# GraspIt!

## User Manual

Version 2.0 beta

# Contents

# 1 Introduction

## 1.1 What GraspIt! is

GraspIt! was created to serve as a tool for grasping research. It is a simulator that can accommodate arbitrary hand and robot designs. It can also load objects and obstacles of arbitrary geometry to populate a complete simulation world. The GraspIt! engine includes a rapid collision detection and contact determination system that allows a user to interactively manipulate a robot or an object and create contacts between them. Once a grasp is created, one of the key features of the simulator is the set of grasp quality metrics. Each grasp is evaluated with numeric quality measures, and visualization methods allow the user to see the weak point of the grasp and create arbitrary 3D projections of the 6D grasp wrench space.

In our experience, we have found that GraspIt! usually serves one of two purposes. First, it can be used as a development tool, to execute and test various robot control algorithms. In this sense, it serves as a replacement for the real world: in simulation, an algorithm can be tested on many hand designs, many objects and obstacle configurations, at no cost and much faster than in the real world. Second, GraspIt! can be used as a computational platform that backs up a robot that does operate in the real world. For example, a real robot can acquire a model of a target object, then use GraspIt! to quickly evaluate multiple grasping or manipulation scenarios. Often, these scenarios are also combined and the same GraspIt! setup used for development of an algorithm can also be used for computations during real life execution.

GraspIt! has many features that help accomplish these roles; all of these features are documented in the second part of this manual. The most commonly used include the contact detection and grasp quality metrics mentioned above, the dynamics engine and the grasp planning capabilities. The dynamics engine within GraspIt! computes the motions of a group of connected robot elements, such as an arm and a hand, under the influence of controlled motor forces, joint constraint forces, contact forces and external forces. This allows a user to dynamically simulate an entire grasping task, as well as test custom robot control algorithms. The grasp planning algorithms rely on the simulated environment to quickly evaluate many hand postures, and find those that lead to stable grasps. There are many possible implementations of this concept; the planners that are included with GraspIt! can usually find multiple stable grasps of an object in less than 1 minute, taking into account obstacles and other constraints.

Overall, GraspIt! is an open-source virtual environment for simulating robotic grasping tasks accompanied by a number of analysis and development tools. It has been developed in C++ using many other open-source libraries, and is cross-platform, tested on both Windows and Ubuntu Linux.

## 1.2 What GraspIt! is NOT

GraspIt! is not an off-the-shelf product. It is rather a large codebase that is the result of many years of research and development in the Robotics Lab at Columbia University. The tools included can greatly help in the development and testing of new algorithms and approaches. However, we have often found that most new interesting problems can not be completely addressed by applying these tools exactly as they are. It is very possible that, in the process, you will find yourself needing various changes or additions to the simulator; these might be bigger or smaller depending on your problem. This means that you might have to get your hands dirty and tinker with the code itself, which we encourage you to do. We have done our best to create clean, robust and well-documented code. However, we are not a large team of product development software engineers; rather, we are a small group of robotics researchers. As a result, do not be surprised if you find parts of the code that can be improved.

## 1.3 About this manual

This manual is divided into two parts. The first one, containing **Chapters 1 through 4**, is an introduction to the GraspIt! environment, covering the essential aspects for loading and populating a GraspIt! world and interacting with it. The references to the source code are kept at a minimum. In the second part, containing **Chapter 5 through 14**, we discuss the GraspIt! advanced features, or the tools that it offers for solving various problems. Here we overlap discussions of the simulator as a final product with discussion of the source code itself. We also introduce a number of features of GraspIt! that are not completely finished yet, and might not be very robust, but have been included in the distribution in the hope you might find them useful. Such features will be marked by the qualifier "Under Development" - only use them if you're not afraid of delving inside the code to get the most out of them and fix an occasional bug.

Please note that, much like the code itself, this manual is under continuous development. There are many aspects that it does not touch at all, or explains too briefly. If there is a topic that you found particularly confusing, or you would like to see expanded, we would appreciate your feedback.

## 1.4 Troubleshooting and contact

GraspIt! includes two main resources: this manual, and a complete code reference. We have put a lot of work into both of them, and we strongly encourage you to try using them to solve your problem before contacting us. If that fails, we will try to provide support via email; write us at `cmatei@cs.columbia.edu`. General comments on the simulator, and any patches or improvements to the code are always appreciated.

