $\frac{\partial C}{\partial R}$ derivative in equation 3.19. A combination of a point-to-point and an orientation constraint (discussed in the next section) yields twelve terms for the $\frac{\partial C}{\partial R}$ derivative and will therefore cost less computational effort to calculate.

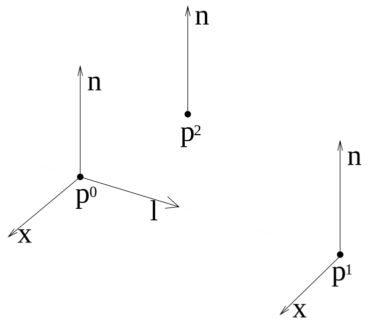


Figure 4.6: Possible reaction forces for a connector constraint

Using a point-to-point/orientation combination yields a connector constraint which has as attributes the point-to-point constraint and the orientation constraint. It only functions as a link between the two: the method for for example setting the `stiffness` attribute of the connector constraint makes sure that the `stiffness` attributes for both constraints are set, etc. Also, if the point-to-point constraint deactivates itself because its maximum force is exceeded, the connector constraint deactivates the orientation constraint as well (and vice versa). In this way the connector constraint has both a `maxforce` and a `maxtorque` attribute, and the constraint deactivates itself forces or torques are encountered which exceed the value stored in either of these attributes.

4.1.8 The orientation constraint

The full orientation constraint specifies that two geometries should have the same orientation (regardless of their relative position). The constraint is three dimensional and it accomplishes its goal using a reaction torque which is applied to both objects (inverted to one of the two of course to abide the *action=–reaction* principle).

In both geometries, three orthogonal directions w, x and y (given in the local coordinate system of the two objects d and g) are chosen (see Figure 4.7 below) in such a way that the correct relative orientation is reached if these directions align when expressed in world-coordinates. For the constraint error to be a measure for the rotation angles that are required to align the vectors, we can express the constraint error in terms of the mutual orthogonality of the three vectors, where the directions in d and in g should be exchangeable if the constraint is met: inner product (x_d, y_g) provides information about the required rotation along w, (y_d, w_g) provides information about the required rotation along x, and (w_d, x_g) provides information about the required rotation along y.This way, the constraint error provides the information about the rotation angles that the constraint torque should accomplish. Using a little more asymmetric approach, we can limit the number of directions required from the two objects from six to four by using as constraint vector error:

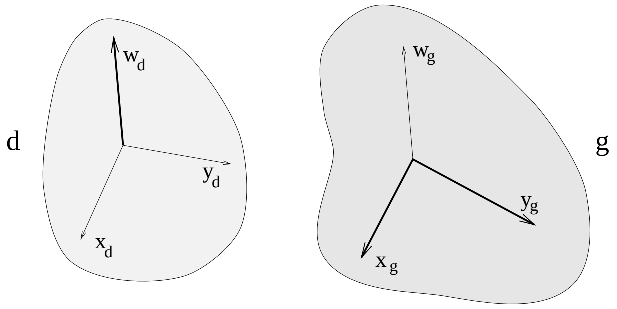


Figure 4.7: For the orientation constraint vector w_d has to be perpendicular to x_g and y_g

$$C = \begin{pmatrix} (x_d, y_g) \\ (w_d, y_g) \\ (w_d, x_g) \end{pmatrix} \tag{4.1}$$

Using this constraint error still means that for an evaluation of the constraint error we need four directions (two from each object), instead of only two positions (one from each object) as for for example a point-to-point constraint: orientational restrictions are more expensive than translational ones. Just like for the other constraints, velocity-terms have to be added to avoid any unwanted high frequency components (as described in Section 3.4.7). So we arrive at:

$$C = \begin{pmatrix} (x_d + \frac{h}{2}\dot{x}_d, y_g + \frac{h}{2}\dot{y}_g) \\ (w_d + \frac{h}{2}\dot{w}_d, y_g + \frac{h}{2}\dot{y}_g) \\ (w_d + \frac{h}{2}\dot{w}_d, x_g + \frac{h}{2}\dot{x}_g) \end{pmatrix} \tag{4.2}$$

The torque that is used to accomplish the constraint correction could simply be expressed as $M = R$ (like the reaction force of a point-to-point constraint). However, because of the required diagonal dominance (discussed in Section 3.4.6 on page 53), the torque is also expressed in the local coordinate system:

$$M = \begin{pmatrix} \vdots & \vdots & \vdots \\ w & x & y \\ \vdots & \vdots & \vdots \end{pmatrix} R \tag{4.3}$$

A maximum torque attribute can be used to set a boundary for when the constraint should deactivate itself.

Since the constraint manager can iterate to improve the estimates for the restriction values until the constraint errors are below a given value, the magnitudes of constraint errors should be a rough guide to how well constraints are met. To ensure that the magnitude of the orientation constraint error matches that of positional constraints (where the errors specify distances instead of the inner products used here), we have to scale the vectors w, x and y appropriately. The vectors are each scaled to length $\frac{\sqrt{det(J)}}{m}$ where J and m are the tensor of inertia and the total mass of the dyna that is controlled by the constraint. This ensures that constraints errors for the orientation constraint are of the same order as the errors for other constraints. The `dyna` class is fitted with a method that returns the required factor.

4.1.9 The prism constraint

The prism constraint only allows translation along a given axis between the two geometries that it constrains (the relative motion is shown in Figure 4.8). Since a prism joint only allows translation along a given axis, the orientation of the prism is completely fixed with respect to the attached object. So, like the connector constraint, the prism constraint can just be fitted with an orientation constraint as an attribute which ensures that the relative orientation remains constant. Of the remaining three translational degrees of freedom, two have to be constrained: just as with the cylinder constraint, two vectors x and y, perpendicular to the prism-direction, can be taken, yielding a partial point-to-point constraint that can prohibit relative translation in those directions.

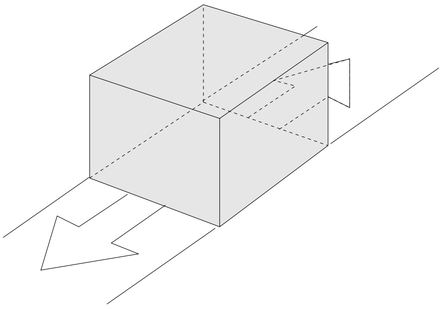


Figure 4.8: The relative motion for the prism constraint

So the remaining constraint is only two-dimensional and has as constraint equation:

$$C = \begin{pmatrix} \cdots x \cdots \\ \cdots y \cdots \end{pmatrix} (p_d - p_g + \frac{h}{2}(\dot{p}_d + \dot{p}_g))$$

(where p is the point of the prism axis, and x and y are the two vectors that are perpendicular to the direction of the prism axis and each other).

Th reaction force in p is then the force perpendicular to the prism direction:

$$f^p = \begin{pmatrix} \vdots & \vdots \\ x & y \\ \vdots & \vdots \end{pmatrix} R$$

Like the connector constraint, the prism constraint has both a maximum torque and a maximum force attribute, and the constraint deactivates itself if either is exceed.

4.1.10 The plane constraint

A plane constraint specifies that two planes –each in a different geometries– should always coincide. In each geometry g, a plane is given by providing point p_g, and normal n_g (both in local coordinates of g). Figure 4.9 below shows this for cube q and larger geometry p.

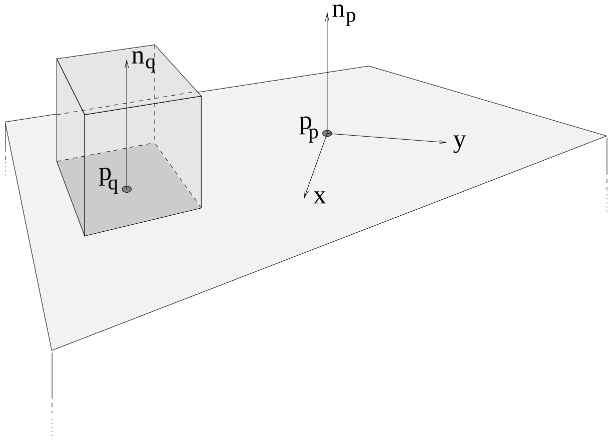


Figure 4.9: A plane constraint between geometries p and q

The constraint specifies that the two planes have to coincide. That means that the relative motion between the two geometries is restricted to translation in the plane, and rotation around the plane's normal. So the constraint has to restrict one translational degree of freedom (translation along the plane's normal), and two rotational degrees of freedom. In order to do so, two vectors x and y perpendicular to the plane's normal are chosen in one of the geometries, allowing the following formulation of the constraint error (when the vectors are transformed to world coordinates): $C = \begin{pmatrix} [p_p - p_q]_n \\ (n_q, y) \\ (n_q, x) \end{pmatrix}$.
(here notation $[v]_w$ denotes the component of vector v in the direction of w)

The constraint can correct for errors using a force in the direction of the plane's normal (applied to point p)[4]: $f_p = nR_0$, and using torques around axis x and y: $M = xR_1 +$

---

[4] a central force would also suffice to correct translational errors, but applying the force in point p allows

yR_2 (notice the exchange of vectors x and y between the application function and the error function: this keeps the corresponding submatrix of $\frac{\partial C}{\partial R}$ diagonally dominant, which is needed as was explained in Section 3.4.6 on page 53).

Like other constraints, the plane constraint also has to be augmented with velocity terms, and can be fitted with a `maxforce` attribute and a `maxtorque` attribute which can be used to set a maximum strength for the constraint.

4.1.11 The bar constraint

A bar constraint imposes a distance constraint between two points of geometries.

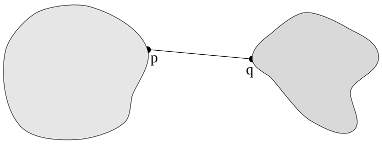


Figure 4.10: A bar constraint between points p and q

The constraint specifies that the distance between the two attachment points should always be a given distance l, hereby modeling a weightless bar of length l.

For $d = p - q$, from $||d|| = l$ we get constraint error function $C = (d, d) - l^2$ (possibly augmented with velocity terms). The reaction forces are derived from the one-dimensional restriction vector R using $f^p = \hat{d}R$ and $f^q = -f^p$, where $\hat{d}$ is the normalized version of d: $\hat{d} = \frac{(d)}{||d||}$. For the analytical derivatives calculations we need $\frac{\partial C}{\partial p}$, $\frac{\partial C}{\partial q}$, $\frac{\partial F_p}{\partial R}$ and $\frac{\partial F_q}{\partial R}$, which –from the equations above– can be seen to be:

$$\frac{\partial C}{\partial p} = 2d \qquad \frac{\partial C}{\partial q} = -2d \qquad \frac{\partial f^p}{\partial R} = \hat{d} \qquad \frac{\partial f^q}{\partial R} = -\hat{d}$$

Like other constraints, the bar constraint can also be fitted with a `maxforce` attribute which can be used to set a maximum strength for the constraint.

The constraint can also be used to model a rope: a rope is half a bar constraint: unlike a bar which can exert pushing and pulling forces to enforce its length, a rope can only enforce pulling forces. By deactivating the constraint as soon as a pushing force is detected, the behavior of a rope can be modeled. When a the constraint is disabled because of a pushing force, it activates a sort of controller which (at the start of each frame) monitors the actual length of the rope, and activates the bar constraint again when the actual length of the rope exceeds its rest length (because then the rope will need to start pulling again).

The state of a rope (whether it is exerting forces) can easily be obtained using the constraint's `active()` method.

---

the reaction force to later be combined with a friction force (which is likely not to be a central force since it will have to be applied to a point in the plane). This way, a user will be able to specify to which point in the plane the friction is applied.

4.1.12 The multibar constraint

A multibar constraint imposes a distance constraint between several attachment points.

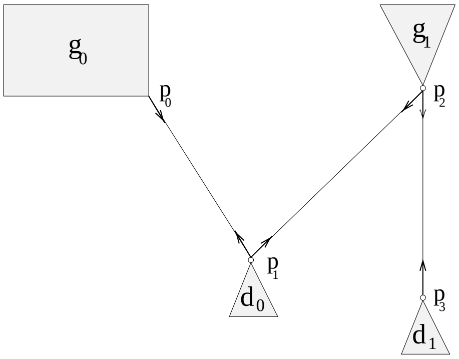


Figure 4.11: A multibar constraint between points p_0 to p_3

The constraint is inspired by the rope variant of the `bar` constraint, which can be generalized to be able to make pulley systems. In such a pulley system (see Figure 4.11 for an example), a rope is spanned from one attachment point to the next, where the connection points can move freely along the rope (except for the end points). As with the `bar` constraint, the `multibar` tries to maintain the its total length by exerting forces along the direction of the rope. The rope variant of the constraint models a rope in a pulley system as described above.

The constraint error equation for a multibar of length l with N attachment points is:

$$C = -l + \sum_{i=0}^{N-2} ||d_i|| \qquad \text{for } d_i = p_{i+1} - p_i$$

And the forces exerted in the attachment points are:

$$\begin{aligned} f^{p_0} &= \hat{d}_0 R \\ f^{p_i} &= (\hat{d}_i - \hat{d}_{i-1}) R \quad (0 < i < N-1) \\ f^{p_{N-1}} &= -\hat{d}_{N-1} R \end{aligned}$$

From these equations the derivatives can be easily determined, similar to the derivatives of the bar constraint.

Like the bar constraint, the multibar also has a `maxforce` attribute, and a ‘rope mode’ (along with a multirope controller) in which the constraint is deactivated as soon as a pushing force is detected.

The multibar constraint also has a method using which an attachment point can be added to the end of the current list of attachment points (for initialization).

4.1.13 The wheel constraint

Using the point-to-surface constraint, objects can be modeled that slide across surfaces. However, this contact is frictionless. In the other extreme, an object can not slide at all: in that case such an object would have to roll in order to move across the surface. As a first step into modeling rolling objects, a wheel will be modeled using a so called wheel constraint.

A wheel constraint models the interaction between the wheel and the surface which the wheel is rolling on. It calculates and applies the reaction force that is required for the wheel to exactly stay in contact with the surface (not go through it, nor lose contact). That way, the wheel constraint will not have to rely on complicated and expensive collision detection to avoid the wheel penetrating the surface. The wheel constraint also insures that the motion of the wheel is a rolling motion, i.e. the wheel can't move sideways, and it has to roll to go forward. This rolling motion means that in order to move a given distance in the surface, the same distance along the wheel circumference will have to be traversed.

This leads to the notion of *rolling distance*. Once the rolling distance is known, the next contact points on the wheel and the surface can be found, and a regular point-to-point constraint can then take care of calculating the appropriate reaction force that ensures that the contact points on the wheel and on the surface coincide at the next time instance.

The rolling distance can be determined from the property that in the contact point the wheel and the surface should be tangent. If this is not the case, the wheel will intersect with the surface as shown in Figure 4.12 below. This figure also shows how an estimate for the next point of contact can be improved.

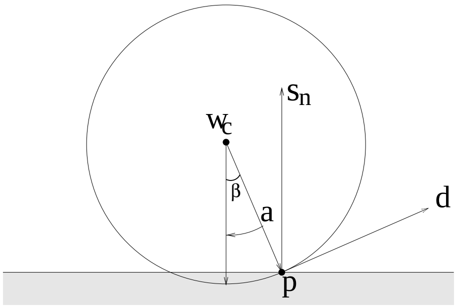


Figure 4.12: Rolling distance calculation for the wheel constraint

The point of contact on the wheel is characterized by the angle of rotation in the wheel plane with respect to the body fixed coordinate system of the wheel. The derivative of the circumference circle in the contact point should be perpendicular to the normal of the surface (at the contact position). If it is not, the angle between that derivative and the surface normal can be used to compute the adjustment angle β and thereby the change in rolling distance $w_r\beta$ (with w_r being the radius of the wheel).

Once this rolling distance is known, the same distance will have to be traversed in the surface that the wheel is rolling on. Such a surface is assumed to be parameterized in the same way as for the point-to-surface constraint. We can then linearize the surface locally and approximate it by the plane through the current contact point and spanned by the two vectors that are the derivatives in both parameters. The surface parameters are then adjusted in a manner which would correspond to the wheel rolling forward over the rolling distance in the plane.

The wheel exhibits a problem with energy conservation. It is caused by the discretization of the position of the (changing) application point of the reaction force to the wheel. This is depicted in Figure 4.13a: if the reaction force is applied to the contact point at time t, the reaction force clearly accelerates the wheel due to the torque exerted by the force as the reaction force is applied to the wheel as indicated by the dotted arrows between time t and $t+h$. If the reaction force is applied to the contact point at time $\frac{t+h}{2}$, the situation depicted in Figure 4.13b arises: here the torque effects due to the wrong point of application of the reaction force cancel each other during the integration interval (in first order).

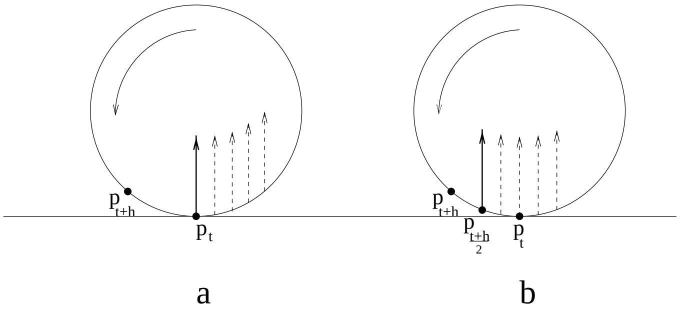
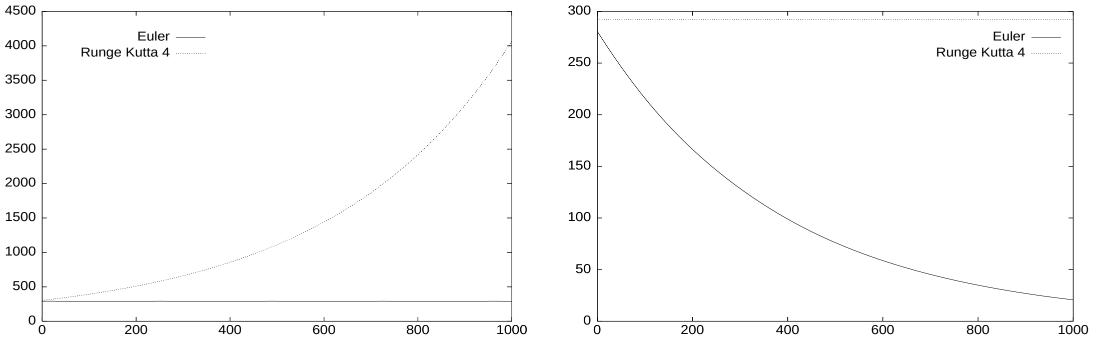


Figure 4.13: Wheel reaction force contact point without and with correction

This improvement has been implemented in the `wheel` constraint. At time t the exact contact point at time $\frac{t+h}{2}$ is not known yet, but a first order estimation is made based on the contact points at times t and $t-h$, and this suffice since the correction is also only of first order.

Figures 4.14a and 4.14b depict the results of tests in which the energy for a free-rolling wheel is measured over time for the cases without and with the correction to the attachment point. The figures show that the correction only needs to be applied for the Runge Kutta 4 integrator, which also follows from he fact that the Euler integrator only samples the reaction force at time t. The Runge Kutta 4 integrator samples the reaction force at times t, $t+\frac{h}{2}$ (twice), and $t+h$, so for this integrator, the correction causes the cancelation of the unwanted torque. The Runge Kutta 2 integrator samples at times t and $t+h$, so for that integrator the same correction as for the Runge Kutta 4 integrator should be applied, but for the double Euler



a) no correction **b)** correction of $\frac{1}{2}h$

Figure 4.14: Energy over time for a free-rolling wheel

integrator (which samples at t and $t + \frac{h}{2}$) the reaction force should be applied to $p_{t+\frac{h}{4}}$ to provide the correct cancelation (these correction factors correspond to the factors derived for the modified `ode` method in Appendix B).

In order to support the different correction factors, the motion integrator class is fitted with a method that returns the factor f that indicates that the integrator samples on average at time $t + fh$. So the Euler integrator returns zero, the double Euler integrator returns $\frac{1}{4}$, and both Runge Kutta integrators return $\frac{1}{2}$ from this method.

Two attempts were made to further improve the precision of the modeling of a wheel. Below is explained why these were not incorporated.

First, an attempt was made to more precisely model the trajectory of the wheel in the surface (along which the contact point is moved by the rolling distance). For a wheel which is angled with respect to the surface, one would expect a circular motion in the surface, as depicted in Figure 4.15, so the trajectory would better be approximated by a circle (the radius of which might be determined from the rolling speed of the wheel, the angle between the wheel and the surface, and also the gravity vector). However: when other forces act on the wheel, the shape of this trajectory can not easily be determined. For example: the trajectory of the rear wheel of a bicycle (see Section 5.2) is mostly determined by the way the whole bicycle is steered (i.e. by the angle between the fore fork and the frame), and not as much by the local orientation of the wheel relative to the surface. So an improved estimate for the trajectory of the wheel in the surface can not easily be made from the local configuration. Future research could show if determining the circle from the previous contact positions in the surface would improve the behavior of the simulated wheel.

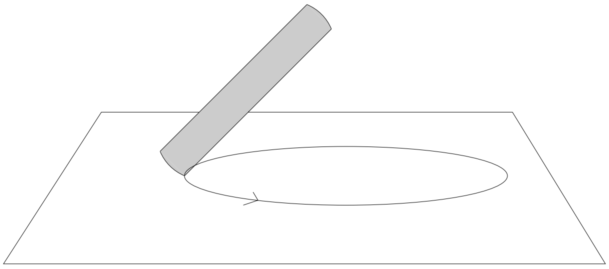


Figure 4.15: The rolling trajectory in the surface for the wheel constraint

Second, the thickness of the wheel (also depicted in Figure 4.15) might be taken into account, by offsetting the contact point in the wheel according to a circle or ellipsoid perpendicular to the rolling direction. This offset would be in proportion to how much the angle between the wheel and the surface would deviate from perpendicular (bounded by a not too large maximum offset). This way, as the wheel tilts, the contact point moves inward a little, providing for a little stabilization of the motion of the wheel, since a force exterted to such a shifted point would not excert as large a torque as a force to the original point. So the wheel would not fall over so easily. However, the effects of this adjustment were found to be marginal, so it is not warranted to spend the extra computational effort. Furthermore, since the wheel constraint explicitly maintains the position of the contact point in the wheel, it should only be used for wheels that have not fallen too far sideways: for such wheels the contact point moves around the wheel's circumference at an ever higher frequency as the wheel falls more and more flat, resulting in high frequencies which can not be handled properly by the mechanism from Figure 4.12 which moves the contact point along the wheel by linear approximations. Currently, in cases where the wheel falls too flat, the wheel constraint deactivates itself.