## 1.5 Authors and acknowledgements

**Andrew Miller** and **Matei Ciocarlie** are the main authors of GraspIt!, having designed most of its features and implemented most of the code. The GraspIt! simulator was developed in the Robotics Lab, Department of Computer Science, Columbia University, under the guidance of **Prof. Peter Allen**.

Many people contributed to GraspIt!, either in the form of new ideas, suggestions or guidance, or by implementing new features in code. These include: **Prof. Jeff Trinkle**, who provided valuable advice on the dynamics system; **Danica Kragic** who developed the real time vision system allowing GraspIt! to work with real robots; **Prof. Henrik Christensen** and **Steffen Knoop** who helped build the automatic grasp planner; **Raphael Pelossof**, who implemented the first GraspIt! based machine learning approach; **Claire Lackner**, who helped design and implement the soft finger contact algorithms. We would like to thank all of them for their contribution.

Thanks also to Prof. Gerd Hirzinger and Dr. Max Fischer, Prof. Contantinos Mavroidis and Katheryn DeLaurentis, Dr. Myron Diftler, Marco Reichel and the The Shadow Robot Company for providing us with models of their robotic hands. We would also like to thank **Willow Garage** for their support towards this new release.

# 2 Installation

The GraspIt! installation process consists of two steps. First, you must install a set of external libraries, listed below. The installation for each library is further detailed in the system specific sections of this installation guide. Then, you must compile the GraspIt! code itself and set up the environment variables.

GraspIt! needs the following external libraries:

**Qt** Qt is a cross-platform application and UI framework. It allows GraspIt! to have a platform-independent windowing system, dialogs etc. GraspIt! currently uses Qt version 4, which is available under two different licenses. This installer assumes you will be using the open source version, see the Qt website for more details on licensing options.

**Coin** Coin is a Scene Graph API that GraspIt! uses for all of its 3D rendering needs. It is an Open Inventor clone. At this time, the latest version of Coin is Coin 3, but due to some compatibility problems we advise using Coin 2 with GraspIt!. Like Qt, Coin is available under a dual licensing model; here we assume you will be using the open source license. See the Coin3D website for more licensing details.

**SoQt** SoQt is a GUI binding, essentially the "glue" between Qt and Coin. It is available, under the same conditions as Coin, from the Coin website. The latest version of SoQt that we have tested GraspIt! with is SoQt 1.4.1, which provides seamless integration with Qt4.

**Lapack** Lapack is a library for scientific computing that GraspIt! uses for various matrix related tasks, such as matrix multiplication, linear system solving, singular value decomposition, etc.

The GraspIt! project information comes in the form of a cross-platform Qt project file, which is processed by QMake. Depending on your system, you will convert this into either a Makefile or a Microsoft Visual Studio project file.

## 2.1 GraspIt! Installation - Linux

### Qt

Download Qt for C++ from the Qt website.

Follow the installation instructions (this usually amounts to just doing `configure` and `make`). The installation automatically goes to `/usr/local/Trolltech/Qt-4.4.3`, unless you specify a different installation path. Set the following environment variables:

- `QTDIR` - the path to your Qt installation
- `PATH` - add the path to `$QTDIR/bin`

On Ubuntu systems it is probably possible to get Qt from the package manager. We have not tested this, but you will probably need all of the following packages:

- libqt4-core
- libqt4-debug
- libqt4-dev
- libqt4-gui
- libqt4-qt3support

### Coin

Download Coin3d from the Coin3D website and follow installation instructions. The examples below assume you choose to install Coin in `/usr/local/Coin` (i.e. use `configure --prefix=/usr/loc`
Set the following environment variables:

- `COINDIR` - the directory where you installed Coin (e.g. `/usr/local/Coin`)
- `PATH` - add `$COINDDIR/bin` (e.g. `/usr/local/Coin/bin`)
- `LD_LIBRARY_PATH` - add `$COINDIR/lib` (e.g. `/usr/local/Coin/lib`)

### SoQt

Download SoQt from the Coin3D website and follow the installation instruction. The configure script needs the path to your Coin installation - e.g. `configure --prefix=/usr/local/Coin`. Then just type `make install`.