4.1.14 The collision constraint

Collision handling can be implemented by constraint handling, if the right constraint can be found to model the collision process. Collision detection is a separate problem treated elsewhere (for example in [Berg 99]). When collision handling is implemented with a collision constraint, the collision detector only has to create a (temporary) collision constraint upon detection of a collision. The constraint manager will then handle the collision.

Collision handling comprises calculating and applying reaction forces/torques/impulses that repel two objects which otherwise would penetrate each other. For now we will assume a friction-free contact between the colliding surfaces: the collision forces only act in the direction of the so-called *collision normal*. For a point colliding with a surface (as is often the case), this collision normal is the normal of the surface at the contact point. Unless otherwise indicated, all forces, impulse changes, relative velocities and distances will be assumed to be in the direction of the surface normal, for the remainder of this section.

Since the simulated objects are completely rigid, collisions only take an infinitely short amount of time: the energy exchange between the objects takes the form of an exchange of impulse at the time of the collision. But since the dynamics system samples time at regular discrete time intervals, applying a change of impulse at exactly the collision time implies sub-sampling. This can gravely slow down a simulation, and this can hamper interactive simulations. So we will not use sub-sampling (see design decision 5). The best we can then do in terms of impulse exchanges is to do these exchanges at either the sampled time stamp before the collision, or at the sampled time stamp after the collision.

An alternative is to solve the collision using collision forces: treat the constraint just like any of the previous ones, and apply a constant force over the whole time-interval in which the collision takes place. As will be shown below, this can still give rise to some aliasing problems (but they can be remedied), and it corresponds better to collisions of not-entirely-rigid bodies where there is a finite contact time due to small deformations at the contact points.

Let us consider in which terms to formulate the collision constraint. In a collision with point masses (such as depicted in Figure 4.16 below), we know that the motions conserve the total amount of momentum and energy in the system. These can not be used directly as constraint equations: a set of reaction forces and torques each being zero also abides these laws but does not handle the collision properly. Also, for articulated rigid bodies it is not possible to determine exactly what energy to measure (there may be energy losses in the other connections due to friction).

A generally used heuristic is to simply mirror the relative velocity of the two colliding points (Newton's impact law). For most cases this seems to work (it is correct for colliding point masses, see below), although Brian Mirtich reports in [Mirti 96] that the use of this law can increase the total energy in the system when friction is present. He also presents more advanced ways of formulating the collision equations, but these involve sub-sampling as they require information about properties of the colliding geometries at the time of the collision.

Let us examine how Newton's impact law can be used to model the collision, starting with a proof that mirroring the velocity does not violate the laws of conservation of momentum and energy.

Consider the collision between two point masses depicted in Figure 4.16. Both point masses have linear motion before and after the collision. One point mass has mass M, velocity V before the collision, and velocity W after the collision. The other has mass m, velocity v before the collision, and velocity w after the collision. To this situation the laws of conservation of momentum and energy apply:

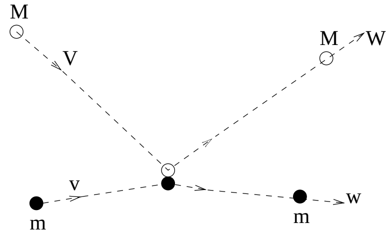


Figure 4.16: A collision between two point masses

$$mv + MV = mw + MW = 0 \tag{4.4}$$

$$mv^2 + MV^2 = mw^2 + MW^2 \tag{4.5}$$

Choosing the total momentum to be zero in equation 4.4 determines a coordinate system in which the following derivation is easily expressed (and does not constitute a loss in generality).

Let us see what happens if we impose mirroring the relative velocities of the two colliding points:

$$v - V = W - w \tag{4.6}$$

From equation 4.4 we can see that $v = -\frac{M}{m}V$ and $w = -\frac{M}{m}W$. Using this in equation 4.6 we get:

$$\begin{array}{lrcl} & v - V & = & W - w \\ \equiv & & & \\ & -\frac{M}{m}V - V & = & W + \frac{M}{m}W \\ \equiv & & & \\ & -(\frac{M}{m} + 1)V & = & (1 + \frac{M}{m})W \\ \equiv & & & \\ & -V & = & W \end{array}$$

$$\begin{array}{lrcl}
\equiv\!> & & & \\
 & V^2 & = & W^2 \\
\equiv & & & \\
 & (\frac{M}{m}+1)MV^2 & = & (\frac{M}{m}+1)MW^2 \\
\equiv & & & \\
 & m(-\frac{M}{m}V)^2+MV^2 & = & m(-\frac{M}{m}W)^2+MW^2 \\
\equiv & & & \\
 & mv^2+MV^2 & = & mw+MW^2
\end{array}$$

So mirroring the relative velocities is consistent with conservation of energy and momentum. When we use mirroring of the velocities as the collision constraint equation, the following factors also come into play:

1. The collision constraint only specifies mirroring of the velocity in the direction the collision normal.
2. External forces can cause non-linear motion of the points.
3. The colliding points are not point masses anymore but points of a rigid body. This can also cause non-linearities in the motion before and after the collision.
4. Time is discrete.

Factor 1 is not a problem, since the reaction force is also in the direction of the collision normal.

Factors 2 and 3 are only problems if the step size is very large: for small step sizes, the higher order terms in the motion of the colliding points will be very small. This is illustrated in Figure 4.17.

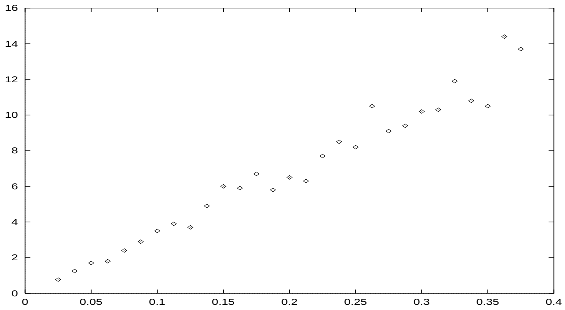


Figure 4.17: Average energy change per collision as a function of the simulation step size

In this figure, the average change in energy (which should be zero) for the first fifty collisions of a bouncing octaeder have been plotted against the simulation step size. It is clear that the error decreases with decreasing step size.

The fourth factor can lead to aliasing problems because any forces in the system are assumed to work over the complete time interval (in this case the time interval in which the collision takes place) and are assumed to be constant. Figure 4.18 shows five trajectories for a colliding point mass.

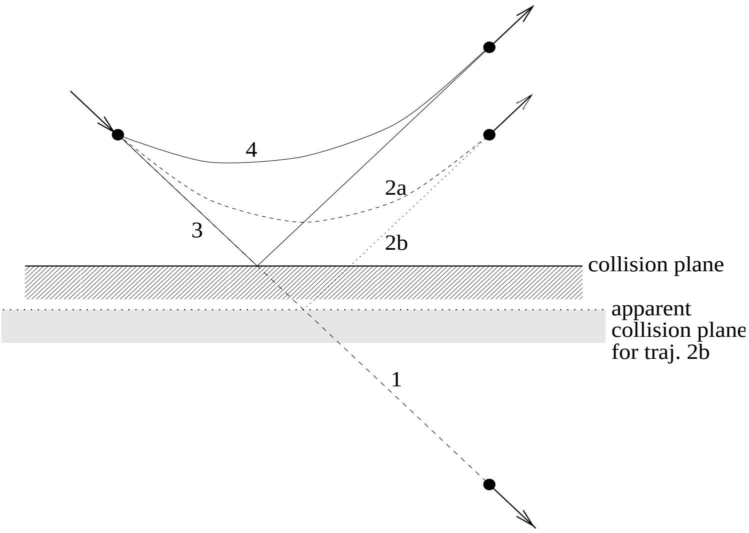


Figure 4.18: Trajectories for a point mass colliding with a surface

These trajectories depict:

1	the trajectory as if there were no collision.
2a	the parabolic trajectory under a constant (upwardly directed) force (mirroring the velocities before and after the collision as measured on at the start and the end of the simulation interval).
2b	the apparent trajectory if you only look at the point mass in trajectory 2a at the time stamps before and after the collision. This trajectory creates the appearance that the collision plane was actually positioned lower (positional aliasing).
3	the ideal trajectory where the collision is instantaneous and the velocity is mirrored
4	a parabolic trajectory that would correspond to the ideal trajectory, where velocity is mirrored, and there is no positional aliasing. Such a trajectory needs an impulsive velocity change at time t to allow for a then-constant collision force between t and $t+h$.

The figure clearly shows positional aliasing as the distance from the point to the collision plane at the end of the interval is different when comparing the simulated trajectory (2a) and the ideal trajectory (3).

This aliasing effect is remedied by adding an extra positional constraint to enforce the resulting position from trajectory 3. This position is obtained by taking the resulting distance from trajectory 1 and mirroring it in the collision plane. This distance is known since the position without collision has to have been calculated before the collision detector can detect that a collision actually took place.

We can use an impulse change at time stamp t to create the extra required degree of freedom in the steering parameters. So the collision constraint then becomes a two-dimensional constraint consisting of a one-dimensional positional constraint and a one-dimensional velocity constraint, which can be enforced by applying an impulse change at the time stamp before the collision, and a collision force during the time interval in which the collision takes place.

Until now we have assumed that collisions are purely elastic. In reality collisions often have a dampening effect. This effect can be easily modeled by applying an elasticity factor when mirroring the relative velocity and relative position. The elasticity of a collision is determined by the material properties of the colliding geometries. So we extend the `geometry` class to hold an extra attribute `elasticity`. When two geometries collide, the average of the two elasticity factors is taken as the collision's elasticity factor.

The scheme above implements point-point collisions. In practice also line-line, or plane-plane collisions may occur. These may be modeled by an appropriately chosen set of point-point collisions, but further attention should be directed to how this set can be determined from collision data that is made available by a collision detector. The amount of work that this brings with it depends largely on the what type of collision data is provided by the collision detector, and how well this fits with the point-point approach of Dynamo. Section 2.2 of [Buck 98] suggests a method of updating sets of point contacts. In [Mirti 98], a method is described to classify contacts as either edge-vertex, face-vertex, or edge-edge allowing the tracking of collisions. Some contact configurations require sets of these three basic contact types to be accurately represented.

The scheme described above works without problems when one vertex of a geometry collides with another geometry, giving rise to one collision constraint. However, there might be problems to overcome when multiple collision constraints are active:

1. The extra positional constraint enforces a linearized estimation of the relative position. For small step sizes, the linearization itself is not a big problem, but when more than one vertex pair between two geometries collide, these local linearizations might lead to slightly conflicting constraints (instead of redundant ones), which can give rise to instable constraint satisfaction calculations.

2. Slightly conflicting constraints might also be the result of the velocity mirroring itself, since the target velocities are based on velocities taken from the simulation. Suppose for example that two geometries are connected by a point-to-point constraint which for some reason is not entirely satisfied at time t, given a slight relative velocity between the points at that time. If the two points would collide between t and $t+h$ with a third geometry (and if one point collides, the likeliness of the other point colliding is very large since they are in close proximity), the collision constraint would enforce mirrored velocities which would likely specify a mirrored relative velocity (in the direction of the collision normal) at $t+h$. This would be contradictory to the point-to-point constraint.

These two problems should be looked into in future research, and can better be studied when Dynamo is combined with a collision detector and the problem of mapping arbitrary collision configurations to point-point collision sets is solved.

4.2 Specific controllers

In Section 3.3, the general structure of a controller was explained. Using the theory and the base classes discussed there, this section presents a few specific controllers. First, a simple controller modeling springs is discussed, after which a generalization is made to PID controllers (which need specific sensors and actuators). Since controllers apply forces and/or torques and impulses, the controller class inherits from the `force_drawable` class discussed in Section 3.5.1.

4.2.1 The spring and damper controller

A spring connects two attachments point and tries to keep these points at a given rest length l_0. To this end, a spring applies a force $f^s_t = c(x_t - l_0)$ in the two attachments points, where c is the spring constant and x the distance between the two attachment points.

Similarly, a damper exerts a force $f^d_t = d\dot{x}_t$ on its attachments points (where d is a damping constant).

The two can be combined into a dampened spring class, which has both d and c and l_0 as its parameters. A spring has methods to set these attributes. The two attachments points (at least one of which belongs to a dyna) should be set upon the spring's construction.

Using simply $f^s_t = c(x_t - l_0)$ for the spring force results in discretization errors: if the force-magnitude is based on a measurement at the start of a frame (when it has to be calculated), and the relative distance between the two attachment points changed a lot, the force magnitude is too low when the attachments points are moving apart, and too high if they are moving toward each other. A better estimate of the average force can be made by estimating the average distance of the attachment points, by taking the first derivative (already needed for the damping force anyway) into account: $f^s_t = c(x_t + \frac{h}{2}\dot{x}_t - l_0)$.

The `calculate_and_apply` method of a dampened spring measures x_t and its derivative, and then applies $f^s_t + f^d_t$ to it attachment points (in opposite directions, of course).

There is a rotational variant of the spring: the (dampened) torque spring. It measures the angle between two given directions (internally integrating differences to keep track of angles bigger than 360 degrees), and applies a torque to both geometries connected to the spring in the same way as the "regular" spring applies forces.

4.2.2 The PID controller

The spring controllers can be seen as specific versions of a much wider class of controllers: the class of PID controllers (for a more indepth discussion of PID controllers, see [Vegte 90]). A PID controller takes an input signal (usually taken as the difference between the output of a sensor and a reference signal) and generates an output signal that is a summation of a term that is **p**roportional to the input signal, a term that is proportional to the **i**ntegral (over time) of the input signal, and a term that is proportional to the **d**erivative of the input signal:

$$o_t = c_p i_t + c_i \int_0^t i_t dt + c_d \frac{\partial}{\partial t} i_t \tag{4.7}$$

Here, i_t is the input signal at time t, o_t is the output signal (which is usually sent to an actuator), and the c_* are the controller parameters (to be provided by the user of the PID controller).

For a discrete system like we are using, equation 4.7 becomes

$$o_t = c_p i_t + c_i \sum_0^t i_t + c_d \frac{(i_t - i_{t-h})}{h} \tag{4.8}$$

In a similar way as for the springs, motion estimation can be added to arrive at a better estimate for the input value halfway the integration interval to arrive at a more precise steering.

A PID controller needs an actuator and a sensor, so those will have to be provided to the constructor of the class. A user should also be able to set and retrieve the current values of the controller constants and the reference value that serves as a target for the controller to match its input with. So the interface of the PID controller class has methods enabling performing these tasks.

The PID controller delegates the actual calculation and application of the controller forces to its actuator (see [Gamma 95] for a description of the *delegation* design pattern), so it will act as a proxy (see also [Gamma 95] for a description of the *proxy* design pattern) to the actuator for the methods used in visualizing forces.

Normally, a PID controller takes care of reading the sensor and activating the actuator at the proper times. When the class library is used in a host system which works with a different implementation language (such as the GDP, which uses the language Looks), it is advantageous if the sensors and actuators can be written in the host system language (an example of this is seen in the roller coaster example in Section 5.5 where there is a controller that pulls the roller coaster carts up a slope at a given velocity: this controller uses the `push` method of the Looks cart class as its actuator). When implemented in a different language, it is not possible for the actuators and sensors to inherit from the C++ base classes provided by Dynamo. For those cases, a method is provided that only implements equation 4.8: when using that method, a user has to take care of providing sensor readings (these have to be given as a parameter to the method) and sending the resulting actuator value (which is returned from the method) to an actuator.

The choice of a sensor and an actuator make a PID controller suitable for a specific task. A very obvious task is to control the remaining degree(s) of freedom of a constraint. To this end, suitable sensors and actuators have been implemented for a few of the constraints. These are discussed next.

Specific actuators and sensors

The prism constraint has one degree of freedom left between the two geometries that are connected by it. This translation along a given vector can be controlled using the proper sensors and actuator. Two sensors are currently available: one that measures the translational distance along the vector (used for controlling relative position), and one that measures the translational velocity along the translation vector (used for controlling relative velocity). Either sensor can be used along with an actuator that applies a force in a given direction (the direction given being the translation direction).

The line hinge constraint also has one degree of freedom left between the two geometries that it connects. For controlling this rotation along a given axis, two sensors and an actuator are available, similar to the ones mentioned above: the sensors measure the relative angle (integrating it to keep track of angles larger than 360 degrees) and relative angular velocity along the rotation axis, and the actuator applies a torque along the rotation axis.

5 Examples

In the previous sections, we have shown the theory and the design of Dynamo. This section shows a few examples of its use. First, the use of the Dynamo library as a stand-alone library is illustrated by showing how a simple example can be programmed using the Dynamo classes. Then some more advanced examples follow, sampling several application areas of the library. The latter are all created using Dynamo within the GDP host system.

Of these more complex examples, the first shows how a bicycle exhibits coherent behavior on the level of the bicycle as a whole: there is emergent behavior when the several geometries out of which the bicycle is composed are connected to each other by the constraints. Interactivity is added in the next two examples: one where the properties of coupled pendula are examined, and another where the parameters in a door spring are fine-tuned. As a last case, the roller coaster example shows how dynamic simulation can help and facilitate the construction of a mechanical system.

These examples serve to show the use of the Dynamo classes to build animations, as well as the performance of Dynamo.

5.1 A simple example

To illustrate the direct use of the Dynamo classes (without the Looks wrapper that was created to make it available to the GDP), a simple example is created, where a chain consisting of connected cubes is swinging in gravity. See Figure 5.1 below. The source code of this example is included in Appendix I.

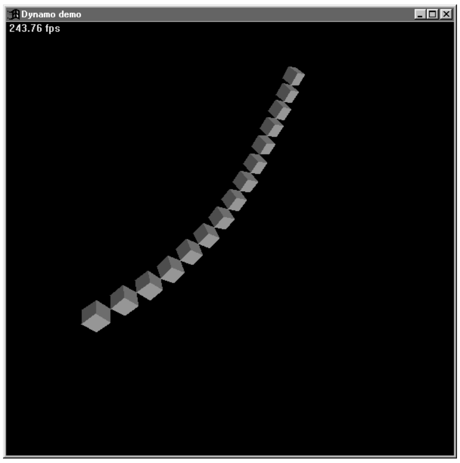


Figure 5.1: The swinging chain of cubes of the simple example

In the example, cubes are used as the geometries that need be controlled. There is a `MyCube` class which models these cubes. The external function `RenderCubes` knows how to render these, so the animation can be visualized. Each cube has a companion `dyna` to perform its motion calculations. To communicate the results of these calculations, two callbacks are implemented (for a more detailed description of these callbacks, see the user manual in [Baren 99]).

A cube can be connected to another cube using the `ConnectTo` method, which creates a point-to-point constraint between the two cubes. Using this method it is also possible to fix a point of the cube in world space (this is used to prevent the upper cube of the chain from falling).

In the main routine of the example, the dyna system is initialized and the cubes are created and connected to each other to form the chain. Then gravity is specified to make the chain swing.

The swinging motion is calculated by Dynamo though the repeated call to its `dynamics` method. After each step, the cubes are rendered. After 70 simulated time units, the lower half of the chain is disconnected from the upper half by removing the middle constraint, so that the lower half starts falling, while still continuing the rotational motion it had at the moment it was released.

Figure 5.2 below shows performance measurements of the example in the form of the frame rate (frames/sec) and the cube rate (the product of the frame rate and the number of cubes in the chain) as a function of the number of cubes in the chain. These were measured on a 300MHz Pentium II PC with 3D graphics accelerator. For a very low number of cubes, rendering is the bottleneck. This accounts for the increase in cube rate for lower numbers of cubes in the chain. For larger number of cubes in the chain, the cube rate decreases slightly, showing an almost linear time complexity in the number of constraints. This is due to the use of the bandwidth-optimized LU decomposition which scales linearly with the number of constraints.

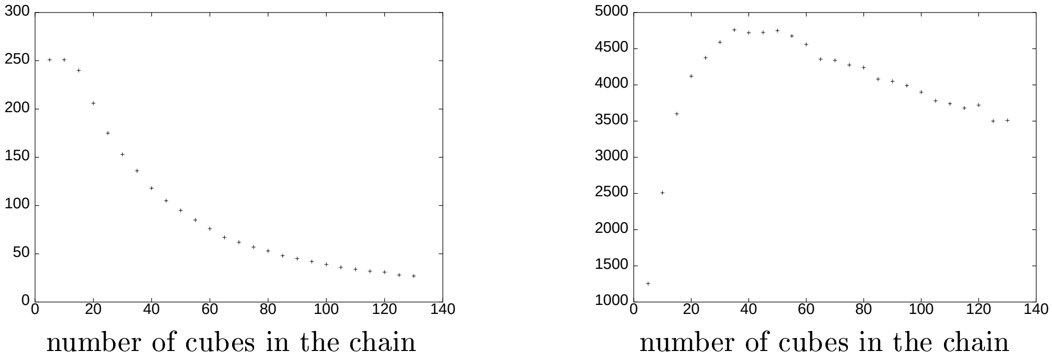


Figure 5.2: Frame rate (frames/sec), and cube rate (cubes/sec)

The slight decrease in cube rate can be attributed to an increase in the number of iterations required in the loop of the `satisfy_constraints` method (see the pseudo code on page 41): two constraint forces are applied to each cube. These two forces mostly cancel each other in their effects, relaying the ‘weight’ of the remaining chain hanging from such a cube to the cube above it in the chain. As the chain gets larger, the resulting force on cubes high in the chain therefore gets smaller compared to the reaction forces that are calculated, requiring more precision in the calculations of these forces for an equal precision in the resulting force (which determines the precision with which the constraints are satisfied).

5.2 The bicycle

In the bicycle example, a bicycle is constructed out of several parts: a frame, a fore-fork, the pedals (mounted on a crank) and two wheels. See Figure 5.3. These are connected using constraints: a line hinge between the fore-fork and the frame, a line hinge between the front wheel and the fore-fork, a line hinge between the rear wheel and the frame, an orientation constraint between the crank and the rear wheel (modeling the chain), and a point-to-point constraint between the midpoint of the crank and the frame (the orientation constraint already takes care of the proper orientation). The bicycle is then connected to the floor on which it is rolling using two wheel constraints.