### Lapack

On Ubuntu, all you need to do is install the following packages form the package manager:

- libblas-dev
- liblapack-dev

### GraspIt!

Download and unzip the GraspIt! code itself. Set the following environment variable:

- `GRASPIT` - the directory where you unzipped GraspIt!

Build QHull. Go to the `$GRASPIT/qhull` directory and type `make`.

Create the GraspIt! Makefile from the Qt project file. Edit `$GRASPIT/src/graspit.pro` to suit your system and installation needs. Then type `qmake graspit.pro`.

Build and go. From `$GRASPIT/src` type `make`.

## 2.2 GraspIt! Installation - Windows

We assume you are using the Microsoft Visual Studio compiler. Some of the paths provided as examples in this installer are taken from MS Visual Studio 2003; the change to your particular version of the compiler should be straightforward.

### Qt

Download Qt for C++ from the the Qt website.

GraspIt! is currently tested using Qt 4.4.3. Download the regular version, not the `mingw` installer. Then unzip it to a directory of your choice.

Set the following environment variables:

- `QTDIR` - the directory where you unzipped Qt (e.g. `C:/Qt/4.4.3`)
- `QMAKESPEC` - this has to be set depending on your c++ compiler and platform. See `$QTDIR/mkspecs` for a list of options. If using Microsoft Visual Studio 2003, set this variable to `win32-msvc2003`.
- `PATH` - add `$QTDIR/bin` to your `PATH`

You might also need to add to some compiler-specific paths to your environment variables. If Qt's configure script fails, try to look what file it failed to find, then add the respective path to either `PATH`, `INCLUDE` or `LIB`. Also be aware of where in the particular environment variable you add a certain path - when searching for a file, the system will use the first one that it finds, which might not be the one that you intended. Here are, as examples, the paths that need to be added when using MS Visual Studio 2003:

- `PATH`:
  - the path to `nmake.exe` (e.g. `C:/Program Files/Microsoft Visual Studio .NET 2003/Vc7/bin` )
  - the path to the common dlls (e.g. `C:/Program Files/Microsoft Visual Studio .NET 2003/Common7/IDE` )
- `INCLUDE`:

  - C:/Program Files/Microsoft Visual Studio .NET 2003/Vc7/include
  - C:/Program Files/Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Include

- LIB:
  - C:/Program Files/Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Lib
  - C:/Program Files/Microsoft Visual Studio .NET 2003/Vc7/lib

Follow the instructions provided with the Qt distribution for installing (note: installation might take a couple of hours). We recommend building both release and debug versions of Qt, so you have all your options for linking later on. Use `configure -debug-and-release` for that.

WARNING: Other programs use Qt binaries, and often include the Qt dll's (such as `QtGui4.dll`) in their distribution. If a path to some other version of a Qt dll exists in you `PATH` environment variable **before** `$QTDIR/bin`, the system will link GraspIt against the wrong version, often causing failure to run. To be sure that you are linking against the right dll's, put `$QTDIR/bin` at the beginning of your `PATH`.

### Coin

Download Coin3d for your C++ compiler from the Coin3D website.

We have tested the installation of GraspIt! using Coin 2.4.6. If using MS Visual Studio 2003, we recommend downloading Coin 2.4.6 VC7 binaries, no installer.

Set the following environment variables:

- `COINDIR` - the directory where you unzipped Coin (e.g. `C:/Coin3D/2.4.6`).
- `PATH` - add the path to `coin2.dll` (e.g. `C:/Coin3D/2.4.6/bin`).

### SoQt

Download SoQt from the Coin3D website.

You will need to build SoQt yourself from source code. The source code comes with MS Visual Studio solution files, choose the appropriate one depending on your compiler. Again, we recommend building both debug and release versions. After the build completes, take a look in `$COINDIR/bin` to make sure that the appropriate SoQt dlls (`soqt1.dll` and/or `soqt1d.dll`) have been built.

### Lapack

Use your favorite Lapack implementation. For example, you can obtain a trial version of Lapack included with the  (MKL).

**GraspIt!**

Download and unzip the GraspIt! code itself. Set the following environment variable:

- `GRASPIT` - the directory where you unzipped GraspIt! (e.g. `C:/Documents and Settings/your_user` `Documents/Graspit`).