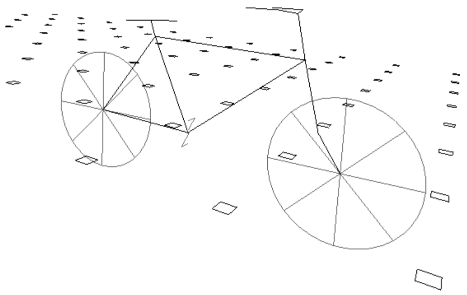


Figure 5.3: The configuration of the bicycle

To set the bicycle in motion, a torque is exerted on crank. The orientation constraint distributes this torque and sets the rear wheel in motion. The rear wheel (because of the no-slip condition of the wheel constraint), has to start rolling. This pushes the frame, which pushes the crank and the fore-fork, which in turns starts the front wheel rolling. All these energy transfers are taken care of by the constraints: a user only has to exert the torque on the pedals after the bicycle has been assembled. The internal forces and torques and energy transfers are all handled by Dynamo.

To show the emergent behavior of the bicycle as a whole, a force is briefly exerted on the frame from the side (after the bicycle has been made to start rolling), pushing the bicycle over. A well known behavior is that a rolling bicycle is self-stabilizing when the fore-fork is properly bent (as it is in Figure 5.3). The rotation axis between the fore-fork and the front wheel has to lie in front of the rotation axis between the fore-fork and the frame. The resulting simulation indeed shows that this is the case for the simulated bicycle: the bicycle over-corrects and starts falling over the other way, which is again corrected, and this continues for a while, while the left-right motion damps and eventually the bike is rolling along again.

The same experiment was performed again, but with the fore-fork substituted by a straight one. This bicycle does not exhibit enough self-stabilizing behavior, and falls over after having been pushed.

A frame of each animation in Figure 5.4 (taken after the same number of frames after the sideways push) shows how the first bicycle remains rolling, while the second bicycle falls over.

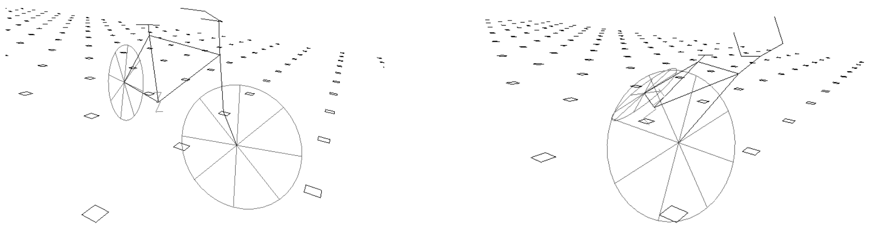


Figure 5.4: Both bicycles after having been pushed

The trajectories of the wheels of both bicycles are shown in Figure 5.5 (the bicycles start at the left, and are pushed toward the top after a while). The figure shows that both bicycles follow the same trajectory until a little while after the sideways push, when the unstable bicycle makes a very steep turn before falling over, whereas the stable bicycle keeps rolling as its left-right motion dampens.

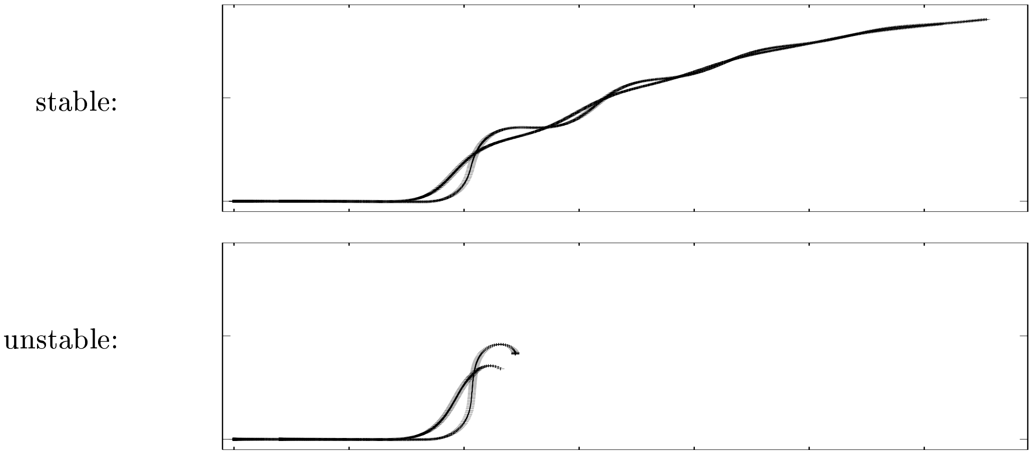


Figure 5.5: View from below of the trajectories of the wheels of both bicycles

These two simulations (links to animations of which are available on the world wide web at the address listed in [Baren 99]) show how an articulated rigid body can start to exhibit behavior on a global scale, which results in a more complex (and often more interesting) motion than that of single rigid bodies.

The constraints used in the bicycle example add up to 30 degrees of freedom, yielding a 30×30 $\frac{\partial C}{\partial R}$ matrix. Because of the high interconnectivity of the bicycle parts, this matrix is dense and the optimizations with respect to sparse matrix techniques discussed in Section 3.4.6, are not useful. Nevertheless, the examples are computed and displayed (using the GDP as a host application) at roughly 25 frames per second on low-end machines as the Sun Sparcstation 5 and the SGI Indy R5000. Simulation on an SGI O2 yields roughly 30 frames per second. These frame rates show that more complex examples are still possible at interactive frame rates.

In the future, interactivity could be added to this example by putting the steering of the bicycle under the control of the user. Just adding a PID controller that governs the angle of the fore-fork with respect to the frame would not not sufficient however: next to steering, there should be a sideways weight-shift to prevent the bicycle from falling over (opposite the turning direction, due to the centrifugal forces). It should even be possible to steer the bicycle by weight-shift only. This could be implemented by connecting a weight to the saddle of the frame of the bicycle, using a prism constraint which allows sideways translation. A PID controller would be required which would control the position of that weight, along with a second PID controller that would control the steering angle by providing a reference signal to the first controller.

The steerable bicycle would then be a good research vehicle for looking into more complex controllers: in this case for a controller that can do trajectory planning and navigate the bicycle through an obstacle course (steering the bicycle through the controllers described above).

5.3 The coupled pendula

The coupled pendula example adds some interactivity. It shows some properties of the dynamic behavior of a spring, and could as such be used for educational purposes in a "virtual physics laboratory" experiment. Figure 5.6 shows the configuration of the experiment: on the left, the square and the short bar model a motor: the primary crank is a kinematic object of `revolute` type (`revolute` is a GDP class for object that can rotate at a given rate). The kinematic parameter that controls the rotation rate of this bar is controlled by the user via a slider widget. Figure 5.7 shows this widget, in a frame from the animation, with the calculated constraint forces visualized. The primary crank drives the coupling bar (it is connected to it by a point-to-point constraint), which in turn drives the left pendulum (again through a point-to-point constraint). The mounting bar is fixed in space and serves as an anchor for the two pendula (since the pendula are connected to it by point-to-point constraints). The left pendulum is driven by the motor (as described above). The right pendulum hangs freely, but is connected to the left pendulum through a spring (the horizontal line between the two bars). This spring transfers the motion of the left pendulum onto the right pendulum.

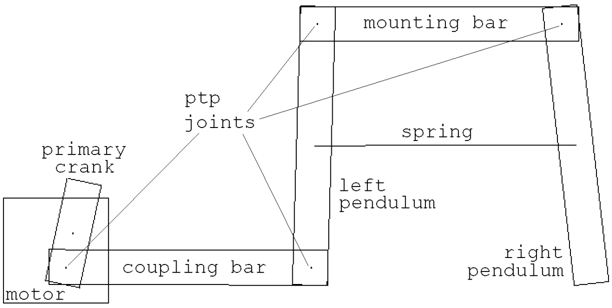
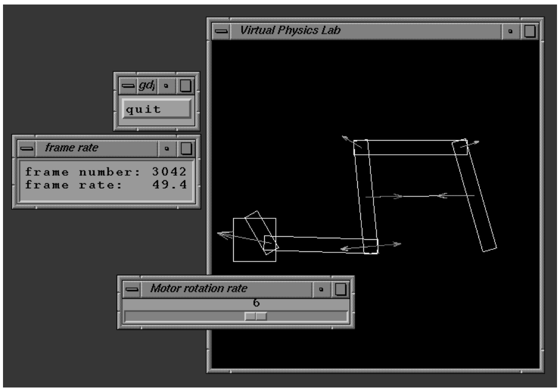


Figure 5.6: The configuration of the coupled pendula example

Through the slider, the user can adjust the motor rotation rate, and therefore the swinging



frame number: 3042
frame rate: 49.4

Motor rotation rate 6

Figure 5.7: The constraint forces within the coupled pendula example

frequency of the left pendulum. In [Fraue 66], a theoretical explanation is given of the behavior of such a coupled pendulum (see below for a comparison between this theoretical behavior and the measured behavior of the example). An informal explanation of the behavior is the following: if the driving frequency is very low, the spring simply transfers the motion of the left pendulum onto the right pendulum, and both pendula move synchronously. But when the frequency is increased, the spring "can not keep up" and the right pendulum starts lagging behind in its motion. That is: until the frequency is increased to the point where resonance occurs: then the amplitude of the swinging motion of the right pendulum even exceeds the amplitude of the left pendulum's motion. At an even higher frequency, the two pendula move in counter phase, and above that frequency the spring "can not keep up" at all, and the swinging motion of the left pendulum is only transfered to the right pendulum in a very dampened manner. This is illustrated in Figure 5.8, where measurements of the amplitude and phase characteristics are shown as a function of the rotation rate.

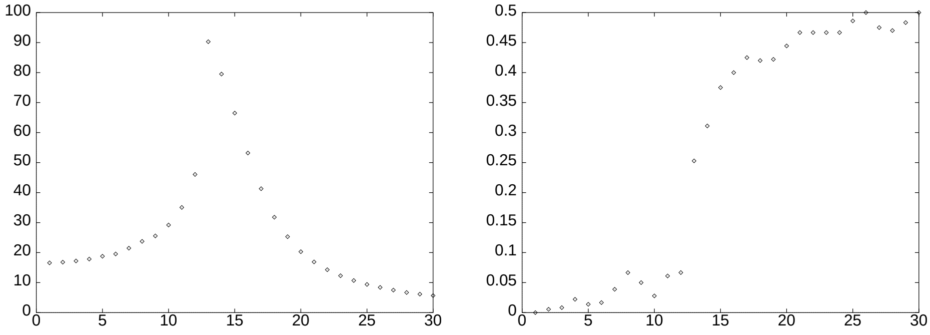


Figure 5.8: The amplitude and phase characteristics of the couple pendulum

The phase characteristic is not measured very precisely because of the large time steps in the simulation, and as a result shows some aliasing. Still, the figure matches very well with Figures 3-64a (amplitude) and 3-65 (phase) on pages 139 and 140 of [Fraue 66], which show the theoretical characteristics for such damped forced oscillations as these. Figure 5.8 shows that the peak in amplitude corresponds to the phase being one quarter, which occurs at the resonance frequency.

This example again shows emergent behavior of a complex system: the phenomenon of resonance is not explicitly programmed in the simulation yet shows as a macroscopic effect of the combined reaction forces.

The coupled pendula example runs at 60 frames per second on an SGI O2.

5.4 The door spring

The door spring example shows how a simulation can help in tuning a mechanical system. The construction shown in Figure 5.9 is used to close a door. The construction consists of a bar which is mounted on a wall via a turning mechanism which has a torque spring as a component. This spring turns the bar, which in turn pushes the door via a point-to-curve constraint that connects to a bar on the door. Figure 5.10 shows a screen capture of a running simulation, in which the door spring construction is shown in 3D.

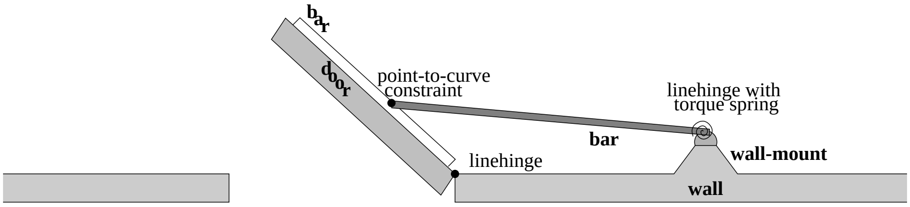
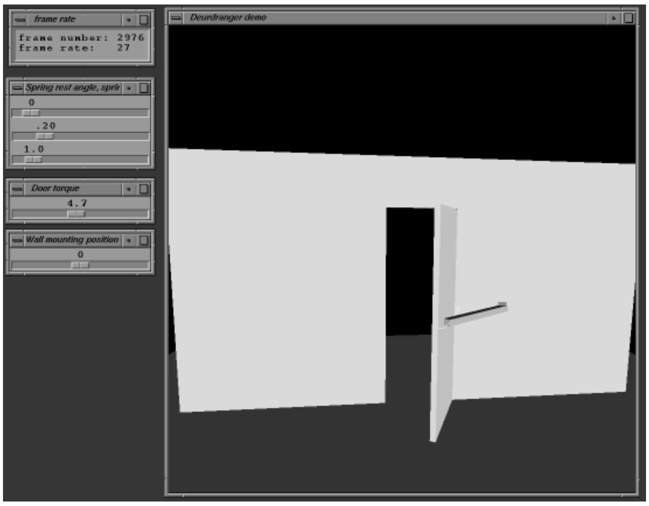


Figure 5.9: Top view of the door spring construction

There are several variables that control how the door closes. The spring constant, damping constant and rest angle of the torque spring are all relevant. Furthermore, the effect of the torque spring's forces on the door can be changed by horizontally moving the mounting point where the bar is connected to the wall (something that would be very hard to easily vary in reality). The door spring example puts all of these parameters under user control through a series of sliders. These are visible at the left of Figure 5.10. Furthermore, there is a slider through which a user can apply a torque to the door to open it.

Someone designing a door spring system has to arrive at a proper set of values for the parameters described above. Choices have to be made when tuning the parameters. When for example the door does not close fast enough, the designer can increase the spring constant of the torque spring. But then it will be harder to open the door. Alternatively, the designer might lower the spring's damping constant. But this might cause the door to swing through the door opening too far. Trying out different combinations, and being able to see as a result how much torque needs to be applied to open the door, how fast the door closes again, and how much the door swings through the door hole, can be very insightful and helpful in finding a good compromise.



```
frame number: 2976
frame rate:   27
```

Spring rest angle, sprir: 0, .20, 1.0

Door torque: 4.7

Wall mounting position: 0

Figure 5.10: Capture of the door spring example

5.5 The roller coaster

The previous example shows how dynamic simulation can be used to tune the parameters of a predefined mechanical system (a topic which is further studied in for example [Louch 96]). This roller coaster example shows how dynamic simulation can help in the construction process of mechanical systems proper.

Originally a very simple roller coaster demo was designed as a test for the point-to-curve constraint. This roller coaster (shown in Figure 5.11) had 'carts' that were hanging from a single track (each using two point-to-curve constraints). This track is modeled by a spline consisting of cubic b-spline segments. The carts would accelerate down the spiral (because of gravity) and then go through a looping before being towed up again. This demo however is not very realistic: the track is far too small with respect to the carts and uninteresting, and single-track roller coasters are not common. Also, some constructs like cork-screws are not generally possible when using a single track.

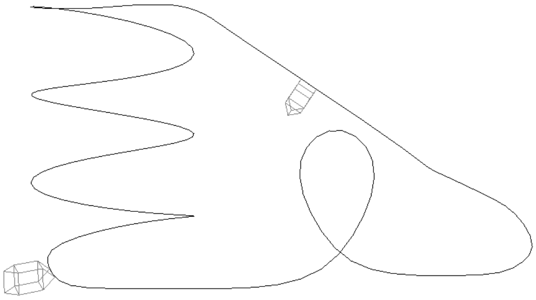


Figure 5.11: The simple roller coaster demo

So a more realistic roller coaster was in order, where there are trains of carts which roll along two tracks. Several snapshots of this roller coaster in operation are shown in Figure 5.12. The shape of the tracks of this roller coaster was based on the "python" roller coaster, which is an attraction in the Efteling amusement park. It is not an exact modeling of the python, but an approximation based on the pictures which were available on the web pages of the Efteling (the web pages have been changed since). The webpages also gave some properties of the python roller coaster: the tracks are 728 meters long, and the highest point of the tracks is 29 meters above ground. One ride takes 105 seconds, and the top speed of the trains is 85 kilometers per hour, at the end of the first loop. The track was modeled (again in terms of cubic bspline segments) to match the length of the actual track. Since the lowest height of the track was not given, the highest point in our virtual roller coaster might not have the same impact as the height of the actual ride. The duration of the ride is also of little use, since it depends mainly on the velocity with which the carts are pulled up the ramp at the start of the ride.

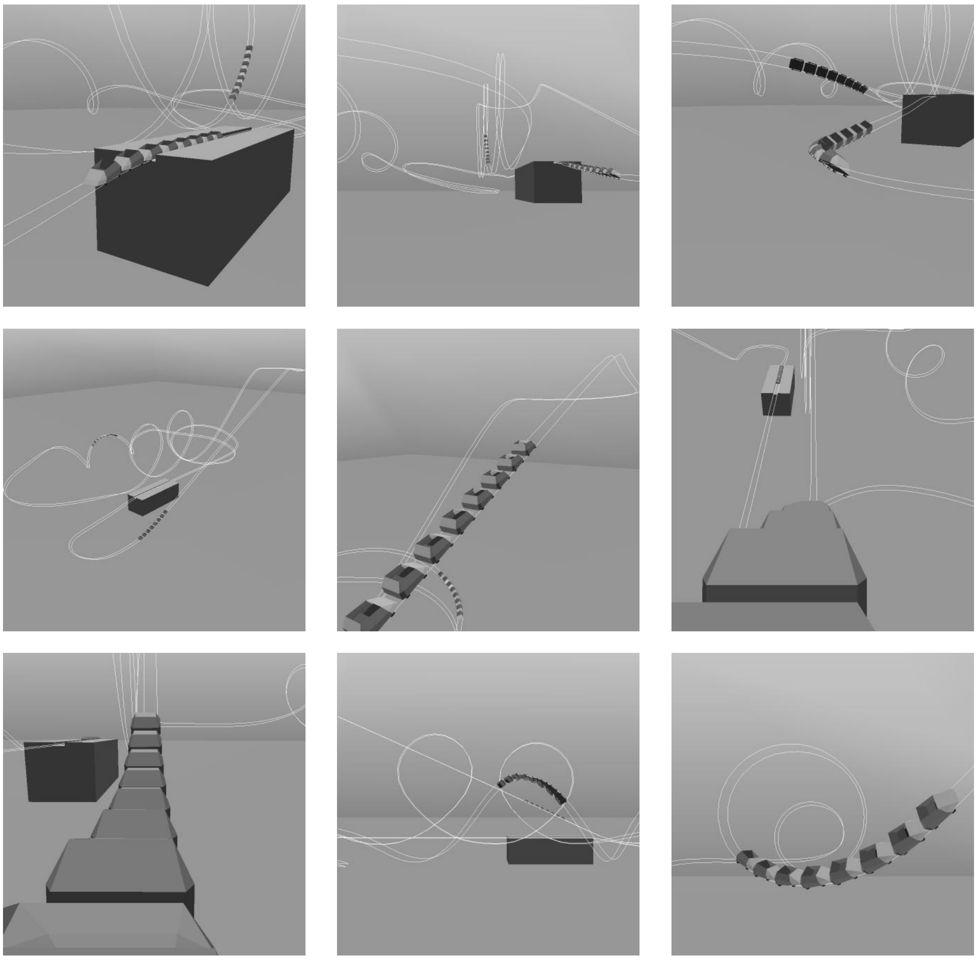


Figure 5.12: Several pictures of the virtual roller coaster ride

One of the problems encountered when extending the simple roller coaster of Figure 5.11 is that the single track has to be split in two (more or less parallel) tracks. Specifying both tracks separately is cumbersome for a track designer, since the two tracks are very much alike. So it would be easier if a track designer could just specify a single curve (as in the simple roller coaster demo) out of which the two final tracks could be derived automatically. If the curve given by the track designer is interpreted as a “spine” then the remaining task is to determine the required amount of rotation of the left and right tracks (which are shifted equal distances opposite each other from this spine) along the spine: i.e. there is one degree of freedom left in the form of the orientation along the spine of the left and right tracks.

A convenient representation of this orientation is an “up” vector, which determines to which direction the top of a cart is oriented. From the up vector and the direction vector of the spine, the left and right tracks can easily be determined. Since each track is also represented by a spline consisting of b-spline segments, we can take the control points of the center spline and assign an up vector in each of them, and from there calculate the control points of the left and right splines. This yields a left and right track where the distance between the left and right tracks is at most the distance between the corresponding control points.

Since the distance between the tracks can vary slightly (the bspline segment curves that represent the tracks are not offset curves, so the distance between the tracks can become less than the distance between the control points), the carts are built up from two halves (depicted in Figure 5.13). These are connected by a prism constraint, to allow the two parts of the cart to slide sideways into each other a little when the tracks come closer to each other). The right half of the cart is connected to the right track by a point-to-curve constraint, and the left half of the cart is connected to the left track by two point-to-curve constraints. So each cart contributes eleven degrees of restricted freedom to the constraint manager (five from the prism constraint, and two for each of the three point-to-curve constraints).

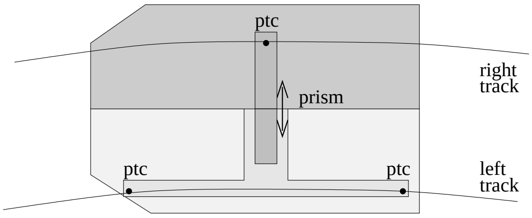


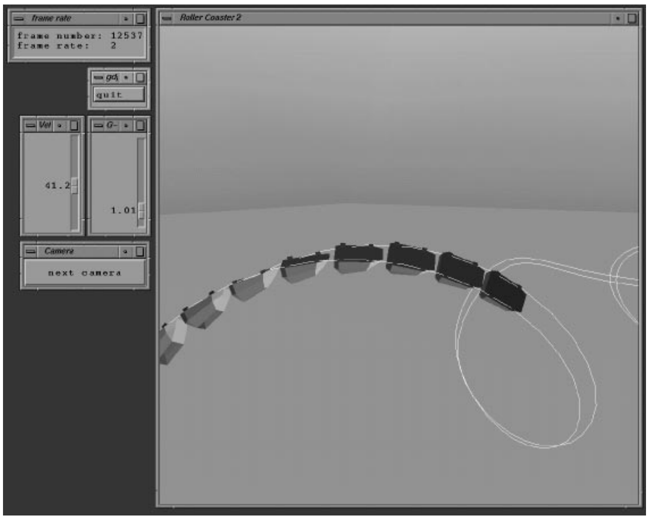
Figure 5.13: The two parts of a cart connected to the tracks

The up vector for the tracks still has to be determined however, and this is where Dynamo is very useful. Through simulation it is very easy to obtain this up vector, because the sideways orientation of the tracks has a very specific goal: to prevent large sideways forces between the tracks and the carts. Such forces would either cause the carts to fall on the inside of the track (if the tracks are too vertical), or cause the carts to be hurtled from the tracks by the centrifugal forces when going through a turn (if the tracks are too horizontal).