Build QHull. Open and build the MS Visual Studio solution `$GRASPIT/qhull/windows/qhull.sln`. Once again, we recommend building both debug and release versions. Make sure `qhull.lib` has been installed in the appropriate directory (`$GRASPIT/qhull/windows/Debug` and/or `$GRASPIT/qhull/windows/Release`).

Edit `$GRASPIT/graspit.pro` to suit your particular installation. Then, create the GraspIt! MS Visual Studio project file from the Qt project file. Execute the following command from inside `$GRASPIT` in a command prompt:

```
qmake -t vcapp -o graspit.vcproj graspit.pro
```

This will create your GraspIt! MS Visual Studio project file. Open `graspit.vcproj` in MS Visual Studio. Build GraspIt! and run.

**IMPORTANT**: based on the choice of Debug vs. Release made in the `graspit.pro` file, make sure the appropriate configuration (Debug or Release) is **also** selected in MS Visual Studio. This ensures linking against the correct Qt libraries.

**WARNING**: it is not enough to switch between Debug and Release builds using only the option in in MS Visual Studio. This will link against the appropriate Qt libraries but NOT against the appropriate Coin, SoQt and QHull libraries! The correct way of choosing a Debug or Release build is to also edit the `graspit.pro` file.

# 3 Getting Started

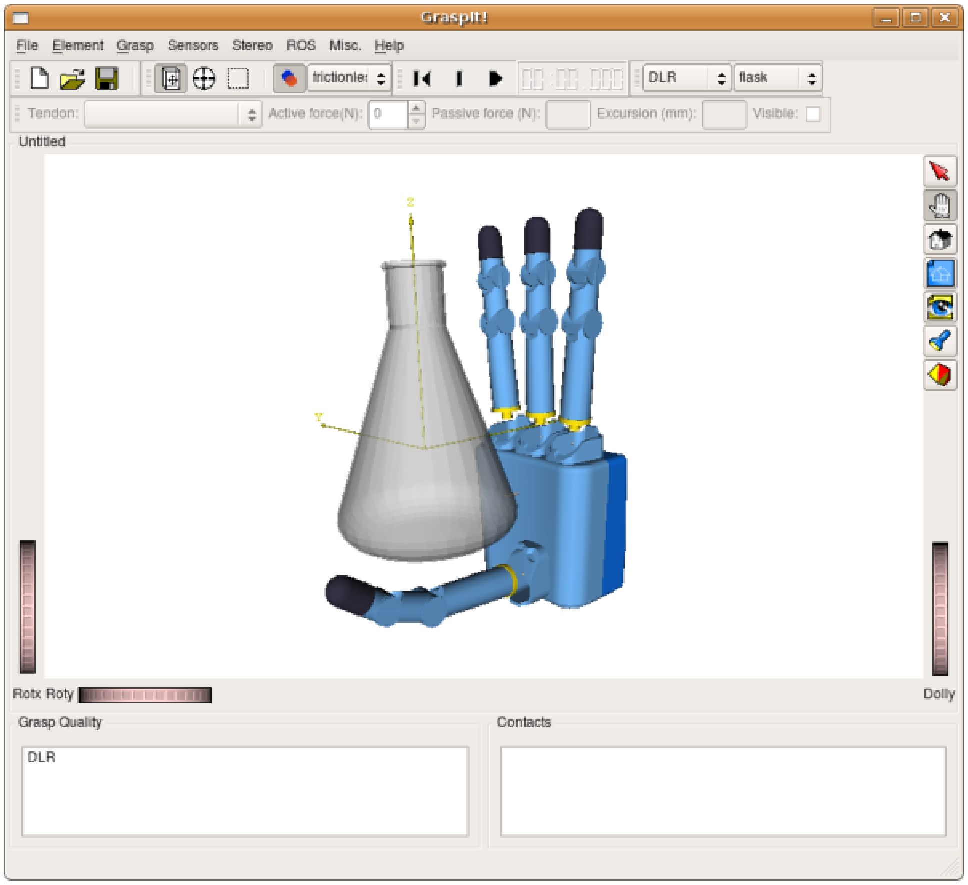
To get started, use the File → Open menu, and load the simulation world file `dlr_flask.wld`. Note that by default GraspIt! will look for world files in `$GRASPIT/worlds`. This is a very simple simulation world containing nothing except a hand (the DLR model) and an object (a flask).

In general, you can also start with an empty simulation world and populate it by importing robots and objects, one at a time, using the Import options in the File menu. You can then save a simulation world into a world file, like the one that we just opened. In this quick tutorial, we will be using a couple of simulation worlds supplied with the distribution.

## 3.1 The main window and controls

The most part of the GraspIt! main window is occupied by the Inventor viewer, which renders the virtual world. On the right side there is a vertical toolbar: this is the Inventor toolbar which is responsible for camera interaction.

The first two buttons on the Inventor toolbar determine which state the viewer is in. When  is selected, the viewer is in **Interaction mode**. This is the only mode in which you can interact with the objects in the simulation world. When  is selected, the viewer is in **Camera mode**. This is the only mode in which you can move the camera.




You can also toggle between **Interaction mode** and **Camera mode** by pressing the `<ESC>` or `<ALT>` keys, although this seems not to work on all systems.

### The Camera mode

When the viewer is in **Camera mode**, you can move the virtual camera in the following ways:

- **Rotate** - Hold down the left mouse button and drag;
- **Translate** - Hold down `<CTRL>` and the left mouse button and drag; alternatively, hold down the middle mouse button and drag;
- **Zoom** - Hold both the left and middle mouse buttons and drag, or rotate the wheel on a wheel mouse.

In addition, the following buttons on the Inventor toolbar are useful:

-  - This automatically moves the camera so that the entire scene fits in the viewer window;
-  - The seek tool allows you to click on an object in the scene. After you click, the camera zooms in on the object, which also becomes the center of rotation.

Take a moment to move the camera around and familiarize yourself with its controls.

### The Interaction mode

When the viewer is in **Interaction mode**, you can interact with the objects in the scene. The type of interaction is determined by the following button in the GraspIt! toolbar: :

- \- **Translate or set joints**. Clicking on a body or the base (palm) of a robot causes a box to be drawn around it. This box acts as a translation manipulator. By clicking on the sides of the box and dragging the mouse, the body or robot can be translated in 2 dimensions that are aligned with the face of the box that was clicked. Holding down `<SHIFT>` constrains this motion to one axis. If a robot is selected and moved.

  Clicking on a kinematic chain causes joint draggers to be drawn for each DOF on that chain. You can then use these draggers to move the joints of the robot.

- \- **Rotate**. Clicking a body or robot base link brings up a spherical rotation manipulator. Dragging the ball allows rotation of the selected item in three dimensions. By clicking one of the stripes, the body can be rotated about one axis at a time. By dragging the cross hairs, the ball can be re-centered to rotate about a different point. The limitations concerning connected robots applies to rotation as well.

- \- **Select**. Clicking a body will select it, and this is indicated with a wireframe overlay. Holding down `<SHIFT>` allows multiple bodies to be selected, or clicking on an already selected link of a robot will select the whole robot. Once a body (or bodies) has been selected, its collision and material properties are shown in the next part of the toolbar. They can then be changed or additional properties can be changed using the menu item described below. After a body is selected, the user may remove the body from the world by pressing `<DEL>`.

Try to use the tool to flex the fingers of the robot. Note that once a finger touches the object, a contact is marked and no more flexion is allowed. The same behavior applies for moving objects or robots around.

The following buttons in the GraspIt! toolbar apply to the currently selected body (if any):

- \- **Toggle Collisions**. There are 3 different ways to use this property:
  - if no body is selected, this button allows collision checking for the entire world to be switched on or off.
  - if one or more than two bodies are selected, this button sets the collision checking for those bodies. A body that has collision checking turned off can pass through ANY other body.
  - if two bodies are selected, this button allows collision checking between ONLY that pair of bodies to be disabled.

- - **Material**. This sets the material for the selected bodies, which affects the coefficient of friction when contacts arise.

Try to disable collisions for the entire simulation world. Note that now you can move objects or flex fingers freely, even if that results in a collisions. Make sure you move all objects out of collisions **before** you re-enable collision checking; otherwise, you will not be able to move them around.

You can also create one or more contacts between the hand and the object. Once you have a contact, select one of the bodies in contact (such as the robot link that is touching the flask, or the flask itself) and change its material. Notice how the friction cone that marks the contact changes as well.

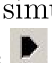
## 3.2 Grasp example

Start by loading the simulation world `dlr_flask.wld` again, to make sure all the world elements are in their original positions. The use the menu Grasp → Auto Grasp. This will cause all the fingers of the robot to flex (more details can be found in the ) until contact with the flask prevents all further motion. You now have a grasp.