An initial estimate for the up direction is easily made from the curvature of the track (the second derivative is a good estimate for the direction of the centripetal forces, although their magnitude depends greatly on the velocity with which a turn is taken, which is exactly the information not taken into account for this first estimate), and the gravity vector (the gravity and centripetal forces together determine the total force that the track and the carts exert on one another). Based on this estimate, initial tracks can be constructing from the spine. A cart can then be put on the track and a ride can be simulated.

During the test ride, the direction of the total force between the tracks and the cart can be measured (it is the sum of the forces in the point-to-curve constraints, for which the constraints can be queried) when passing each of the control points in the spine, where we want to know the up directions. The track should be perpendicular to these forces, so this direction can then be used as an improved "up" direction (or "down", depending on how the forces are measured) for a next round. Between rounds, the track is re-built using the new up directions. In this way, the shape of the track can be iteratively improved. This 'learning' process is applied iteratively, since changing the tracks can affect the velocity of the cart in later parts of the ride. However, these effects are not large, and for the python track, the up directions do not change notably anymore after three iterations (three simulated rounds for the test cart).

The implementation of the example in Looks has several features in the form of feedback about the velocity and g-forces (which, for safety reasons, should not exceed certain limits, so monitoring these yields very important information about the roller coaster), and a button that can be used to switch between camera views. These are shown in Figure 5.14 (note that the shown frame rate in this figure, and some of the figures of the other examples, is very low because of the intervention of the tool that takes the snapshots).



```
frame number: 12537
frame rate:    2
```

Figure 5.14: The roller coaster example with its controls

The example itself is implemented in a modular fashion. There is a class for the carts. The `cart` class adminstrates the two `dynas` that comprise the cart, and the constraints that hold the two halves together and connect the cart to the track. The `cart` class abstracts from these implementation details by providing a method `push` which takes one real number and has as effect that a force of the given magnitude is applied in the cart's forward direction (to both cart halves). A `velocity` method returns the current magnitude of the velocity of the cart. These two methods are used as an actuator and a sensor for the PID controller that is used to tow a cart up the towing slope of the roller coaster track, and to get the cart up to speed when leaving the station (and also to slow it down again when returning to the station). A cart can determine its position by looking at the curve-parameter of the track used by the right point-to-curve constraint. This is useful for the detection of when a cart is entering the station again, or for when towing the cart up the slope has to be started or stopped. In order to support this detection, a cart has a `waitforpos(pos: real)` method, which returns only when the given position has been passed.

Carts can be combined into trains by connecting them with springs. To this end a `connectto(c:cart):void` method is added to the cart class, so a cart can connect its springs and know to which other cart it is connected. The `push` method is extended so that a cart that is pushed, also calls the push method of the cart that it is connected to: this way a whole train can be pushed at once. More than one train can be instantiated on the tracks.

The abstraction to trains is another level in the hierachical way these roller coaster example is constructed: the object oriented paradigm (the object composition and abstraction possibilities in this case) is not only usefull for the construction of Dynamo itself, but extends to the use of Dynamo in a straightforward manner.

The tracks are also represented in Looks by a class, which coordinates the calculations of the “up” vector (with help from the cart class in the form of a method which provides the current force between the cart and the track), and thereby implements the “learning process” described above.

The example is visualized through a visualizer class, which has some (hopefully interesting) viewpoints programmed into it (see Figure 5.12). A button is shown by this class to allow a user to switch between viewpoints. Furthermore two sliders are introduced that show to the user the velocity of the first cart, and the g-forces (the total of the forces exerted on the cart by its three point-to-curve constraints, divided by the magnitude of gravity) that a person would experience when sitting in the first cart. These are the sliders shown in Figure 5.14.

Some measurements were taken to determine the performance and the scaling of the performance when adding more carts. The results are depicted in Figure 5.15. These measurements were taken when running the example on a SGI O2 machine (the most low-end workstation

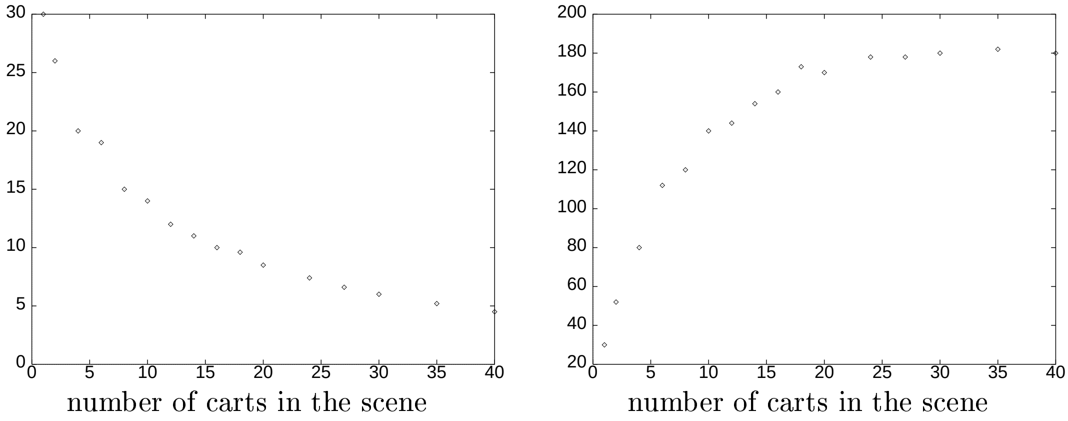


Figure 5.15: Frame rate (frames/sec), and cart rate (carts/sec)

currently on sale by SGI at the time of these measurements). The chart on the right shows that for a low number of carts, the rendering of the scene is still very much a factor in the frame rate, but as the number of carts in the scene passes twenty, the product of the number of carts in the scene (each contributing eleven degrees of freedom as was discussed before) and the frame rate is more or less a constant 180: there is a linear connection between the number of constrained degrees of freedom, and the work that has to be done to resolve the associated coupled equations (this is due to the bandwidth optimization discussed in Section 3.4.6).

Using the classes for the carts, tracks and controllers, it is very easy to describe a round of a cart/train on the roller coaster track. The train starts at the station, and has to be brought up to speed. Then it moves freely until it has to be towed up. When at the top of the towing ramp, it should start moving freely again, until entering the station again, where it has to be stopped. The Looks code for this is the following (this is part of a method of a class that also acts as the controller for the towing and accelerating and decelerating; this class has a local attribute `car` which holds a reference to the cart/train that is under its control):

```
i:=0; // count the number of rounds...
while true do
begin
  i:=i.plus(1);
  out.write("Starting with round ").write(i).
      write(" at time ").writeln(dsystem.time());
  // train is at the station with the "brakes" on:
  // start by releasing the breaks and pushing it out

  target_velocity(7); activate();
  t:=dsystem.time().plus(1);
  while dsystem.time().lt(t) do synchronize;
  deactivate();

  // now the train will roll until it has to be towed up:
  car.waitforpos(10.6);

  target_velocity(15); controller_parameters(-2500,0,100);
  activate();
  // now keep towing it up until the top of the ramp is reached
  car.waitforpos(14.5.plus(nrcarts.times(0.1)));
  deactivate();

  // now let the rollercoaster do its thing until we're back
  // at the station.
  car.waitforpos(0.995.plus(nrcarts.times(0.01)));

  // At the station, put on the brakes and keep them on for
  // enough time to let the people out and new people in:
  target_velocity(0); controller_parameters(-500,0,50);
  activate();
  t:=dsystem.time().plus(10);
  while dsystem.time().lt(t) do synchronize;

  // ready for another round!
end;
```

Simulations with this roller coaster example show that the maximum velocity of the virtual roller coaster is about 84 kilometers per hour (varying slightly depending on the number of carts), which comes very close to the 85 kilometers per hour for the actual python. The virtual roller coaster however attains its top speed before the first looping, and not after. A very likely cause of this is that the modeling of the loopings is very crude, and does not match the actual loopings precisely enough.

6 The design process

6.1 Introduction

During the design process, a problem specification is transformed info a problem solution, and the description of this solution and the ideas behind it, is the design. During the design process, design decisions show which possible solutions are chosen out of available alternatives.

Many methods and models are available for structuring and guiding the design process. The result of the design process should be software that is well structured so it can be easily used, extended and maintained. The object oriented design methodology is a good tool for creating well structured software, since it provides powerful means to create and describe structure by dividing the software into parts (classes) and describing how those modules are related (through all sorts of class relationships, see [Rumb 96]).

The object oriented design methodology also clearly distinguishes between interface and implementation. This allows for a separation of the concerns of specifying what exactly needs to be done (interface) and how it should be done (implementation).

6.2 Design phases

Figure 6.1 shows the classic waterfall software engineering life-cycle. Requirement engineering has been left out of the scope of this thesis, since the requirements (as presented in Section 2.3) were clear from the beginning. From the requirements, through specification and global and detailed design, a software structure is derived that can be implemented. Implementation details are not directly discussed in this thesis, and the same is true for testing and maintenance. However, the design of the software strongly influences testability and maintainability, so although not directly described, these phases are accounted for in the design through a constant focus on modularity and appropriate level of abstraction.

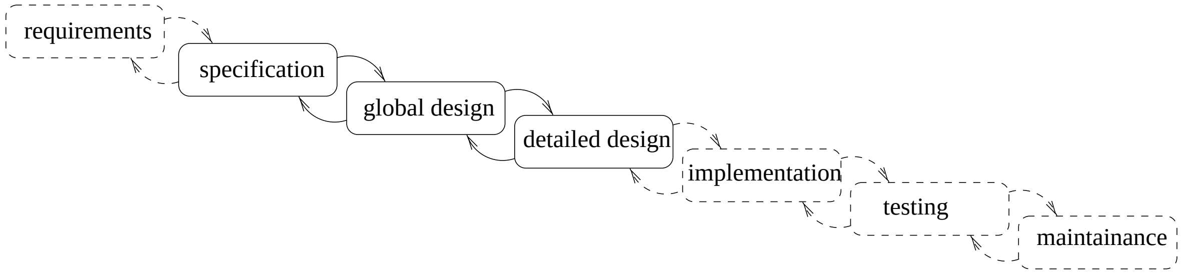


Figure 6.1: The waterfall life-cycle model

The design phase classically starts with a structural design in which the problem is subdivided into a number of smaller problems. For Dynamo this meant the distinction between the three subsystems (forward dynamics, controllers, inverse dynamics), and the classes within those subsystems. Subdivision in classes however is often already done in the specification phase (or earlier), to be able to reason about the problem. A clear specification is often not reached until the interfaces of those classes are defined. So specification and design are slightly intermixed.

In the design of Dynamo, the following phases better reflect the approach used:

1. interface design
2. algorithm design
3. code design

The interface design allows for a precise specification of the problem. In Dynamo for example, one of the central concepts is the notion of dynamically moving geometries. This can be specified by reusing the geometry specification (by inheritance) and adding methods like `applyforce` and `applytorque`. Interfaces also specify how classes can interact with each other, and therefore specify the relationships between classes. Algorithm design then aims at finding a way to provide the functionality required by the interfaces. Algorithms often involve more than one class, and deciding exactly which class is responsible for which part of the algorithm is one of the decisions to be made in the code design phase. This often leads to new notions which will have to be encapsulated in new classes and interfaces. The code design phase corresponds best to the classical detailed design phase.

The overall design process is not a sequence of phases one to three, but sometimes a designer has to go back to a previous phase to obtain a more refined solution. For example, the introduction of the `applyforce` method (with the underlying support for inertia) in the interface design phase of the `dyna` class, requires the design of an algorithm for the required integration step. Since we want to keep a part of the algorithm flexible, we decide to code this part in a separate class: the motion integrator class. This, in turn, leads to the design of the interface by which the dyna and the motion integrator class communicate.

Below, each of the three phases are treated in more detail.

6.2.1 Interface design

The interface design takes a description of the problem and results in a specification of exactly how the environment will be able to interface with the software to solve the problem. This often entails a division in classes, where each of the classes is responsible for a subproblem.

Using interfaces has many advantages:

- it separates thinking of what functionality is needed from thinking about how it should be provided.
- it separates the concerns of the user of a piece of software and the provider of that software and defines who is responsible for what.

- the classes and methods in the interface define a vocabulary for reasoning about the problem.
- it provides abstraction from (hidden) implementation details, which can be changed or optimized later without affecting the interface and therefore the user code.
- since interfaces stem from the required interaction between the classes, they aid in describing how the different modules of a piece of software interact with each other (through the specification of for example the parameter types)
- once interfaces have been defined, the application software and the implementation of the software can be developed in parallel
- the different software modules can be black-box tested to see if they adhere to their interface. Since interfaces are often small (since many details are hidden), this can result in less testing.

One of the main advantages that object orientation offers over the more traditional approach of procedural programming, is that there are many ways to specify relations between interfaces. In stead of providing software in the form of simply a collection of separate functions, a class library has much more structure, providing a better level of abstraction and a clearer insight in how the different parts of the class library interact. The overall structure of Dynamo, for example, can be shown in one diagram (see Appendix G, where the basic structure is shown, abstracting from a lot of details), while the underlying software spans some twenty thousand lines of source (some ten thousand more for the GDP wrapper). The notation used to denote the relationships between classes is the notation used in OMT (see Appendix F and [Rumb 96]).

One relationship for providing structure is inheritance. It is a powerful abstraction mechanism for specifying common interfaces between several similar classes (the constraint interface being an example of this use). It thereby promotes reuse of specification (as was seen when the `dyna` class was introduced as specialization of the geometry class). It is less suitable for reuse of implementation since inheritance provides so-call white-box reuse: when inheriting from a class which has (parts of) its methods implemented, the subclass becomes dependent on the implementation details of the superclass. Composition, where the composed classes are only accessed via their interface, is a much better mechanism for implementation re-use (which is probably one of the reasons why inheritance is not used as much).

6.2.2 Algorithm design

Once the interface has been established, it can function as a specification, determining exactly what has to be done. How it can be done is worked out in the algorithm design phase. In this phase, standard algorithms can be combined, or new algorithms devised to provide the functionality required by the interface. This sometimes leads to additional interface elements, as some algorithms require steering parameters (the maximum error attribute of the constraint manager class for example), or to specific code design where is determined which classes take responsibility for which parts of the algorithms (which in turn leads to new interface design as it has to be determined how the different classes communicate with each other).

6.2.3 Code design

The calculations described by the algorithms have to be performed by the classes. Determining in which manner an algorithm is distributed over different methods of (possibly different) classes is performed in the code design phase.

Sometimes, an algorithm inherently requires distribution over more than one class. The constraint correction algorithm is an example of this, since it involves the constraint manager, the constraints, and also the dynas. Even when the algorithm is known, it can be a non-trivial task to determine how to distribute the algorithm over the classes, and how they interface with each other. For example, when calculating $\frac{\partial C}{\partial R}$, the algorithm specifies that it can be calculated as the product of three partial derivatives (see equation 3.19). As was shown in Section 3.4.5, the manner in which the classes interface to calculate this product severely impacts extendibility and number of required methods. Another example is the friction computation as described in Section 3.5.2. The algorithm is explained in that section, but code design still has to be performed.

Often, parts of an algorithm are delegated to different (sometimes new) classes, to allow for more flexibility, or to allow for easily changing between different solutions. The introduction of the motion integrator class in Section 3.2 is an example of the introduction of a new class to allow for flexibility (in the way the forward integration is done).

6.3 Planning

The project was planned according to the requirements presented in Section 2.3. The first part of the project followed a more global planning: after familiarization with the GDP for a while, a start was made on the dynamics system, as detailed in [Baren 96]. The inverse dynamics system was added to that, but improvements and extensions to the forward dynamics subsystem were made while working on the inverse dynamics, so there was no fixed deadline for the forward dynamics subsystem. For the first part as a whole there was of course the nine-month time limit of the OOTI final project.

For the second part of the project, more precise planning was possible, because the problem description was more precise, and consisted of more individual parts which could be planned separately. This in contrast to the first part of the project, for which the requirements were more vague, and only the research done during the project revealed how the system could be decomposed into smaller parts, and what the complexity of each of those parts was. For each of the requirements for the second phase, Table 6.1 shows the estimations made at the beginning of the second part of the project.

In addition to these activities, this thesis had to be written, the on-line documentation had to be created and kept up to date, and several miscellaneous activities (presentations, meetings), most of which fall under the category professional development, had to be performed.

The activities mentioned above were executed in a more or less interleaved manner: when an idea for the solution to one problem became apparent, that topic was taken up. But each topic was tracked to ensure that not too much time was spent on it. Most of the activities were performed within their assigned time span.

Phase	**Planned duration**
Separation from the GDP	3 weeks
Writing and maintaining a user manual	5 weeks
Addition of examples	8 weeks
Addition of constraints	2 weeks
Improving collision handling	5 weeks
Modeling of friction	6 weeks
Addition of curve and surface types	5 weeks
Addition of ropes	7 weeks
Addition of controllers	4 weeks
Improving inverse dynamics calculations	6 weeks
Working specifically for TNO	10 weeks
Total	61 weeks

Table 6.1: Planning for the second part of the project

Creating the examples was an exception: it took longer, mostly because of the roller coaster example. Creating this example took longer because of the writer's inexperience with modeling (especially getting the shape of the track spine right took a lot of time).

Extra topics were introduced during the project: addition of the possibility to visualize forces (as detailed in Section 3.5.1). This cost extra time. Also, an attempt was made to use Dynamo within VRML (using the extended authoring interface), based on the Java version of the GDP. That attempt was unfortunately unsuccessful because of differences between the tools (the VRML browser plug-in, the java virtual machine, the java compiler and the C++ compiler) for the workstation platform. After some time was spent on the topic, it had to be abandoned because other topics had higher priority. The incompatibilities were specific to dynamic library formats used by the operating system on the workstation, and not caused by Dynamo. Another attempt might be made on a PC, but since the Java-GDP is not available on the PC platform that will need some additional work. This consists mainly of writing the Java wrapper for Dynamo. The manner in which to use the extended authoring interface of VRML is detailed in [Pau 98].

Because of the few delays and the extra topics there was not enough time to completely meet two of the initial requirements.

First, there was no time to fully investigate the topic of modeling of friction. The algorithms to be used are clear, and there is some indication of the manner in which the Dynamo constraints can be used, but further code design and the corresponding design of the interfaces between the classes that perform the friction calculations is needed. Once these classes have been determined, also their interface to the user (for specifying friction constants for example) can be designed.

Second, the design and implementation of extra types of curves and surfaces was not performed as extensively as originally envisioned, but the manner in which to add more new types is clear, so new types can be added later as needed.

During the project, each topic was documented separately as the topic was investigated. At the end, all the pieces (along with the OOTI report) had to be assembled. Due to the timing constraints during the project, this assembling was finished afterwards.

7 Conclusions and future work

7.1 Conclusions

Using the (inverse) dynamics engine described in this thesis, it is possible to provide “natural looking” animations at speeds that are sufficiently fast for interactive use by performing simulations of the mechanical systems in an animation.

In the forward dynamics subsystem a user can exchange precision for speed by choosing an appropriate motion integrator and the step size taken in each simulation step.

Controllers can be used to actively steer the motions of the dynas that are in a scene and controller forces are explicitly available, allowing insight in the forces that are required to obtain the prescribed motion.

The inverse dynamics subsystem provides constraints which allow a user to connect several dynas in a variety of ways. Constraints can be non-holonomic and can be mixed freely, since the constraint correction algorithms allow for loops in the constraint configurations. Even so, computation time is linear in the number of constraints in many cases due to several optimizations (it is quadratic in worst case).

Since the constraints are solved by applying constraint forces, constraints that restrict few degrees of freedom in the relative motions are cheaper to calculate than constraints that restrict many. The explicit calculation of the constraint forces can provide insight in the dynamics of the animated system, since the constraint forces are available and can therefore be used directly in monitoring the system. Systems of dynas that are connected by constraints often show emergent behavior which can be studied using the simulation, and which provides for interesting looking animations.

Constraints and controllers can also be used to specify relationships between dynas and geometries of which the motion is obtained in another manner (kinematically, or through motion capture for example). Being able to freely combine dynamics with other motion specification techniques provides an even more powerful tool.

A lot of attention has been paid to the design of the software that implements these simulation algorithms. Object orientation helps in structuring the software and allows for separate focusing on the interface of the software on the one hand, and the internal calculations on the other. The separate attention to the interface design helps in creating interfaces that provide a higher level of abstraction and are thereby easier to use by a programmer who needs to access the functionality provided by Dynamo.

Extendibility of the software is obtained through abstract base classes that allow the existing software to interface with yet to be developed components, as well as through interfaces that provide enough abstraction from the internal operations of the classes to map very well to the ‘natural interfaces’ of the objects that are simulated.

7.2 Future work

7.2.1 Extensions

Since Dynamo is easily extendible, many additions are possible.

Since Dynamo has been made available on the world wide web (in March 1999), most reactions have been about the collision handling and the addition of friction. So these should be of top priority.

For constraint handling several issues were presented at the end of Section 4.1.14, and these should be resolved. Research in this area will be greatly facilitated if Dynamo is first integrated in a host system which provides a suitable collision detector (or if a collision detector is built into the GDP, which currently already functions as a host system for Dynamo).

For friction handling, the algorithm design is presented in Section 3.5.2, but proper code design needs to be performed to allow friction to be added to contact constraints.

Next to friction, more constraints can be added as specialization of the generic `constraint` interface. Dynamo also provides generic `curve` and `surface` classes, and the library of actual specializations of these classes can be extended to include more types of curves and surface representations (especially representations fitted for use in an interactive environment, so that they can easily be edited).

Dynamo interfaces with a user at programming level and a host system is required for its use. Therefore Dynamo should be incorporated in more host systems, which can provide an interactive environment (maybe based on the Animo system described in [Thom 96], or on the Vission system described in [Telea 99m]) and can provide a more intuitive and faster user interaction. Such a host system could also incorporate more aspects of the modeling phase of making an animation or simulation (see [Barz 88]).

The design and implementation of more controllers is another field of interest. While Dynamo offers very basic controllers, more advanced controllers such as those described in [Panne 95] can be used to simulate more complex systems. These advanced controllers can make use of the basic controllers as their actuators by providing their actuator values to the basic controllers as steering values.

If sufficient computing power is available, the dynamics of non-rigid dynas could be simulated. Such non-rigid dynas require a specialization of the current `dyna` class which, for example, re-implements the `applyforce` method so that the energy of a force is not only used for a change in its motion, but also for a change in its shape. Note that this does not affect the inverse dynamics and controller subsystems: only the `dyna` class needs to be specialized, next to a re-implementation of the callbacks that are used to communicate between Dynamo and the host system, to also exchange the information about the deformations.

Another welcome addition would be an event mechanism that would allow Dynamo to inform the host system of internal events such as the breaking of constraints (because their maximum constraint force magnitude was exceeded), or switching to a different solver method (because of redundant constraints). Simply adding a callback function for each event is not an elegant solution, since there is a lot of structure in the kinds of events. There are for example several constraints which can deactivate themselves, and events signifying this could differ in small details (whether for example the constraint was deactivated because the maximum force was exceed, or whether it was deactivated because of other factors such as a point-to-curve constraint where the curve parameter gets out of its domain). Inheritance could be used in this case to bring this structure into the software components: a code design phase is still required to incorporate an event mechanism into Dynamo.

7.2.2 Internal changes

As simulations become more complicated, the computational power of a single computer might not suffice anymore, and parallelizing the algorithms described here also becomes a topic of interest. Different `dynas` can integrate their motion equations in parallel, and algorithms such as described in [Ames 99] can be used to speed up the global matrix calculations.

Another topic of study could be towards the further reduction of the required computational cost. The computational cost of the inversion of $\frac{\partial C}{\partial R}$ might be reduced if one can exploit the fact that $\frac{\partial C}{\partial R}$ as well as its inverse from the previous frame are already known, using something like the Sherman-Morrison formula (see [Press 92]). Investigating whether this formula is applicable, and if the resulting jacobians are accurate enough, would be a point of interest.

When constraints are added, a brief analysis of the constraint configuration is made by the traversal of the constraint graph that leads to the CutHillMcKee ordering (see Section 3.4.6). During this traversal, it is detected if the constraint graph consists of several disconnected subgraphs. Constraints in the roller coaster example for instance only influence the other constraints within the same cart, and not those of other carts (since the carts are connected by springs, and not by constraints). So the complete constraint graph for the roller coaster example consists of disjoint subgraphs which each represent the constraints of one cart. Design decision 15 could therefore be refined to allow separate grouping of the constraints in each disjoint subgraph. The `satisfy_constraint` algorithm presented on page 41 could then be applied to each of these groups of constraints individually, allowing some constraint groups to be solved in fewer iterations, or with a faster solver method than could be used in the current approach (where a redundancy in one of the subgraphs would switch to a slower solver method for the complete constraint problem). The worst case time complexity of the constraint correction would be improved by this approach from the square of the sum of the number of constraints in each of the subgraphs, to the sum of the squares of the number of constraints in each of the subgraphs.

These are just some possible future fields of interest, and further research in these areas will undoubtedly reveal new possibilities and challenges.

Appendix A Dyna's partial derivatives

A.1 Introduction

To be able to analytically determine the $\frac{\partial C}{\partial R}$ matrix (see Section 3.4.5, the dyna class will have to offer support in the form of methods which calculate the partial derivatives indicating how the (motion dependent) properties of the dyna change as a result of a change in the forces, torques and impulses that govern the motion of the dyna. For now, the properties that are supported for use in a constraint are:

1. the position in world coordinates of a fixed point on the dyna (given by a set of local coordinates). This property corresponds to the variants of the `to_world` method (with a point as parameter) of the dyna class.
2. the velocity in world coordinates of a fixed point on the dyna (given by a set of local coordinates). This property corresponds to the variants of the `get_velocity` method (with a point as parameter) of the dyna class.
3. the direction in world coordinates of a fixed vector in the dyna (given by a set of local coordinates). This property corresponds to the variants of the `to_world` method (with a vector as parameter) of the dyna class.
4. the change in direction in world coordinates of a fixed vector in the dyna (given by a set of local coordinates). This property corresponds to the variants of the `get_velocity` method (with a vector as parameter) of the dyna class.

These four properties (later perhaps extended with properties like the potential and/or kinetic energy of the dyna) can be influenced by constraints at time stamps t^+ and $t+h$, so we need partial derivatives for these two time stamps.

Table A.1 below shows all combinations of properties of a dyna, and ways to change the motion of a dyna. The names in the table refer to the names of the methods which calculate

	p_{t+h}	$\dot{p}_{t+h}$	d_{t+h}	$\dot{d}_{t+h}$	p_{t^+}	$\dot{p}_{t^+}$	d_{t^+}	$\dot{d}_{t^+}$
applycenterforce	dpdF	ddpdF	0	0	0	0	0	0
applytorque	dpdM	ddpdM	dvdM	ddvdM	0	0	0	0
applyforce	dpdfq	ddpdfq	dvdfq	ddvdfq	0	0	0	0
applyimpulse	dpdi	ddpdi	dvdi	ddvdi	0	ddptdi	0	ddvtdi

Figure A.1: The method names for dyna's analytical inverse dynamics support

the required partial derivatives, and a 0 indicates that a no such method is required since the corresponding partial derivative is zero.

In the following sections the expressions for each of the partial derivatives will be determined, by differentiating the motion equations presented in Section 3.2. A motion integrator is required. Since we are only looking for a first-order approximation for the relationships between the various variables, using the integration equations of the Euler integrator (with the modified `ode` method presented in Appendix B) suffices.

A.2 Partial derivatives for central forces

A.2.1 Determining $\frac{\partial p_{t+h}}{\partial F_t}$ analytically

The position in world coordinates of a point (given in local coordinates) is calculated as:

$$p_\tau = z_\tau + A_\tau \rho \tag{A.1}$$

Since central forces do not affect the orientation of a geometry, we have:

$$\frac{\partial p_{t+h}}{\partial F_t} = \frac{\partial p_{t+h}}{\partial z_{t+h}} \frac{\partial z_{t+h}}{\partial F_t}$$

and from equation A.1 and integration equation $z_{t+h} = z_t + v_t h + \frac{1}{2}\frac{F}{m}h^2$ we can see that

$$\frac{\partial p_{t+h}}{\partial z_{t+h}} = 1 \quad \text{and} \quad \frac{\partial z_{t+h}}{\partial F_t} = \frac{h^2}{2m}$$

Combining these equations yields

$$\frac{\partial p_{t+h}}{\partial F_t} = \frac{h^2}{2m} \tag{A.2}$$

A.2.2 Determining $\frac{\partial \dot{p}_{t+h}}{\partial F_t}$ analytically

The velocity in world coordinates of a point given in local coordinates is calculated as:

$$\dot{p}_\tau = v_\tau + \omega_\tau \times A_\tau \rho \tag{A.3}$$

Similar to the derivation in the previous section, we can see that

$$\frac{\partial \dot{p}_{t+h}}{\partial F_t} = \frac{\partial \dot{p}_{t+h}}{\partial v_{t+h}} \frac{\partial v_{t+h}}{\partial F_t}$$

where (from equation A.3 and $v_{t+h} = v_t + \frac{F_t}{m}h$):

$$\frac{\partial \dot{p}_{t+h}}{\partial v_{t+h}} = 1 \quad \text{and} \quad \frac{\partial v_{t+h}}{\partial F_t} = \frac{h}{m}$$

yielding:

$$\frac{\partial \dot{p}_{t+h}}{\partial F_t} = \frac{h}{m} \tag{A.4}$$

A.3 Partial derivatives for torques

A.3.1 Determining $\frac{\partial p_{t+h}}{\partial M_t}$ analytically

Equation A.1 can be rewritten as

$$p_\tau = z_\tau + A_\tau \rho = z_\tau + \sum_{i=0}^{2} \rho_i A_{i\tau} \tag{A.5}$$

showing that

$$\frac{\partial p_{t+h}}{\partial M_t} = \sum_{i=0}^{2} \frac{\partial p_{t+h}}{\partial A_{it+h}} \frac{\partial A_{it+h}}{\partial M_t} \tag{A.6}$$

and from equation A.5 we can see that

$$\frac{\partial p_{t+h}}{\partial A_{it+h}} = \rho_i \tag{A.7}$$

Matrices $\frac{\partial A_{it+h}}{\partial M_t}$ are derived from the integration equations for the orientation (which we will assume to be integrated using the Euler integrator with the modified `ode` method discussed in Appendix B). We have:

$$\frac{\partial A_{it+h}}{\partial M_t} = \frac{\partial A_{it+h}}{\partial q_{t+h}} \frac{\partial q_{t+h}}{\partial \dot{q}_\gamma} \frac{\partial \dot{q}_\gamma}{\partial \dot{\omega}_t} \frac{\partial \dot{\omega}_t}{\partial M_t} \tag{A.8}$$

The $\frac{\partial A_{it+h}}{\partial q_{t+h}}$ can be derived from equation 3.1 yielding:

$$\frac{\partial A_{0t+h}}{\partial q_{t+h}} = 2 \begin{pmatrix} 2q_0 & 2q_1 & 0 & 0 \\ -q_3 & q_2 & q_1 & -q_0 \\ q_2 & q_3 & q_0 & q_1 \end{pmatrix} \tag{A.9}$$

$$\frac{\partial A_{1t+h}}{\partial q_{t+h}} = 2 \begin{pmatrix} q_3 & q_2 & q_1 & q_0 \\ 2q_0 & 0 & 2q_2 & 0 \\ -q_1 & -q_0 & q_3 & q_2 \end{pmatrix} \tag{A.10}$$

$$\frac{\partial A_{2t+h}}{\partial q_{t+h}} = 2 \begin{pmatrix} -q_2 & q_3 & -q_0 & q_1 \\ q_1 & q_0 & q_3 & q_2 \\ 2q_0 & 0 & 0 & q_3 \end{pmatrix} \tag{A.11}$$

Matrix $\frac{\partial q_{t+h}}{\partial \dot{q}_\gamma}$ is derived from Euler's integration equation $q_{t+h} = q_t + \dot{q}_\gamma h$ yielding

$$\frac{\partial q_{t+h}}{\partial \dot{q}_\gamma} = h \tag{A.12}$$

Derivative $\frac{\partial \dot{q}_\gamma}{\partial \dot{\omega}_t}$ stems from the modified equation (see Appendix B) for $\dot{q}$ (being $\dot{q}_\gamma = (\omega_t + \frac{h}{2}\dot{\omega}_t) \star q_t$), resulting (using the $\star$-operator defined on page 14) in:

$$\frac{\partial \dot{q}_\gamma}{\partial \dot{\omega}_t} = \frac{h}{4} \begin{pmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{pmatrix} \tag{A.13}$$

And finally, differentiating equation $\dot{\omega}_\tau = A_\tau J^{-1} A_\tau^T (M_\tau - \omega_\tau \times A_\tau J A_\tau^T)$ yields:

$$\frac{\partial \dot{\omega}_t}{\partial M_t} = A_t J^{-1} A_t^T \tag{A.14}$$

Several of these result can be combined[1] yielding:

$$\frac{\partial A_{0t+h}}{\partial \dot{\omega}_t} = h^2 \begin{pmatrix} 0 & -(q_0q_2 + q_1q_3) & -(q_0q_3 - q_1q_2) \\ q_0q_2 + q_1q_3 & 0 & -\frac{1}{2}(q_0^2 + q_1^2 - q_2^2 - q_3^2) \\ q_0q_3 - q_1q_2 & \frac{1}{2}(q_0^2 + q_1^2 - q_2^2 - q_3^2) & 0 \end{pmatrix} \tag{A.15}$$

$$\frac{\partial A_{1t+h}}{\partial \dot{\omega}_t} = h^2 \begin{pmatrix} 0 & -(q_2q_3 - q_0q_1) & -\frac{1}{2}(q_1^2 + q_3^2 - q_0^2 - q_2^2) \\ q_2q_3 - q_0q_1 & 0 & -(q_0q_3 + q_1q_2) \\ \frac{1}{2}(q_1^2 + q_3^2 - q_0^2 - q_2^2) & q_0q_3 + q_1q_2 & 0 \end{pmatrix} \tag{A.16}$$

$$\frac{\partial A_{2t+h}}{\partial \dot{\omega}_t} = h^2 \begin{pmatrix} 0 & -\frac{1}{2}(q_1^2 + q_2^2 - q_0^2 - q_3^2) & q_0q_1 + q_2q_3 \\ \frac{1}{2}(q_1^2 + q_2^2 - q_0^2 - q_3^2) & 0 & -(q_0q_2 + q_1q_3) \\ -(q_0q_1 + q_2q_3) & q_0q_2 + q_1q_3 & 0 \end{pmatrix} \tag{A.17}$$

which are all anti-symmetrical matrices: only three of the nine elements of these matrices will have to be calculated and stored when manipulating with them. Furthermore, combining these results with equations A.6, A.7, and A.8 yields:

$$\frac{\partial p_{t+h}}{\partial M_t} = \left(\sum_{i=0}^{2} \rho_i \frac{\partial A_{it+h}}{\partial \dot{\omega}_t} \right) \frac{\partial \dot{\omega}_t}{\partial M_t} \tag{A.18}$$

where the complete summation is still an anti-symmetrical matrix (which is then multiplied by the (symmetrical) $A_t J^{-1} A_t^T$).

A.3.2 Determining $\frac{\partial \dot{p}_{t+h}}{\partial M_t}$ analytically

The equation for the velocity of a point can be rewritten:

$$\begin{aligned} \dot{p}_\tau &= v_\tau + \omega_\tau \times A_\tau \rho \\ &= v_\tau + \omega_\tau \times \sum_i \rho_i A_{i\tau} \\ &= v_\tau + \sum_i \rho_i (\omega_\tau \times A_{i\tau}) \end{aligned} \tag{A.19}$$

As a torque only affects orientation, we can see that

$$\frac{\partial \dot{p}_{t+h}}{\partial M_t} = \sum_{i=0}^{2} \frac{\partial \dot{p}_{t+h}}{\partial(\omega_{t+h} \times A_{it+h})} \frac{\partial(\omega_{t+h} \times A_{it+h})}{\partial M_t} \tag{A.20}$$

---

[1] only if we take the q in equations A.9–A.11 at time stamp t (just as the q in equation A.13) instead of at their proper time stamp $t + h$. This is however warranted since a matrix which closely resembles the analytical $\frac{\partial C}{\partial R}$ also ensures a rapid convergence of the pseudo-Newton inverse dynamics iteration, and the combination results in matrices which can be calculated and manipulated with far less computational effort than the analytically correct matrices.

where equation A.19 shows that

$$\frac{\partial \dot{p}_{t+h}}{\partial(\omega_{t+h} \times A_{it+h})} = \rho_i \tag{A.21}$$

We can calculate $\frac{\partial(\omega_{t+h} \times A_{it+h})}{\partial M_t}$ as

$$\frac{\partial(\omega_{t+h} \times A_{it+h})}{\partial M_t} = \frac{\partial(\omega_{t+h} \times A_{it+h})}{\partial \omega_{t+h}} \frac{\partial \omega_{t+h}}{\partial M_t} + \frac{\partial(\omega_{t+h} \times A_{it+h})}{\partial A_{it+h}} \frac{\partial A_{it+h}}{\partial M_t} \tag{A.22}$$

where

$$\frac{\partial(\omega_{t+h} \times A_{it+h})}{\partial \omega_{t+h}} = -\tilde{A}_{it+h} \quad \text{and} \quad \frac{\partial(\omega_{t+h} \times A_{it+h})}{\partial A_{it+h}} = \tilde{\omega}_{t+h} \tag{A.23}$$

Just like we approximated q_{t+h} in equations A.9–A.11 by q_t, we will here approximate $\tilde{A}_{it+h}$ with $\tilde{A}_{it}$ and $\tilde{\omega}_{t+h}$ with $\tilde{\omega}_t$. We already have the expression for $\frac{\partial A_{it+h}}{\partial M_t}$ (see equation A.8), and from

$$\frac{\partial \omega_{t+h}}{\partial M_t} = \frac{\partial \omega_{t+h}}{\partial \dot{\omega}_t} \frac{\partial \dot{\omega}_t}{\partial M_t} \quad \text{and} \quad \omega_{t+h} = \omega_t + \dot{\omega}_t h \tag{A.24}$$

we can see that:

$$\frac{\partial \omega_{t+h}}{\partial M_t} = h \frac{\partial \dot{\omega}_t}{\partial M_t} \tag{A.25}$$

Since both terms in equation A.22 now share derivative $\frac{\partial \dot{\omega}_t}{\partial M_t}$ we can write:

$$\frac{\partial(\omega_{t+h} \times A_{it+h})}{\partial M_t} = (\tilde{\omega}_t \frac{\partial A_{it+h}}{\partial \dot{\omega}_t} - \tilde{A}_{it} h) \frac{\partial \dot{\omega}_t}{\partial M_t} \tag{A.26}$$

and therefore

$$\begin{aligned} \frac{\partial \dot{p}_{t+h}}{\partial M_t} &= \sum_{i=0}^{2} \rho_i \left((\tilde{\omega}_t \frac{\partial A_{it+h}}{\partial \dot{\omega}_t} - \tilde{A}_{it} h) \frac{\partial \dot{\omega}_t}{\partial M_t} \right) \\ &= \left(\sum_{i=0}^{2} \rho_i (\tilde{\omega}_t \frac{\partial A_{it+h}}{\partial \dot{\omega}_t} - \tilde{A}_{it} h) \right) \frac{\partial \dot{\omega}_t}{\partial M_t} \end{aligned} \tag{A.27}$$

A.3.3 Determining $\frac{\partial d_{t+h}}{\partial M_t}$ analytically

Derivative $\frac{\partial d_{t+h}}{\partial M_t}$ is based on equation

$$d_\tau = A_\tau \psi = \sum_{i=0}^{2} \psi_i A_{i\tau} \tag{A.28}$$

(where ψ are the local coordinates of vector d, like the ρ used for position). This yields:

$$\frac{\partial d_{t+h}}{\partial M_t} = \sum_{i=0}^{2} \psi_i \frac{\partial A_{it+h}}{\partial M_t} \tag{A.29}$$

which can be combined with the results for equation A.8.

A.3.4 Determining $\frac{\partial \dot{d}_{t+h}}{\partial M_t}$ analytically

Derivative $\frac{\partial \dot{d}_{t+h}}{\partial M_t}$ is based on equation:

$$\dot{d} \;=\; \omega_\tau \times A_\tau \psi = \omega_\tau \times \sum_{i=0}^{2} A_{i\tau}\psi_i = \sum_{i=0}^{2} \omega_\tau \times A_{i\tau}\psi_i = \sum_{i=0}^{2} \psi_i(\omega_\tau \times A_{i\tau}) \tag{A.30}$$

Therefore:

$$\frac{\partial \dot{d}_{t+h}}{\partial M_t} = \sum_{i=0}^{2} \psi_i \frac{\partial(\omega_{t+h} \times A_{it+h})}{\partial M_t} \tag{A.31}$$

for which the results from equation A.22 can be used.

A.4 Partial derivatives for (non-central) forces

A.4.1 Determining $\frac{\partial p_{t+h}}{\partial f_t^{p'}}$ analytically

Each non-central force $f^{p'}$ is first split in a central force F (affecting only the position of the geometry) and a torque M (affect only the orientation of the geometry), yielding:

$$\frac{\partial p_{t+h}}{\partial f_t^{p'}} = \frac{\partial p_{t+h}}{\partial F_t}\frac{\partial F_t}{\partial f_t^{p'}} + \frac{\partial p_{t+h}}{\partial M_t}\frac{\partial M_t}{\partial f_t^{p'}} \tag{A.32}$$

From $F_\tau = f_\tau^{p'}$ and $M_\tau = (A_\tau \rho') \times f_\tau^{p'}$ we can see that:

$$\frac{\partial F_t}{\partial f_t^{p'}} = 1 \quad \text{and} \quad \frac{\partial M_t}{\partial f_t^{p'}} = \tilde{A_t \rho'} \tag{A.33}$$

which can be combined with the results presented in sections A.2.1 and A.3.1.

A.4.2 Determining $\frac{\partial \dot{p}_{t+h}}{\partial f_t^{p'}}$ analytically

Using the same approach as in the previous section, we can see that

$$\frac{\partial \dot{p}_{t+h}}{\partial f_t^{p'}} = \frac{\partial \dot{p}_{t+h}}{\partial F_t}\frac{\partial F_t}{\partial f_t^{p'}} + \frac{\partial \dot{p}_{t+h}}{\partial M_t}\frac{\partial M_t}{\partial f_t^{p'}} \tag{A.34}$$

so we can use the earlier results from sections A.2.2 and A.3.2 and equation A.33.

A.4.3 Determining $\frac{\partial d_{t+h}}{\partial f_t^{p'}}$ analytically

Using the same approach as in the previous sections, we can see that

$$\frac{\partial d_{t+h}}{\partial f_t^{p'}} = \frac{\partial d_{t+h}}{\partial F_t}\frac{\partial F_t}{\partial f_t^{p'}} + \frac{\partial d_{t+h}}{\partial M_t}\frac{\partial M_t}{\partial f_t^{p'}} \tag{A.35}$$

and since $\frac{\partial d_{t+h}}{\partial F_t} = 0$ we can use the earlier results from Section A.3.3 and equation A.33.

A.4.4 Determining $\frac{\partial \dot{d}_{t+h}}{\partial f_t^{p'}}$ analytically

Using the same approach as in the previous sections, we can see that

$$\frac{\partial \dot{d}_{t+h}}{\partial f_t^{p'}} = \frac{\partial \dot{d}_{t+h}}{\partial F_t}\frac{\partial F_t}{\partial f_t^{p'}} + \frac{\partial \dot{d}_{t+h}}{\partial M_t}\frac{\partial M_t}{\partial f_t^{p'}} \tag{A.36}$$

where $\frac{\partial \dot{d}_{t+h}}{\partial F_t} = 0$. So we can use the earlier results from Section A.3.4 and equation A.33.