Use the Grasp → Quality Measures... menu to create a new quality measure that will be used on this grasp. By default, the quality measure dialog that appears will create a new quality measure called **New Quality Measure** of the **Epsilon** type using an L1 Grasp Wrench Space. Click **Add/Edit**, and then click **OK**. The new quality measure, along with its value, will be displayed in the lower left part of the GraspIt! main window.

You can also create a projection of the Grasp Wrench Space for this grasp. Use the Grasp → Create GWS Projection menu. Then click the three checkboxes marked **tx, ty** and **tz** and click **OK**. GraspIt! will display the space of forces that this grasp can apply without a net torque. Note that if you change the camera in the main GraspIt! viewer, the camera that shows the GWS projection moves as well. The axes of the GWS projection are always aligned with the axes of the main viewer.

## 3.3 Dynamics example

Start by loading the simulation world `barrettGlassDyn.wld`. Then, start the dynamics engine by pressing the  button on the Dynamics toolbar. Note that the robot joints move slightly and the glass slowly rolls on the table. The PD controllers in the robot joints are simply maintaining the current position against gravity.

Use the Grasp → Auto Grasp menu to close the fingers of the hand. Note that the hand starts closing, then lifts the glass into its grasp. After the grasp stabilizes, select the glass and change its material properties to `frictionless`. The glass then slips out of the grasp and ends up rolling off the table. You can pause the dynamics engine at any time by clicking the pause button in the dynamics toolbar.

## 3.4 The menus

Here is a subset of the functionality in the menus (I hope to update this section soon):

- **File Menu**
  - **New** Empties the current world and resets the simulation time.
  - **Open...** Loads a new world configuration.
  - **Save, Save As...** Saves the current world configuration. Velocities are not saved.
  - **Save Image...** Renders the scene using the current camera and saves the image in jpg format. The image will be antialiased.
  - **Import Robot...** Loads a robot from a robot configuration file and places it at the world origin. After importing a robot or body, click view all if the imported object does not fit in the viewer.
  - **Import Object...** Loads an Inventor model and places it in the world as a graspable body. The Inventor file must have mass and material defined in the comments. The body is made transparent so that contacts can be seen.
  - **Import Obstacle...** Loads an Inventor model and places it as a static object in the world. These objects may be moved by the user but their motions are not computed during dynamic simulation.
  - **Edit Settings...** Allows the user to change persistant program settings. The settings are stored in the registry in Windows, and in an rc file in Linux. See below for a description of this dialog box.
  - **Exit** Exits the program.
- **Element**
  - **Translate** Same as translate toolbar button. See above.
  - **Rotate** Same as rotate toolbar button. See above.
  - **Select** Same as select toolbar button. See above.
  - **Collisions ON/OFF** Same as collision toggle toolbar button. See above.
  - **Body Properties...** Brings up a dialog box that allows the user to edit the properties of the currently selected bodies. See below for a description of this dialog box.
- **Grasp**
  - **Auto Grasp** Starts an auto-grasp. When the dynamics are not running, this closes the fingers of a hand until joint limits are reached or contacts prevent further motion. The relative velocity of the joints is defined in the robot configuration file. When dynamics are running, a trajectory generator will create a position trajectory that will close the fingers. The joint controllers will then use these positions as set points and adjust the joint forces.
  - **Create GWS Projection...** Opens a dialog box to allow the user to choose a projection of the 6D grasp wrench space. After the projection is chosen, GraspIt! opens a new window showing the projected wrench space. The volume is updated each time the grasp changes.

- **Quality Measures...** Allows the user to create a new quality measure that will be evaluated each time the grasp changes. (More documentation on this soon.)
- **Planner...** Opens a dialog box containing settings for the automatic grasp planner. At this time this only works with the Barrett hand.

# 4 Main Data Types and Data Files

There are three main types of entities in GraspIt!. The first one is a **Body**, which is characterized by having some geometry (a shape), a transform (its position) and material properties (such as friction coefficient). The second one is a **Robot**, which is comprised of multiple Bodies (such as its links) as well as information on kinematics, actuation, etc. Finally, a **World** groups together instances of the other classes and places them in the correct positions relative to each other. Each of these classes has its own data file format, which GraspIt! can load (but, with the exception of the World, not save to). We will detail all of them in this section.