A.5 Partial derivatives for impulses

A.5.1 Determining $\frac{\partial \dot{p}_{t+}}{\partial \iota_t}$ analytically

We have:

$$\begin{aligned}
\frac{\partial \dot{p}_{t+}}{\partial \iota_t} &= \frac{\partial(v_{t+} + (\omega_{t+} \times A_t\rho_p))}{\partial \iota_t} \\
&= \frac{\partial((v_{t-} + \Delta v_t) + ((\omega_{t-} + \Delta\omega_t) \times A_t\rho_p))}{\partial \iota_t} \\
&= \frac{\partial((v_{t-} + \Delta v_t) + (\omega_{t-} \times A_t\rho_p) + (\Delta\omega_t \times A_t\rho_p))}{\partial \iota_t} \\
&= \frac{\partial(\Delta v_t + (\Delta\omega_t \times A_t\rho_p))}{\partial \iota_t} \\
&= \frac{\partial \Delta v_t}{\partial \iota_t} + \frac{\partial(-(\widetilde{A_t\rho_p})\Delta\omega_t)}{\partial \iota_t} \\
&= \frac{\partial \Delta v_t}{\partial \iota_t} - (\widetilde{A_t\rho_p})\frac{\partial \Delta\omega_t}{\partial \iota_t}
\end{aligned}$$

From equation 3.7 we can see that

$$\frac{\partial \Delta v_t}{\partial \iota_t} = \frac{1}{m}$$

Equation 3.8, for application of an impulse to point q, can be rewritten as:

$$\begin{aligned}
\Delta\omega &= J_w^{-1}(r_q \times \iota) \\
&= J_w^{-1}(A\rho_q \times \iota) \\
&= J_w^{-1}(\widetilde{A\rho_q})\iota \\
&= J_w^{-1}A\tilde{\rho}_q A^T\iota \\
&= AJ^{-1}A^T A\tilde{\rho}_q A^T\iota \\
&= AJ^{-1}\tilde{\rho}_q A^T\iota
\end{aligned}$$

showing that

$$\frac{\partial \Delta\omega_t}{\partial \iota_t} = A_t J^{-1} \tilde{\rho}_q A_t^T \tag{A.37}$$

which, when combined with the previous results from this section, yields:

$$\begin{aligned}
\frac{\partial \dot{p}_{t+}}{\partial \iota_t} &= \frac{1}{m} - (\widetilde{A_t \rho_p}) A_t J^{-1} \tilde{\rho}_q A_t^T \\
&= \frac{1}{m} - A_t \tilde{\rho}_p A_t^T A_t J^{-1} \tilde{\rho}_q A_t^T \\
&= \frac{1}{m} - A_t \tilde{\rho}_p J^{-1} \tilde{\rho}_q A_t^T
\end{aligned}$$

A.5.2 Determining $\frac{\partial \dot{d}_{t+}}{\partial \iota_t}$ analytically

Using the results from the previous section, and $\dot{d} = \dot{p} - v$, we can see that:

$$\frac{\partial \dot{d}_{t+}}{\partial \iota_t} = -A_t \tilde{\rho}_p J^{-1} \tilde{\rho}_q A_t^T$$

A.5.3 Determining $\frac{\partial p_{t+h}}{\partial \iota_t}$ analytically

From $p_{t+h} = z_{t+h} + A_{t+h}\rho_p$ we can see that $\dfrac{\partial p_{t+h}}{\partial \iota_t} = \dfrac{\partial z_{t+h}}{\partial \iota_t} + \dfrac{\partial (A_{t+h}\rho_p)}{\partial \iota_t}$.

We have:

$$\begin{aligned}
\frac{\partial z_{t+h}}{\partial \iota_t} &= \frac{\partial (z_{t+} + v_{t+}h + \frac{F}{2m}h^2)}{\partial \iota_t} \\
&= \frac{\partial (v_{t+}h)}{\partial \iota_t} \\
&= \frac{\partial ((v_{t-} + \Delta v_t)h)}{\partial \iota_t} \\
&= h\frac{\partial \Delta v_t}{\partial \iota_t} \\
&= \frac{h}{m}
\end{aligned}$$

And also:

$$\begin{aligned}
\frac{\partial (A_{t+h}\rho_p)}{\partial \iota_t} &= \frac{\partial \left(\sum\limits_{j=0}^{2} (\rho_{p_j} A_{j_{t+h}}) \right)}{\partial \iota_t} \\
&= \sum_{j=0}^{2} \left(\rho_{p_j} \frac{\partial A_{j_{t+h}}}{\partial \iota_t} \right) \\
&= \sum_{j=0}^{2} \left(\rho_{p_j} \frac{\partial A_{j_{t+h}}}{\partial \omega_{t+}} \frac{\partial \omega_{t+}}{\partial \iota_t} \right)
\end{aligned}$$

$$\begin{aligned}
&= \sum_{j=0}^{2}\left(\rho_{pj}\frac{\partial A_{j\,t+h}}{\partial \omega_{t^+}}\right)\frac{\partial \omega_{t^+}}{\partial \iota_t} \\
&= \sum_{j=0}^{2}\left(\rho_{pj}\frac{\partial A_{j\,t+h}}{\partial \omega_{t^+}}\right)\frac{\partial (\omega_{t^-}+\Delta\omega_t)}{\partial \iota_t} \\
&= \sum_{j=0}^{2}\left(\rho_{pj}\frac{\partial A_{j\,t+h}}{\partial \omega_{t^+}}\right)\frac{\partial \Delta\omega_t}{\partial \iota_t} \qquad \text{(A.38)}
\end{aligned}$$

We can follow the same steps as in Section A.3.1 to see that

$$\frac{\partial A_{it+h}}{\partial \omega_{t^+}} = \frac{\partial A_{it+h}}{\partial \dot{q}_\gamma}\frac{\partial \dot{q}_\gamma}{\partial \omega_{t^+}}$$

In that section, we used $\dot{q}_\gamma = (\omega_{t^+} + \frac{h}{2}\dot{\omega}_t)\star q_t$ to arrive at $\frac{\partial \dot{q}_\gamma}{\partial \dot{\omega}_t}$ by differentiating the $\star$-operator and multiplying the result with $\frac{h}{2}$. To determine $\frac{\partial \dot{q}_\gamma}{\partial \omega_{t^+}}$, we once again have to differentiate the $\star$-operator, but now use the result without the extra factor $\frac{h}{2}$. This means that $\frac{\partial \dot{q}_\gamma}{\partial \omega_{t^+}} = \frac{2}{h}\frac{\partial \dot{q}_\gamma}{\partial \dot{\omega}_t}$. This means that:

$$\begin{aligned}
\frac{\partial A_{j\,t+h}}{\partial \omega_{t^+}} &= \frac{\partial A_{j\,t+h}}{\partial \dot{q}_\gamma}\frac{\partial \dot{q}_\gamma}{\partial \omega_{t^+}} \\
&= \frac{\partial A_{j\,t+h}}{\partial \dot{q}_\gamma}\frac{2}{h}\frac{\partial \dot{q}_\gamma}{\partial \dot{\omega}_t} \\
&= \frac{2}{h}\frac{\partial A_{j\,t+h}}{\partial \dot{q}_\gamma}\frac{\partial \dot{q}_\gamma}{\partial \dot{\omega}_t} \\
&= \frac{2}{h}\frac{\partial A_{j\,t+h}}{\partial \dot{\omega}_t} \qquad \text{(A.39)}
\end{aligned}$$

This means we can share calculations, since $\frac{\partial A_{it+h}}{\partial \dot{\omega}_t}$ is a partial derivative which we decided to cache (see again Section 3.4.5).

Combining equations A.38, A.39 and A.37 yields:

$$\frac{\partial (A_{t+h}\rho_p)}{\partial \iota_t} = \sum_{j=0}^{2}\left(\frac{2\rho_{pj}}{h}\frac{\partial A_{j\,t+h}}{\partial \dot{\omega}_t}\right)A_t J^{-1}\tilde{\rho}_q A_t^T$$

and therefore:

$$\frac{\partial p_{t+h}}{\partial \iota_t} = \frac{h}{m} + \frac{2}{h}\sum_{j=0}^{2}\left(\rho_{pj}\frac{\partial A_{j\,t+h}}{\partial \dot{\omega}_t}\right)A_t J^{-1}\tilde{\rho}_q A_t^T$$

A.5.4 Determining $\frac{\partial d_{t+h}}{\partial \iota_t}$ analytically

Using $d = A\rho = p - z$ and the results of the previous section we can conclude that:

$$\frac{\partial d_{t+h}}{\partial \iota_t} = \frac{2}{h}\sum_{j=0}^{2}\left(\rho_{p_j}\frac{\partial A_{j_{t+h}}}{\partial \dot{\omega}_t}\right) A_t J^{-1}\tilde{\rho_q} A_t^T$$

A.5.5 Determining $\frac{\partial \dot{p}_{t+h}}{\partial \iota_t}$ analytically

We have:

$$\begin{aligned}
\frac{\partial \dot{p}_{t+h}}{\partial \iota_t} &= \frac{\partial(v_{t+h} + (\omega_{t+h} \times A_{t+h}\rho_p))}{\partial \iota_t} \\
&= \frac{\partial v_{t+h}}{\partial \iota_t} + \frac{\partial(\omega_{t+h} \times A_{t+h}\rho_p)}{\partial \iota_t} \\
&= \frac{\partial(v_{t+} + \frac{F}{m}h)}{\partial \iota_t} + \frac{\partial(\omega_{t+h} \times A_{t+h}\rho_p)}{\partial \iota_t} \\
&= \frac{\partial v_{t+}}{\partial \iota_t} + \tilde{\omega}_{t+h}\frac{\partial A_{t+h}\rho_p}{\partial \iota_t} - \widetilde{(A_{t+h}\rho_p)}\frac{\partial \omega_{t+h}}{\partial \iota_t} \\
&= \frac{\partial(v_{t-} + \Delta v_t)}{\partial \iota_t} + \tilde{\omega}_{t+h}\frac{\partial\left(\sum_{j=0}^{2}\left(\rho_{p_j}A_{j_{t+h}}\right)\right)}{\partial \iota_t} - \widetilde{(A_{t+h}\rho_p)}\frac{\partial \omega_{t+h}}{\partial \iota_t} \\
&= \frac{\partial \Delta v_t}{\partial \iota_t} + \tilde{\omega}_{t+h}\sum_{j=0}^{2}\left(\rho_{p_j}\frac{\partial A_{j_{t+h}}}{\partial \iota_t}\right) - \widetilde{(A_{t+h}\rho_p)}\frac{\partial \omega_{t+h}}{\partial \iota_t} \\
&= \frac{1}{m} + \tilde{\omega}_{t+h}\sum_{j=0}^{2}\left(\rho_{p_j}\frac{\partial A_{j_{t+h}}}{\partial \omega_{t+}}\frac{\partial \omega_{t+}}{\partial \iota_t}\right) - \widetilde{(A_{t+h}\rho_p)}\frac{\partial(\omega_{t+} + \dot{\omega}_t h)}{\partial \iota_t} \\
&= \frac{1}{m} + \tilde{\omega}_{t+h}\sum_{j=0}^{2}\left(\rho_{p_j}\frac{\partial A_{j_{t+h}}}{\partial \omega_{t+}}\right)\frac{\partial \omega_{t+}}{\partial \iota_t} - \widetilde{(A_{t+h}\rho_p)}\frac{\partial \omega_{t+}}{\partial \iota_t} \\
&= \frac{1}{m} + \tilde{\omega}_{t+h}\left(\sum_{j=0}^{2}\left(\rho_{p_j}\frac{\partial A_{j_{t+h}}}{\partial \omega_{t+}}\right) - \widetilde{(A_{t+h}\rho_p)}\right)\frac{\partial \omega_{t+}}{\partial \iota_t} \\
&= \frac{1}{m} + \tilde{\omega}_{t+h}\left(\sum_{j=0}^{2}\left(\rho_{p_j}\frac{2}{h}\frac{\partial A_{j_{t+h}}}{\partial \dot{\omega}_t}\right) - \widetilde{(A_{t+h}\rho_p)}\right)\frac{\partial \omega_{t+}}{\partial \iota_t} \\
&= \frac{1}{m} + \tilde{\omega}_{t+h}\left(\frac{2}{h}\sum_{j=0}^{2}\left(\rho_{p_j}\frac{\partial A_{j_{t+h}}}{\partial \dot{\omega}_t}\right) - \widetilde{(A_{t+h}\rho_p)}\right)A_t J^{-1}\tilde{\rho_q} A_t^T
\end{aligned}$$

As in Section A.5.3 we use $\frac{\partial A_{j_{t+h}}}{\partial \dot{\omega}_t}$ from the cache, and as in Section A.3.2, we approximate $\tilde{\omega}_{t+h}$ with $\tilde{\omega}_t$, and the A_{t+h} in $\widetilde{(A_{t+h}\rho_p)}$ by A_t. This then yields:

$$\frac{\partial \dot{p}_{t+h}}{\partial \iota_t} = \frac{1}{m} + \tilde{\omega}_t\left(\frac{2}{h}\sum_{j=0}^{2}\left(\rho_{p_j}\frac{\partial A_{j_{t+h}}}{\partial \dot{\omega}_t}\right) - \widetilde{(A_t\rho_p)}\right)A_t J^{-1}\tilde{\rho_q} A_t^T$$

A.5.6 Determining $\frac{\partial \dot{d}_{t+h}}{\partial \iota_t}$ analytically

Using $\dot{d} = \dot{p} - v$ and the results from the previous section we can see that:

$$\frac{\partial \dot{d}_{t+h}}{\partial \iota_t} = \tilde{\omega}_t \left(\frac{2}{h} \sum_{j=0}^{2} \left(\rho_{p_j} \frac{\partial A_{j_{t+h}}}{\partial \dot{\omega}_t} \right) - (\widetilde{A_t \rho_p}) \right) A_t J^{-1} \tilde{\rho}_q A_t^T$$

Appendix B Discrete time motion integration

As was seen in Section 3.2.4, Dynamo users can choose their own motion integrators. Four integrators are available:

1. the Euler integrator: this is a linear integrator (and therefore not very accurate), but it only requires one evaluation of the motion equations 3.3–3.6, i.e. only one call to `ode`. It uses
$$y_{t+h} = y_t + ode(y_t)h \tag{B.1}$$
for the integration. Pseudo-code for this integrator can be found on page 24.

2. the double Euler integrator: this integrator first takes an Euler-step for the first half of the integration interval, and –taking the results of that first integration step as a starting point– another Euler step for the second half of the integration interval, thereby simulating Euler-integration with half the step size. It is more accurate than the regular Euler integrator therefore (though still of linear order), but it requires two evaluations of the motion equations: using $k_1 = ode(y_t)$ and $k_2 = ode(y_t + k_1\frac{h}{2})$ it defines y_{t+h} as:
$$y_{t+h} = y_t + k_1\frac{h}{2} + k_2\frac{h}{2} \tag{B.2}$$

3. the second order Runge Kutta integrator: this integrator (discussed more formally and in more detail in [Henr 64]) uses two evaluations of the motion equations like the double Euler integrator, but its accuracy is of quadratic order. It is therefore preferred to the double Euler integrator. Using $k_1 = ode(y_t)$ and $k_2 = ode(y_t + k_1h)$ it defines y_{t+h} as:
$$y_{t+h} = y_t + \frac{k_1 + k_2}{2}h \tag{B.3}$$

4. the fourth order Runge Kutta integrator: this integrator (also described in [Henr 64]) is very accurate, but requires four evaluations of the motion equations. Pseudo-code for this integrator can be found at page 24. Using $k_1 = ode(y_t)$, $k_2 = ode(y_t + k_1\frac{h}{2})$, $k_3 = ode(y_t + k_2\frac{h}{2})$ and $k_4 = ode(y_t + k_3h)$ it defines y_{t+h} as:
$$y_{t+h} = y_t + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}h \tag{B.4}$$

The linearity of the Euler integrator (and the double Euler integrator to a lesser extent) can cause severe problems when it is used to evaluate the effects of test forces and test torques. The `ode` method of the dyna class would naively implement equations 3.5 and 3.6 using assignments $\dot\omega := AJ^{-1}A^T(M - \tilde\omega AJA^T\omega)$; $\dot q := \omega \star q$. Since the Euler integrator only calls `ode` once, any applied torque is only reflected in $\dot\omega$ (and therefore ω), and not in $\dot q$ (so also not in q): orientation changes due to the torque are hidden by the Euler integrator. If we would not have solved the motion equations for the center of mass analytically, the same would be true for the effect of any applied force on the position of the center of mass. In that case any effect of test forces and torques on constraints that are expressed in terms of position (and orientation) only (such as point-to-point constraints) would be zero, yielding a $\frac{\partial C}{\partial R}$ matrix that is not even invertible.

A solution to this problem is to already incorporate a fraction of the new angular velocity change in the change in orientation using $\dot{q} := (\omega + \rho\dot{\omega}h) \star q$ after $\dot{\omega}$ has been assigned its value (instead of $\dot{q} := \omega \star q$). Likewise, if the positional equations would be numerically integrated: $\dot{z} := v + \rho\dot{v}h$ would be used after the assignment of $\frac{F}{m}$ to $\dot{v}$ instead of $\dot{z} := v$. Here ρ is the factor which determines the portion of the velocity change which is incorporated. Its value is determined below for each of the motion integrators.

For the incorporation in the Dynamo classes, this means that the `ode` method is implemented using the modified assignment to $\dot{q}$, and that it gets an extra (real-valued) parameter in which the factor ρh is communicated. The motion integrators will have to provide this factor (each on the basis of their own ρ and step size h).

Since we can compare the numerical and analytical solutions for the positional motion equations, we will use these to determine the value of ρ for each of the integrators.

The Euler integrator uses $z_{t+h} = z_t + \dot{z}_t h$. Unfolding the modified assignment to $\dot{z}$ yields: $z_{t+h} = z_t + v_t h + \rho\frac{F}{m}h^2$ showing that the effect of the reaction force is the term $\rho\frac{F}{m}h^2$. The analytic solution $z_{t+h} = z_t + v_t h + \frac{1}{2}\frac{F}{m}h^2$ (equation 3.10 on page 21) shows a position-change of $\frac{1}{2}\frac{F}{m}h^2$ after the integration interval h. Comparing the two yields for the Euler integrator: $\rho = \frac{1}{2}$.

The double Euler integrator first transforms the motion state at time t to the one at time $t + \frac{h}{2}$, which means for the center of mass:

$$v_{t+\frac{h}{2}} = v_t + \frac{F}{m}\frac{h}{2} \quad \text{and} \quad z_{t+\frac{h}{2}} = z_t + (v_t + \rho\frac{F}{m}\frac{h}{2})\frac{h}{2}$$

Then the motion state is integrated for the second half of the integration interval to

$$\begin{aligned} v_{t+h} &= v_{t+\frac{h}{2}} + \frac{F}{m}\frac{h}{2} &&= v_t + \frac{F}{m}h \quad \text{and} \\ z_{t+h} &= z_{t+\frac{h}{2}} + (v_{t+\frac{h}{2}} + \rho\frac{F}{m}\frac{h}{2})\frac{h}{2} &&= z_t + v_t h + (\tfrac{1}{4} + \tfrac{1}{2}\rho)\frac{F}{m}h^2 \end{aligned}$$

The latter expression shows that for double Euler also $\rho = \frac{1}{2}$. Since this integrator takes stepsizes of $\frac{h}{2}$, it will provide the `ode` method with factor $\frac{h}{4}$.

The second order Runge Kutta integrator first calculates the derivatives at t:

$$\dot{v}_t = \frac{F}{m} \quad \text{and} \quad \dot{z}_t = v_t + \rho\dot{v}_t h$$

and uses those to arrive at a first estimate for the motion state at $t + h$:

$$v_{t+h} = v_t + \dot{v}_t h \quad \text{and} \quad z_{t+h} = z_t + \dot{z}_t h$$

It then calculates the derivatives based on the latter motion state:

$$\dot{v}_{t+h} = \frac{F}{m} \quad \text{and} \quad \dot{z}_{t+h} = v_{t+h} + \rho\dot{v}_{t+h} h$$

It then averages the two sets of derivatives found to arrive at

$$\dot{v}_{avg} = \frac{\dot{v}_t + \dot{v}_{t+h}}{2} = \frac{F}{m} \quad \text{and} \quad \dot{z}_{avg} = \frac{\dot{z}_t + \dot{z}_{t+h}}{2} = v_t + \frac{1+2\rho}{2}\frac{F}{m}h$$

which (after multiplication with h) are added to v_t and z_t to arrive at the final motion state for $t+h$. The effect of the reaction force on the position is therefore $\frac{1+2\rho}{2}\frac{F}{m}h^2$, yielding $\rho = 0$, which shows the quadratic nature of the second order Runge Kutta integrator.

The fourth order Runge Kutta integrator is of higher order than the second order Runge Kutta integrator: its ρ is also zero.

Another way of looking at this modification of the `ode` method is to view integrators as objects that define a function a (combining several evaluations of the *ode* function) for use in $y_{t+h} = y_t + a(y_t)h$. From this equation we see that function a is used as a kind of average derivative over the interval from t to $t+h$. The second order Runge Kutta integrator arrives at this interval by averaging the derivatives at t and $t+h$, but the regular Euler integrator can only afford one call to the `ode` method since for Euler $a = ode$. The change to the `ode` method described above modifies `ode` from implementing $\dot{y}_t = ode(y_t)$ to implementing an approximation for $\dot{y}_{t+\frac{h}{2}} = ode(y_t)$. The modified `ode` method results in a better approximation for the average derivative between t and $t+h$ than the result of the unmodified `ode` method. The reason that it is relatively easy to predict the derivative at $t+\frac{h}{2}$ from the motion state at time t is that it is known that F and M remain constant (in first order) during the interval from t to $t+h$ [1]: the modified `ode` can be seen as a method which uses a little prediction to arrive at its results. A more formal treatment of the modification can be found in Appendix A of [Court 96].

With this modification all integrators (including the Euler variants) can be used to integrate the test forces with which $\frac{\partial C}{\partial R}$ is obtained. Since the Euler integrator is a lot cheaper in computational effort, it is can even be advantageous to always use the Euler integrator when applying test forces regardless of the integrator used for applying the actual reaction forces. The resulting $\frac{\partial C}{\partial R}$ is only used in a linear fashion so any higher order terms obtained are not used as such anyway. The test described below shows that this is the case, and therefore shows that the analytical approach mimmicking discrete Euler integration described in Section A.1 is also possible.

In the test, one corner of a cube (a cube with initial center of mass $(150, 150, -150)$ and sides of length 300) was fixed in the origin. Gravity was then used to cause a swinging motion of the cube. After 500 frames (just over eight full swings of the cube) several properties of the cube and the inverse dynamics algorithm were measured: the average magnitude of the constraint error after the constraint satisfaction algorithm is done (the constraint satisfaction algorithm was set to stop iterating if the constraint error became less than 0.001), the average number of iterations used by the constraint satisfaction algorithm, and the position and orientation of the cube:

---

[1] The modified `ode` still calculates M once from A_t. It would be more accurate to base M on an estimate for $A_{t+\frac{h}{2}}$ (based on ω_t using the assumption that $\dot{\omega}_t$ is small compared to ω_t, justifying omitting its term in integrating q_t to $q_{t+\frac{h}{2}}$), but calculating this estimate would require so much extra computational effort that using the Euler integrator would almost cost as much as using the second order Runge Kutta integrator, and we still want the Euler integrator to be a cheaper alternative (be it at less accuracy).

	$\overline{\text{error}}$	$\overline{\#\text{iterations}}$	position	orientation
test 1	$4.1 \cdot 10^{-5}$	1	$\begin{pmatrix} 92.92 \\ 224.124 \\ -92.921 \end{pmatrix}$	$\begin{pmatrix} 0.943441 & -0.326681 & 0.0565596 \\ 0.326681 & 0.886882 & -0.32668 \\ 0.0565587 & 0.326681 & 0.943441 \end{pmatrix}$
test 2	$1.9 \cdot 10^{-4}$	1.6	$\begin{pmatrix} 92.927 \\ 224.119 \\ -92.927 \end{pmatrix}$	$\begin{pmatrix} 0.955525 & -0.291537 & 0.0444756 \\ 0.291537 & 0.91105 & -0.291538 \\ 0.0444747 & 0.291538 & 0.955525 \end{pmatrix}$
test 3	$4.2 \cdot 10^{-5}$	1	$\begin{pmatrix} 108.619 \\ 209.715 \\ -108.619 \end{pmatrix}$	$\begin{pmatrix} 0.974109 & -0.224593 & 0.0258892 \\ 0.224592 & 0.948217 & -0.224593 \\ 0.0258934 & 0.224592 & 0.974109 \end{pmatrix}$

In the first test, the fourth order Runge Kutta integrator was used for integration of test forces as well as for integration of the actual reaction forces. In the second test, the integrator for test forces was replaced with the Euler integrator, and in the third test also the integrator for reaction forces was changed to the Euler integrator.

When comparing the results of the first two tests it can be seen that the choice of the integrator used for applying test forces matters little to the trajectory of the cube: the position of the cube matches almost perfectly, and the inner products of the corresponding base vectors comprising the orientation are 0.9992354, 0.998471 and 0.9992355. These inner products are the cosines of the angles between the vectors since their lengths are one. The manner in which the test forces are calculated does however influence the convergence of the iteration given the 1.6 iterations on average when the "foreign" Euler integrator is used in the second test, versus the one iteration per frame in in the first and the last test where the same integrator is used for test forces and reaction forces. The configuration used in the second test is still considerable cheaper: for n constraints of average dimension d with a test integrator which evaluates `ode` k times and a reaction force integrator which evaluates `ode` l times and and average number of iterations i, the `ode` method is called $dnk + nli$ times on average (by the inverse dynamics algorithm). This is for the case where the testing is based on the current motion state and the motion state after applying a test force f_t. For the case where each column of $\frac{\partial C}{\partial R}$ is based on the difference in motion state after applying testforces $\frac{1}{2}r_t$ and $-\frac{1}{2}r_t$ (which was used to obtain the results in the table above; see page 43), this is even $2dnk + nli$. For the first test ($d = 3$, $n = 1$, $k = 4$, $l = 4$, $i = 1$) this works out at 28 evaluations of `ode` per frame while it takes only 12.4 evaluations of `ode` per frame for the second test ($k = 1$ instead of $k = 4$ and $i = 1.6$ instead of $i = 1$).

Using the Euler integrator for the integration of the reaction forces in the third test yields a trajectory that is a little different at the low cost of only 7 `ode` evaluations per frame. The difference in trajectory is not even that large given the fact that the center of mass of the cube lies at distance $150\sqrt{3}$ from the point that is fixed in space. The center of mass swings in the $x = -z$ plane and initially has an angle of approximately 0.96 with the gravity vector. A full swing therefore causes an angular motion over angle 3.82, corresponding to an arc-length of $3.82 \cdot 150\sqrt{3}$, so the eight swings comprise a trajectory with a length of almost 8.000, showing that the approximate error of 26.5 is very small (note that errors will be higher near a quarter of the period of the swing, where the velocities are higher).

The same tests performed with a $\frac{\partial C}{\partial R}$ based on the difference between the current motion state and the motion state after applying a test restriction change r_t, instead of on the difference in motion state after applying test changes $-\frac{1}{2}r_t$ and $\frac{1}{2}r_t$ yields almost identical results, showing that the extra effort required for the latter (see page 43) is not warranted.

Appendix C Velocity terms

In Section 3.4.7, it was shown that constraints that are based on purely positional terms can exhibit problems with oscillating reaction forces and velocities. This stems from the discrete time stamps at which the constraints are enforced, in combination with the absence of any constraints on velocity. Simply adding velocity constraints to rectify this absence is not possible since these extra constraints would require additional reaction forces to give the constraint solver the required degrees of freedom in the steering parameters. Instead, we add velocity terms to the positional constraint, keeping a constraint of the same dimension.

In a constraint C_{pos} with only positional terms, we replace any such term p with $p + ah\dot{p}$ (for a later to be determined factor a). In first order, the resulting modified constraint C_{vel} is again the original constraint C_{pos}, but at a different moment in time: since $p(t) + ah\dot{p}(t) \approx p(t + ah)$ and C_{pos} is based solely on positional terms, evaluating the modified constraint C_{vel} at time stamp $t + h$ (as is done by the constraint correction algorithms) roughly corresponds to evaluating C_{pos} at time stamp $t + (1 + a)h$. So instead of requiring $C_{pos}(t + h) = 0$ for subsequent values of t, we require (in first order) that $C_{pos}(t + (1 + a)h) = 0$ for subsequent values of t, in effect still trying to make sure that $C_{pos}(\tau) = 0$ for all time points τ. Although the slight shift in time removes the guarantee that the constraint is exactly met at the time points at which the scene is rendered, we now have the velocity explicitly present in the constraint equation, eliminating the freedom for the oscillation that would be hidden for $a = 0$.

From controller theory, such velocity terms are known to cause a dampening effect. Indeed: we want to dampen any oscillations that are present. We optimize the value of a to maximize the dampening effect.

For this, we concentrate on the part of the constraint that causes the oscillation: the positional term. So we investigate the behavior of constraint $C_{pos}(t) = p(t)$ (leaving out other terms that do not influence the oscillation). The modified version of this constraint is $C_{vel}(t) = p(t) + ah\dot{p}(t)$

Suppose that $C_{vel}(t) = 0$ (through constraint correction with the modified constraint at time stamp $t - h$). Enforcing $C_{vel}(t + h) = 0$ will cause the constraint solver to compute a force that results in an effective acceleration $\ddot{p}$ which ensures $C_{vel}(t + h) = 0$. Since $\ddot{p}$ is the result of constant forces between times t and $t + h$ (see design decision 7), higher order terms in $\ddot{p}$ are very small. Disregarding these higher order terms (as we successfully did in the previous appendix by applying the ρ factor also to the rotational part of the motion equations), we have:

$$p(t + h) = p(t) + h\dot{p}(t) + \frac{h^2}{2}\ddot{p}(t) \qquad \text{and} \qquad \dot{p}(t + h) = \dot{p}(t) + h\ddot{p}(t)$$

Inserting these in $C_{vel}(t + h) = 0$, while using $C_{vel}(t) = 0$, yields:

$$\ddot{p}(t) = -\frac{2}{2a + 1}\frac{\dot{p}(t)}{h}$$

Again using $C_{vel}(t) = 0$, this means:

$$p(t+h) = \frac{2a-1}{2a+1} p(t) \qquad \text{and} \qquad \dot{p}(t+h) = \frac{2a-1}{2a+1} \dot{p}(t)$$

From these equations we can see that choosing $a = \frac{1}{2}$ yields zero error in both the positional and velocity parts of the constraint after two frames of applying the modified constraint (apart from the influences of higher order terms). So despite the shift in time, choosing $a = \frac{1}{2}$ will still ensure that the constraint is not only met at the shifted time points, but also at $t + h$ (again: to first order).

Note that for $a = 0$ the alternating behavior described in Section 3.4.7 results.

In case only the unmodified constraint C_{pos} is valid at time t, changing to the modified version will result in $p(t+h) = -\frac{1}{2a+1}\dot{p}(t)$. In such a case there is a short one-frame positional error half the magnitude of the original velocity error: this is then nullified the next frame by applying the modified constraint as shown by the previous calculations, or even by applying the original constraint again.

Appendix D The tensor of inertia and one dimensional dynas

To be able to integrate motion equation 3.6 (from page 15), a dyna's tensor of inertia is required. This tensor matrix specifies the mass distribution within the dyna. Since Dynamo does not know the topology of a dyna (it only administrates the center of mass and orientation), the determination of the tensor of inertia is up to the host system.

In the test host system, the GDP, geometries consist of point clouds, so it is a natural extension to consider the mass of the dyna concentrated in these vertices. Each of the vertices of such a cloud is therefore equiped with an attribute signifying its mass. This might give counter-intuitive mass distributions since in reality the mass distribution depends on the volume encompassed by the primitives based on the vertices, but since the mass of each point can be arbitrarily chosen, each mass distribution can be obtained (if necessary 'invisible' vertices can be added to add some weight to a given part of the dyna). By default each vertex is assigned a mass of one (when an input file does not contain any mass-information), so a Looks programmer has to be careful when for example certain parts of a dyna contain a lot of detail (and therefore a lot of vertices): such parts get very 'heavy'.

Given mass m_i for each of the points p_i in the dyna's vertex cloud (where p_i is given in world coordinates), the total mass m of the dyna (required in motion equation 3.4) is the sum of all the m_i and the center of mass z of the dyna can be calculated as $\frac{1}{m}\sum_i m_i p_i$. The tensor of inertia is then given by $J_w = \sum_i m_i(|r_i|I - r_i\!:\!r_i)$, where $r_i = p_i - z$. This tensor can then be diagonalized (using Jacobi decomposition, which decomposes J_w in orientation A and diagonal matrix J in such a way that $J_w = AJA^T$) to obtain an initial orientation for the dyna and a constant diagonal tensor of inertia (in local coordinates). This functionality is implemented in the GDP wrapper for Dynamo.

A problem arises with this tensor of inertia when the dyna in question is one-dimensional, i.e. all the vertices of the dyna lie on one line. In that case there is a zero on the diagonal of J, which means that J cannot be inverted as is required in motion equation 3.6. This corresponds with the notion that rotation and torque around the line of which the dyna consists are meaningless. Observe however that in the case of a line-shaped dyna, any force exerted on the dyna has to be applied on that line: such a force can never exert a torque around the line and therefore never cause a rotation around the line. So the only special care required for one-dimensional dynas is to ensure that any torque applied to the dyna (via the `applyforce` or `applytorque` methods) does not have any components in the direction of the line (if they do, such components can safely be removed).

Small 'rotations' around the dyna's line can arise when a higher order integrator is used: such an integrator might evaluate the right-hand side of equation 3.6 when the dyna has already rotated slightly. This makes it possible for the (unrotated) torque M to have a small component in the direction of the (rotated) line, so at each evaluation of the motion equations, either this component or the similar component in the resulting ω has to be removed.

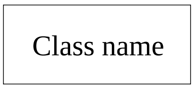
Appendix E List of symbols

These are the most important symbols used throughout this thesis:

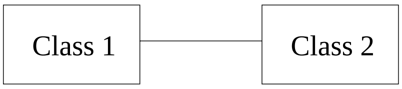
m	mass of a dyna
J_w	the tensor of inertia of a dyna in world coordinates
J	the diagonal tensor of inertia of a dyna in local coordinates
z_τ	position of the centroid of a geometry at time τ
v_τ	velocity of the centroid of a geometry at time τ
A_τ	orientation of a geometry at time τ in matrix representation
$A_{i\tau}$	The $i-th$ column of A_τ
q_τ	orientation of a geometry at time τ in Euler parameter representation
ω_τ	angular velocity of a geometry at time τ
p_τ	the position of a point in world coordinates at time τ
ρ	the position of a point in local coordinates
d_τ	a direction vector within a dyna in world coordinates at time τ
ψ	a direction vector within a dyna in local coordinates
h	step size of the simulation
F_τ	a force F applied to the centroid of a dyna at time τ
M_τ	a torque M applied to a dyna at time τ
f_τ^p	a force f applied to point p of a dyna at time τ
ι_τ^p	an impulse change applied to point p at time τ
C_τ	constraint error vector at time τ
$\wp_\tau^{ik}$	a motion-state based property $\wp$ of the k-th geometry of constraint i at time τ
Γ_τ^{jl}	a force, torque or impulse change applied to the l-th dyna of constraint j at time τ
R_τ	restriction value vector at time τ

Appendix F OMT notation

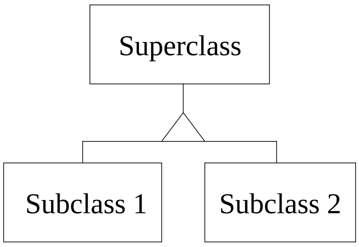
Class:



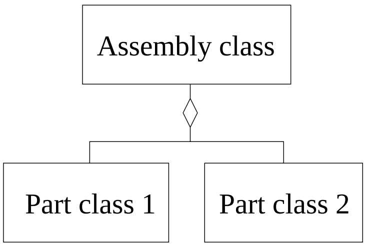
Association:



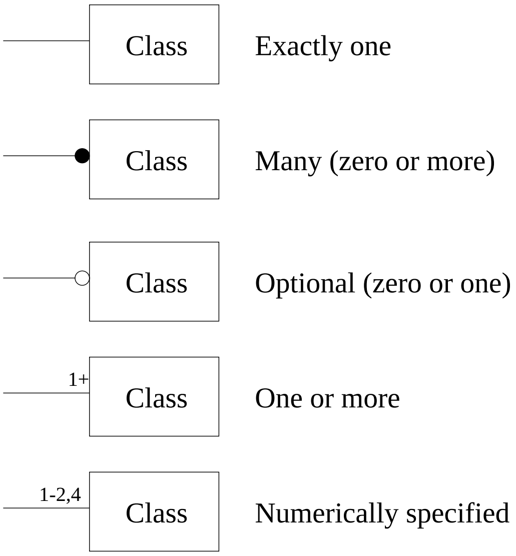
Inheritance (generalization):



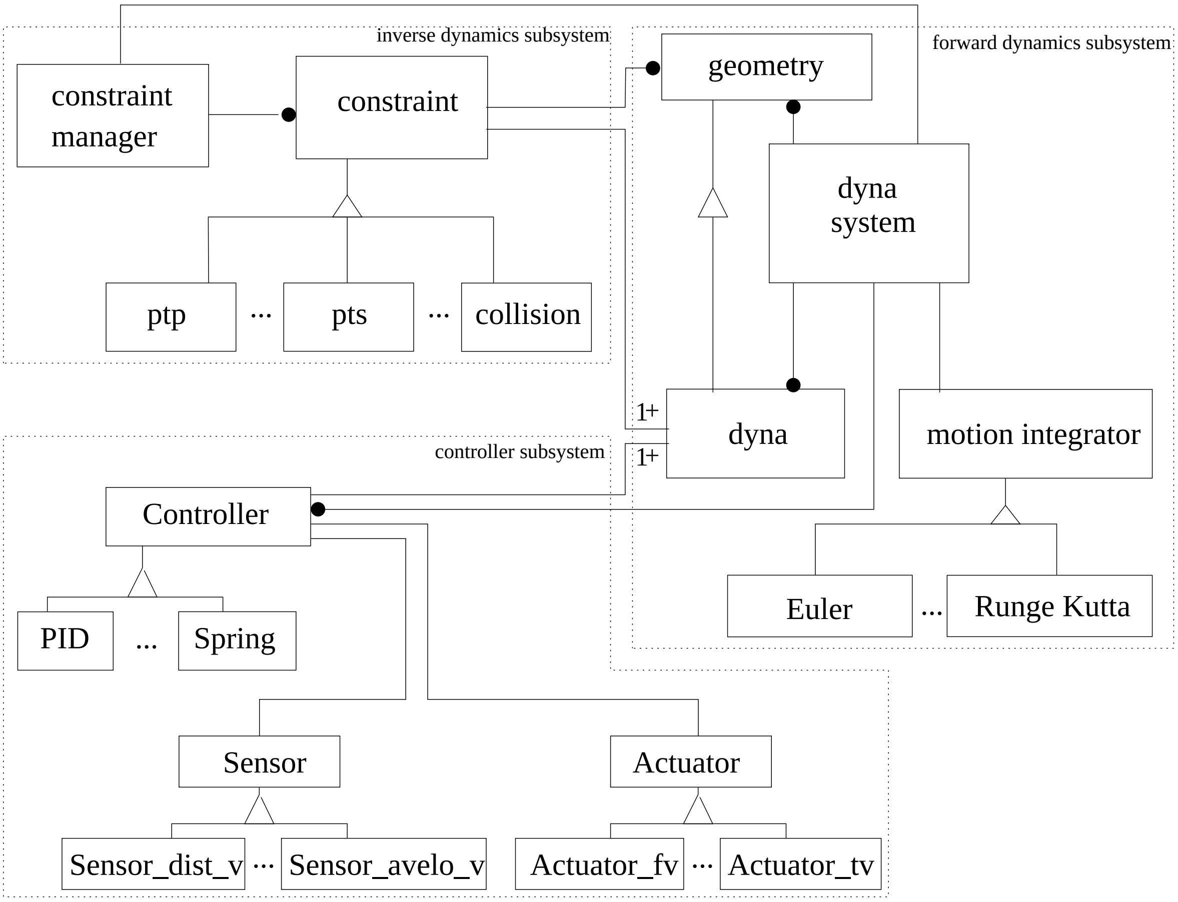
Composition (aggregation):



Mulitplicity of associations:



Appendix G Dynamo classes overview



Appendix H Constraint satisfaction

This section gives an abstract overview of the constraint satisfaction problem and the solution used in the terminology often found within the field of constraint correction.

H.1 The constraint satisfaction problem

We have a set C of constraints C_i. Each of the constraints will have to remain satisfied during the time of its existence. Time is discreet, so this means that we have a sequence of moments in time at which the constraints will have to be satisfied. Each constraint has a vector of variables $\bar{x}_i \in \Re^n$ associated with it. In general a constraint can also depend on the variables of other constraints, so a constraint is a function of $\bar{x}$ (the concatenation of all $\bar{x}_i$). There may also be an external influence $\bar{e}$ on the constraint. Evaluation of a constraint function yields a vector in $\Re^m$: $C_i(\bar{x}, \bar{e}) \in \Re^m$. We will assume $m = n$, and call C_i an n-dimensional constraint. A constraint is satisfied if it evaluates to $\bar{0}$. Any non-zero constraint value can be seen as a measure of the deviation of a constraint: the function C_i is also called the *constraint error* function.

The problem is assigning values to each of the x_{ij} for subsequent time steps, such that all constraints in C evaluate to $\bar{0}$ at each time step.

H.2 Domain specific issues

The focus here is on a special case of the problem, where the environment in which this problem has to be solved is an animation system where the $\bar{x}_i$ represent forces that can be used to move geometries, and where the constraints represent geometrical constraints (for example constraints on the position of a point of such a geometry). The constraints have to be valid for every frame of the animation, and these frames are generated sequentially, one at a time. This means that there is an initial configuration of the geometries where all constraints are met, and then –for each subsequent frame– the values of the x_{ij} will have to be adjusted to compensate for external changes (such as gravity, inertia and externally applied forces) to the geometries. So in this case we are more concerned with *constraint correction*, using values for the x_{ij} from previous frames, than with satisfying all constraints from scratch each frame.

H.3 Iterative constraint correction

The constraint correction process is then guided by the constraint errors for the constraints: as long as there is still a constraint error, the values of the x_{ij} will have to be corrected to improve the solution. Since adjustments are only small (as described in the previous section), we can linearize the constraint function and use a Newton-Rhapson iteration to converge to a solution:

Linearizing the constraint function yields: $C_i(\bar{x} + \bar{\Delta x}, \bar{e}) = C_i(\bar{x}, \bar{e}) + \frac{\partial C_i(\bar{x}, \bar{e})}{\partial \bar{x}}\,\bar{\Delta x}$.

Knowing that our goal is to make sure that $C_i(\bar{x} + \bar{\Delta x}, \bar{e}) = 0$ we can solve for $\bar{\Delta x}$ from this equation: $\bar{\Delta x} = -\left(\frac{\partial C_i(\bar{x}, \bar{e})}{\partial \bar{x}}\right)^{-1} C_i(\bar{x}, \bar{e})$.

So once we know the square matrix $\frac{\partial C_i(\bar{x}, \bar{e})}{\partial \bar{x}}$, we can use the error vector $C_i(\bar{x}, \bar{e})$ (iteratively, to account for non-linear effects) to solve for the changes in the variable values. Any time the external influence $\bar{e}$ changes (between time steps), constraint correction will have to be performed again.

H.4 Constraint correction considerations

Since the $\frac{\partial C_i(\bar{x}, \bar{e})}{\partial \bar{x}}$ matrix depends on the current value of $\bar{x}$, it should be recalculated after every change to $\bar{x}$. However, the matrix is only used as a guide to determine the direction in which to change the value of $\bar{x}$, so precise accuracy is not required for the matrix: the constraint correction iteration then reverts from being a Newton process to being a pseudo-Newton process. This also means that some shortcuts can be taken in recalculating the matrix: there is a balance between the time spent on calculating the matrix, and the time spent on the actual iteration (which will converge slower if the matrix is not very accurate). For the process environment described in Section H.2, the choice has been made to recalculate the matrix only once for each time step (design decision 16), or possibly even once every few time steps (this is user controllable), instead of after each iteration step. An option for future research is to try to use Broyden iteration to use the coherence between the $\frac{\partial C_i(\bar{x}, \bar{e})}{\partial \bar{x}}$ matrices on subsequent time steps (this coherence is now not used: the matrix is recalculated from scratch at the start of each new time step).

Since each constraint C_i has its own variables $\bar{x}_i$ associated with it, it usually does not depend very heavily on the variables of other constraints: the $\frac{\partial C_i(\bar{x}, \bar{e})}{\partial \bar{x}}$ is sparse, so sparse matrix techniques can be used in solving for $\bar{\Delta x}$.

Appendix I Source of an example

Below is the source code for the example described in Section 5.1, which illustrates the use of the C++ Dynamo classes to perform the dynamics calculations.

```
// A demo for the use of the Dynamo classes.
// It shows a chain of cubes that swings under influence of gravity.

#include "rungekutta2.h"
#include "ptp.h"

// How many cubes make up the chain?:
#define NCUBES 15

// Here is the class for our cubes, these function mainly as an intermediate
// between the dyna companion, the control routine, and the rendering software.
class MyCube {
  public:
    // the attributes of the cube:
    DL_point pos;       // the position of the center of the cube
    DL_matrix orient;   // the orientation of the cube
    DL_dyna* companion; // the companion which does all the dynamics
                        // calculations for us
    DL_ptp* link;       // used to connect this cube to another

    // the functions needed for the DL_dyna_callback to copy position and
    // orientation to/from the companion:
    void get_new_geo_info(DL_geo *g){
      g->move(&pos,&orient);
    }
    void update_dyna_companion(DL_dyna *d){
      pos.assign(d->get_position());
      orient.assign(d->get_orientation());
    }

    // show an example of a constraint: this method uses a point-to-point
    // constraint to connect one of the cube’s corners to a corner of the
    // other cube.
    void ConnectTo(MyCube *cube){
      DL_point posm(1,1,1),posp(-1,-1,-1);
      link=new DL_ptp();
      if (cube) link->init(companion,&posm,cube->companion,&posp);
      else { // connect to the current position of the corner in the world.
        DL_point posw;
        companion->to_world(&posm,&posw);
        link->init(companion,&posm,NULL,&posw);
      }
    }
    void Disconnect(){
      if (link) delete link;
      link=NULL;
    }

    MyCube(DL_point& newpos){
    // constructor: its main task is creating and initialising the companion
      pos.assign(&newpos);
      orient.makeone();
      companion=new DL_dyna((void*)this);
      companion->set_mass(1);
      companion->set_inertiatensor(1,1,1);
      companion->set_velodamping(0.995); // introduce a bit of friction
      link=NULL;
    }
    ~MyCube(){
      if (link) delete link;
      delete companion;
    }
};

// we need to implement the callbacks that allow the dyna system to commicate
// its calculated positions and orientations to us:
class My_dyna_system_callbacks : public DL_dyna_system_callbacks {
  public:
    virtual void get_new_geo_info(DL_geo *g){
      ((MyCube*)(g->get_companion()))->get_new_geo_info(g);
    }
    virtual void update_dyna_companion(DL_dyna *d){
      ((MyCube*)(d->get_companion()))->update_dyna_companion(d);
    }
};

// rendering is taken care of elsewhere:
#include "render.cpp"

// here comes the main control function: it initialises the dyna system
// and then creates a chain of cubes
void main(){
  int i;
  MyCube* cube[NCUBES];
  My_dyna_system_callbacks *dsc=new My_dyna_system_callbacks();
  DL_m_integrator* my_int=new DL_rungekutta2();
  DL_dyna_system dsystem(dsc,my_int);
  DL_constraint_manager *constraints=new DL_constraint_manager();
  constraints->max_error=0.0001;

  DL_point pos(0.75*NCUBES,1.75*NCUBES,4*NCUBES);
  DL_vector vec(-2,-2,-2);

  for (i=0;i<NCUBES;i++){
    cube[i]=new MyCube(pos);
    pos.plusis(&vec);
  }

  cube[0]->ConnectTo(NULL);
  for (i=1;i<NCUBES;i++) cube[i]->ConnectTo(cube[i-1]);

  vec.init(0,-1,0);
  dsystem.set_gravity(&vec);
  my_int->set_stepsize(0.02);

  InitRender(dsystem,cube,NCUBES);

  // everything initialised. Now let the animation run:
  while (RenderCubes(cube)) {
    dsystem.dynamics();
    if (fabs(dsystem.time()-70)<0.01) cube[NCUBES/2]->Disconnect();
  }

  // all done: clean up:
  for (i=0;i<NCUBES;i++) delete cube[i];
  delete constraints; delete my_int; delete dsc;
}
```

Bibliography

[Aksit 92] Mehmet Aksit and Lodewijk Bergmans *Obstacles in Object-Oriented Software Developement*, Proceedings OOPSLA '92, ACM SIGPPLAN Notices, Vol. 27, No. 10, October 1992, pages 341-358

[Alonz 70] Marcelo Alonzo and Edward J. Finn *Physics*, Addison-Wesley London, 1970

[Ames 99] P.R. Amestoy, I.S. Duff, J,-Y L'Excellent and J. Koster *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, Technical Report RAL-TR-1999-059, Rutherford Appleton Laboratory, Chilton, Didcot, England

[AVS] The AVS homepage at `http://www.avs.com/`

[Baraf 92] David Baraff *Dynamic simulation of non-penetrating rigid bodies*, Cornell University, 1992

[Baraf 96] David Baraff *Linear-Time Dynamics using Lagrange Multipliers*, in Siggraph 96 Computer Graphics Proceedings, ACM 1996

[Baren 94] B. Barenbrug *Using a constraint driven approach to dynamic simulation of rigid body movements in computer animation*, Master's Thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1994

[Baren 96] B. Barenbrug *Designing a library for constraint driven dynamics in the GDP* Final report of the postgraduate programme Software Technology, Eindhoven University of Technology, 1996

[Baren 99] *The Dynamo Library homepage* at `http://www.win.tue.nl/~bartb/dynamo/`

[Baum 72] J. Baumgarte *Stabilisation of constraints and integrals of motion in dynamical systems*, in Computer Methods in Applied Mechanics, volume 1, pages 1-36, 1972

[Barz 88] Ronen Barzel and Alan H. Barr *A Modeling System Based On Dynamic Constraints*, In ACM Computer Graphics, Volume 22, pages 179-188, August 1988

[Berg 99] G. van den Bergen *Collision Detection in Interactive 3D Computer Animation*, Ph.D Thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, March 1999

[Boro 66] Sidney Borowitz and Arthur Beiser *Essentials of Physics*, Addison-Wesley, 1966

[Buck 98] Matthias Buck and Elmar Schömer *Interactive Rigid Body Manipulation with Obstacle Contacts* 6th International Conference in Central Europe on Computer Graphics and Visualization. (available at `http://www-hotz.cs.uni-sb.de/~schoemer/`)

[Caban 68] H. Cabannes *General Mechanics*, Blaisdell Waltham, 1968

[Chenn 98] S. Chenney, J Ichnnowski. and D. Forsyth *Efficient dynamics modeling for VRML and Java*, Proceedings of the VRML 1998 Symposium, available online at `http://www.ece.uwaterloo.ca/vrml98/`

[Court 96] M.J.M. Courtin *Numerieke simulatie van de dynamica van starre lichamen*, internal report, Eindhoven University of Technology, 1996

[Faure 96] F. Faure *An Energy-Based Approach for Contact Force Computation*, Proceedings of EuroGraphics 1996, vol 15, No 3, 1996

[Faure 99] F. Faure *Fast Iterative Refinement of Articulated Solid Dynamics* To appear in TVCG, 1999, available as `http://www.cg.tuwien.ac.at/~francois/papers/refinable.solution.html`

[Fraue 66] P. Frauenfelder and P. Huber *Introduction to physics Volume 1* Pergamon, London, 1966.

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns – Elements of reusable object-oriented software*, Addison-Wesley, 1995

[Gasc 94] J. Gascuel and M. Gascuel *Displacement constraints for interactive modeling and animation of articulated structures*, in The Visual Computer, issue 10 of 1994, pages 191-204, Springer Verlag 1994.

[Haug 89] E. J. Haug *Computer aided kinematics and dynamics of mechanical systems. Volume 1: Basic Methods*, Allyn and Bacon, London, 1989

[Henr 64] Peter Henrici *Elements of numerical analysis*, John Wiley & Sons, Inc. New York • London • Sydney, 1964

[Huiz 97] C. Huizing and B. Barenbrug *Interactive Animation needs Components*, Foundations of Component-Based Systems Workshop, Zurich, September 26, 1997, pp. 151-155

[Kell 93] R.H.M.C. Kelleners *Solving collisions in Walt*, Master's Thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1993

[Kell 99] R.H.M.C. Kelleners *Constraints in Object-Oriented Graphics*, Ph.D. Thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1999

[Kell 96] H. Keller, H. Stolz, A. Ziegler, T Bräunl *Virtual Mechanics, Simulation and Animation of Rigid Body Systems*, `http://www.ee.uwa.edu.au/~braunl/aero/docu.english.ps.gz`

[Krijg 94] R.P.J. de Krijger and M. Loehr *The implementation of (inverse) kinematics and dynamics in the GDP animator*, Master's Thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1994

[Louch 96] Jean Louchet, Jiang Li *An identification tool to build physical models for Virtual Reality*, 3rd Intl. Workshop on Image and Signal Processing IWISP '96, Manchester, UK, November 1996.

[Mirti 96] Brian Mirtich *Impulse-based Dynamic Simulation of Rigid Body Systems*, Ph.D. thesis, University of California, Berkeley, December, 1996

[Mirti 98] Brian Mirtich *Rigid body contact: collision detection to force computation*, Mitsubishi electric research laboratory (`http://www.merl.com/`) document TR-98-01, March 1998

[Overv 95] C.W.A.M. van Overveld and B. Barenbrug *All you need is force: a constraint based approach for rigid body dynamics in computer animation*, In D. Terzopoulos and D. Thalmann (editors), *Computer Animation and Simulation '95*, SpringerWienNewYork, 1995

[Overv 91] C.W.A.M. van Overveld *An iterative approach to dynamic simulation of 3-D rigid-body motions for real-time interactive computer animation*, Visual Computing 7, 1991, pp. 29-38

[Overv 93a] C.W.A.M. van Overveld *A simple approximation to rigid body dynamics for computer animation*, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1993

[Overv 93b] C.W.A.M. van Overveld *Small steps for Mankind: towards a kinematic driven dynamic simulation of curved path walking*, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1993

[Panne 95] Michiel van de Panne and Alexis Lamouret *Guided optimization for balanced locomotion*, In D. Terzopoulos and D. Thalmann (editors), *Computer Animation and Simulation '95*, SpringerWienNewYork, 1995

[Pau 98] Cristian Pau *Java, VRML and Dynamics: extended authoring interface* Internal Report, Eindhoven University of Techonology, July 1998.

[Peet 95] Eric Peeters *Design of an Object-Oriented, Interactive Animation System*, Dissertation, Eindhoven University of Technology, 1995

[Platt 88] J.C. Platt and A.H. Barr *Constraint Methods for Flexible Models* ACM Computer Graphics, Volume 22, Number 4, August 1988, pages 279-288

[Press 92] William H, Press, Saul A. Teukolsky, William T. Vetterling, et al. *Numerical recipes in C : the art of scientific computing*, 2nd edition, Cambridge University Press, 1992

[Rumb 96] James Rumbaugh *OMT insights : perspectives on modeling from the Journal of Object-Oriented Programming*, SIGS Books, 1996

[Saad 95] Yousef Saad *Iterative methods for sparse linear systems*, PWS Publishing Co., 1995

[Telea 99m] A.C. Telea, J.J. van Wijk *Vission: An Object-Oriented Dataflow System for Simulation and Visualization*, presented at the IEEE/EG International Workshop on Visualization and Simulation VisSym'99, Vienna, Austria, May 1999

[Telea 99j] A.C. Telea *Combining Object-Oriented and Dataflow Programming in the Vission Simulation System*, presented at the TOOLS'99 Europe Conference, Nancy, France, June 1999

[Thom 96] Victor Thomasse *The sound of motion : design of a modeler for animations*, Master's Thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1996

[Tsang 93] E. Tsang *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[Vegte 90] John van de Vegte *Feedback control systems*, second edition, Prentice-Hall International, 1990

[Wijck 96] P.M.E.J. Wijckmans *Conditioning of differential algebraic equations and numerical solution of multibody dynamics* Ph.D. Thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1996

[Witt 77] J. Wittenburg *Dynamics of systems of rigid bodies*, B.G. Teubner Stuttgart, 1977

[Zlat 91] Zharari Zlatev *Computational methods for general sparse matrices*, Kluwer Academic, 1991

Index

$\frac{\partial C}{\partial R}$ determination
  analytical, 44
  comparison, 49
  empirical, 42

acknowledgements, i
actuators, 30
  specific, 86
application designer, 6

bandwidth optimization, 51
bar constraint, 74
Broyden's method, 44

caching optimization, 49
class overview, 129
collision constraint, 79
conclusions, 106
conjugate gradient, 51
connector, 70
constraint, 32
  correction, 33
  dimension, 39, 130
  positional issues, 54
  prototyping, 50
  satisfaction, 33, 130
  solving, 50
constraints
  bar, 74
  collision, 79
  connector, 70
  cylinder, 69
  empty, 61
  linehinge, 67
  multibar, 75
  multirope, 75
  orientation, 71
  plane, 73
  prism, 72
  ptc, 63
  ptp, 62
  pts, 66
  rope, 74
  vtv, 63
  wheel, 75
controllers, 28
  damper, 84
  PID, 84
  spring, 84
coordinates
  local, 13
  world, 13
curves, 65
CutHillMcKee ordering, 51
CV, 143
cylinder constraint, 69

damper controller, 84
degrees of freedom
  of a constraint, 34
design, 11
  algorithm design, 103
  code design, 104
  decisions, 12, 16, 17, 19–23, 25, 33, 34, 37, 40, 48, 52
  interface design, 102
  methodology, 11
  phases, 11, 101
  process, 101
diagonal dominance, 53
dynamics
  forward, 1, 13
  inverse, 1, 32
Dynamo, 2
dynas, 7, 17

emergent behavior, 89, 93
empty constraint, 61
Euler integrator, 120
  modified, 121
examples, 87
  basic, 87
  basic, source, 132
  bicycle, 89
  coupled pendula, 91

  roller coaster, 94

friction, 58
future work, 107

geometries, 13, 16
gravity, 19

host system, 7
  GDP, 5, 8, 57

interactivity, 1, 91
interface of
  `actuator`, 30
  `constraint_manager`, 40
  `constraint`, 35
  `curve`, 65
  `dyna_system`, 18
  `dyna`, 17, 24, 25
  `geometry`, 17
  `sensor`, 30
  `surface`, 67
interfaces
  external, 19
  internal, 19
introduction, 1
inverse dyamics algorithm, 37

linehinge constraint, 67
LU decomposition, 51

motion integration, 19, 120
  analytical, 21
  classes, 23
  numerical, 21
  step size, 19
motion state, 7
multibar constraint, 75
multirope constraint, 75

Newton iteration, 38
  pseudo, 40

object orientation, 12, 49, 101
OMT, 128
one dimensional dynas, 126
orientation constraint, 71

partial derivatives, 45, 109
perfomance measurements, 88, 90, 93, 98
physics, 13
  forces, 15
  impulses, 15
  inertia, 14
  torques, 15
PID controller, 84
plane constraint, 73
planning, 104
previous work, 2
prism constraint, 72
ptc constraint, 63
ptp constraint, 62
pts constraint, 66

references, 134
requirements, 8
restrictions, 37
rope constraint, 74
Runge Kutta integrator, 120

sensors, 30
  specific, 86
simulation systems, 6
software architecture, 129
software design, 2, 11
software structure, 1
spring controller, 84
stiffness, 35, 41
summary, 140
surface, 67
SV decomposition, 51
symbol list, 127

tensor of inertia, 126
test forces, 42

users, 2, 6

velocity terms, 55, 124
views
  modeling, 11
  physics, 11
  software, 11
visualizing forces, 56
vtv, 63

wheel constraint, 75

Summary

In many computer animation systems, the motions of the moving objects have to be calculated at a sufficient rate to support the interactivity. Yet those motions should look natural to the user of such a system. A method of making motions look natural is to program into the animation system, the laws of physics which in reality govern the motions of objects. This method of motion calculation is called *dynamics*.

This thesis describes the design of class library *Dynamo*. Dynamo users can use classes from this library to add dynamics functionality to their own animation systems. Dynamo has been designed completely in an object oriented fashion, allowing for interfaces that abstract from implementation details, and for a very structured design. This benefits extendibility, understandability and maintainability of the software. Dynamo consists of three major subsystems, dealing with forward dynamics, controllers, and inverse dynamics, respectively.

The *forward dynamics* subsystem implements the calculation of the motions of (in this case, rigid) bodies, as a function of their inertia and forces that are applied to them. The differential equations that describe the motions are integrated over time, while accounting for the fact that in animation systems time is modeled in a discrete manner. In Dynamo, several motion integrators are available to a user. This allows the user to choose an appropriate balance between computational effort and precision.

The *controller* subsystem provides controllers: devices that are provided with a reference signal and try to calculate forces that steer the motions in an animation in a way that corresponds to the reference signal. Dynamo provides several kinds of controllers, such as controllers which model springs, and so-called PID controllers.

The *inverse dynamics* subsystem provides a user of Dynamo with the option of specifying *constraints* on the motions of the rigid bodies. Constraints can for example be used to connect several rigid bodies to form articulated rigid bodies. While controllers not always instantly match the behavior of the system to the reference signal, constraints have to be valid at all times. Constraint satisfaction is automatically performed by Dynamo, so that a user only has to declare a constraint once to have it enforced from then on. The constraint correction algorithms employed in Dynamo allow for a wide variety of constraints, including non-holonomic constraints, and allow them to be combined freely, allowing for loops in the constraint configurations. Over a dozen constraint types are available, and the software is designed to facilitate the addition of new constraint types.

Several examples show how Dynamo can be used in a variety of application areas, and show that Dynamo is fast enough to provide animation at interactive speeds for non-trivial systems. They also show that systems of rigid bodies that are connected by constraints, often exhibit emergent behavior on a system level. The high-level interfaces of Dynamo, obtained through the object oriented approach, make it very easy to specify systems with such a complex behavior, which can be studied through the simulation of the motions in the animation.

Samenvatting

In veel computer animatie systemen moeten de bewegingen van de geanimeerde voorwerpen voldoende snel berekend worden om interactiviteit te ondersteunen. Desondanks moeten de voorwerpen zich op een voor de gebruiker van zo'n systeem natuurlijke manier bewegen. Een methode om natuurlijke bewegingen te genereren is die waarbij de natuurwetten die in de fysieke werkelijkheid de bewegingen van voorwerpen bepalen in het animatie systeem geprogrammeerd worden. Deze methode wordt *dynamica* genoemd.

Dit proefschrift beschrijft het ontwerp van klasse bibliotheek *Dynamo*. Dynamo gebruikers kunnen de klassen uit deze bibliotheek gebruiken om dynamica functionaliteit aan hun animatie systemen toe te voegen. Dynamo is geheel object-georiënteerd ontworpen. Dit staat interfaces toe die abstraheren van implementatie details, en een zeer gestructureerde software architectuur, die de uitbreidbaarheid, begrijpelijkheid en onderhoudbaarheid van de software ten goede komen. Dynamo bestaat uit drie deelsystemen: het voorwaartse dynamica deelsysteem, het controller deelsysteem, en het inverse dynamica deelsysteem.

Het *voorwaartse dynamica* deelsysteem implementeert de berekeningen van de bewegingen van de (in dit geval starre) voorwerpen, als functie van hun traagheid en krachten die erop werken. De differentiaal vergelijkingen die de bewegingen beschrijven worden naar de tijd geïntegreerd, waarbij er rekening gehouden wordt dat tijd in animatiesystemen discreet gemodelleerd is. In Dynamo zijn voor een gebruiker verschillende bewegingsintegratoren beschikbaar. Een gebruiker kan daardoor een keuze maken in de balans tussen benodigde rekenkracht en behaalde precisie.

Het *controller* deelsysteem maakt controllers beschikbaar. Dit zijn entiteiten die gegeven een referentiesignaal krachten proberen uit te rekenen die de bewegingen in een animatie met dat referentiesignaal overeen laten komen. Dynamo stelt verschillende controllers beschikbaar, zoals controllers die gedempte veren modelleren, en zogenaamde PID controllers.

Het *inverse dynamica* subsysteem geeft een gebruiker van Dynamo de mogelijkheid om beperkingen op te leggen aan de bewegingen van de voorwerpen. Middels zulke *constraints* kunnen bijvoorbeeld starre voorwerpen aan elkaar gekoppeld worden om zodoende zogenaamde gelede starre lichamen te vormen. Terwijl controllers niet altijd instantaan het gedrag tot stand brengen dat door het referentiesignaal beschreven wordt, moet aan de constraints altijd voldaan zijn. Als een constraint eenmaal door een gebruiker gedeclareerd is zorgt Dynamo er daarna automatisch voor dat dit blijvend het geval is. De constraint correctie algoritmes die in Dynamo gebruikt worden staan een brede variëteit aan constraints toe, inclusief niet-holonome constraints, die vrijelijk gecombineerd kunnen worden, waarbij lussen in de constraint configuratie toegestaan zijn. Meer dan een dozijn constraint typen zijn beschikbaar, en de software is zodanig ontworpen dat nieuwe typen makkelijk toegevoegd kunnen worden.

Enkele voorbeelden laten zien hoe Dynamo in een aantal toepassingsgebieden gebruikt kan worden, en dat Dynamo snel genoeg is voor interactieve niet-triviale toepassingen. De voorbeelden laten ook zien dat bij systemen van starre lichamen die middels constraints gekoppeld zijn, vaak gedrag op systeem-niveau naar voren komt. Dynamo maakt het mede door de object-georiënteerde aanpak makkelijk om systemen te specificeren met zulk complex gedrag, dat middels de simulatie van de bewegingen bestudeerd kan worden.

Curriculum Vitae

Bart Barenbrug was born on July 17th 1971 in Oud-Beijerland, The Netherlands, but he grew up in Eindhoven. After attending the Lorentz Lyceum, he started his study Computing Science at the Eindhoven University of Technology in 1989. For his M.Sc. thesis he began his work on computer animation using dynamics under supervision of Kees van Overveld.

After graduation in 1994, he started the two year postgraduate programme Software Technology (OOTI) at the Stan Ackermans Institute. For the final project of this course, he once again turned to the subject of computer animation using dynamics, and started on the design of a software library on this subject. After his graduation in 1996, he continued this work to arrive at the Ph.D. thesis and design you now have before you.

From December 1st 1998, Bart has been working for Philips Electronics N.V. at the Philips Research Laboratories as a research scientist focusing on the domain of computer graphics.