## 4.1 Bodies

There are two main types of bodies that exist in a GraspIt! simulation world: static bodies (also known as obstacles) and dynamic bodies (such as robot links and objects). Static objects do not participate in the dynamics, but provide collision surfaces for dynamic bodies. Note that this difference mainly applies when the dynamic engine is being used; otherwise, dynamic bodies can be used as static bodies as well. This document describes what makes up a basic GraspIt! body, and what a dynamic body adds to that definition.

Every GraspIt! body has the following data associated with it:

**Geometry:** this describes the shape of the body. It is stored as an Inventor scene graph, a format similar to VRML. This structure can contain pure shape nodes such as cubes, spheres, cylinders, and cones, as well as sets of 2D polygons that define a surface. Although GraspIt! (through the Inventor scene graph) can display all of these geometry types, the collision detection system only works with triangles. When any body is imported to a GraspIt! world, it is faceted into triangles, and these are used for detecting collisions and finding contact points. The units in these files are assumed to be millimeters. Note that the origin of a body's coordinate system is the origin of its geometry, as loaded from a file or created by the user. This can be a tricky aspect, as the origin of a body's geometry is fairly arbitrary. To counter this, dynamic bodies also use the notion of center of mass, explained below. However, the origin of a body's coordinate system is always the origin of it's geometry.

**Material:** the material affects the amount of friction possible at contacts on this body. For each pair of materials we define a coefficient of friction. When the dynamic engine is not used, GraspIt! uses a static coefficient of friction for all bodies. When not using dynamics, this coefficient only affects grasp quality computations, not the relative motion of the bodies. During the dynamic simulation, the coefficient of friction affects the relative motion of the bodies in contact. We also use a dynamic coefficient: if the relative speed at a contact point exceeds a threshold currently set at 1mm/sec, a kinetic coefficient of friction is used.

**Transform:** each body keeps track of the 3D position and orientation of its body frame relative to the GraspIt! world coordinate system.

**Name:** a body's name is currently derived from the its filename, except in the case robot links, which are named using the robot's name and their kinematic chain and link numbers.

Dynamic bodies also have the following properties:

**Mass:** this is expressed in grams.

**Center of mass:** this is a 3D position expressed relative to the body coordinate system. There are two main uses for this. The first one is to provide a stable point for grasp quality computations to use as reference. The second one is to be used as a reference point for transformation between forces and torques in dynamics. It should be more stable than the origin of the body's coordinate system, which is arbitrary.

**Inertia tensor:** this is the standard 3x3 mass distribution matrix. It is expressed relative to a coordinate frame that is aligned with the coordinate system of the body, but positioned at the center of mass. When stored in a file, it is scaled by 1/mass so that changes to the mass can be made by changing only the mass value above.

**Dynamic state:** two values, q and v, store the current position and velocity of the body's center of mass relative to world coordinates. q is expressed as a 7x1 vector: the first three values are the position, and the last four are the rotation in quaternion form. v is expressed as a 6x1 vector: the first three values are the linear velocity of the body, and the last three are the rotational velocity. When the dynamics updates each body state, the body transform is also updated. If a body is moved in the static mode, the position value of the dynamic state is also updated.

### Body Files

Currently, GraspIt! employs the following convention: a Body file is just the file that stores the geometry of the body, either in Inventor (.iv) or VRML (.wrl) format. The geometry part is read in straight by the scene graph. However, the geometry file can be enhanced by adding some GraspIt! specific information at the beginning. All of this information must be placed as Inventor comments (lines starting with the character #) so that it does not affect the process of loading the geometry. None of the GraspIt! specific information is required. If all of it is missing, Inventor will read in the geometry from the file and GraspIt! will assign default values to all of its parameters.

GraspIt! parameters follow the convention:

`#parameter_name value`

Where the # sign is needed to "comment out" the line from Inventor's perspective. For the parameter_name , we have the following possibilities:

**material** the value is a string identifying the name of the material. Possible values are: frictionless, plastic, metal, wood, stone, rubber. Example:

```
#material plastic
```

**mass** the value is the mass of the object, expressed in grams. Example:

```
#mass 300
```

**cog** stands for center of gravity. The value is composed of 3 entries, showing the coordinates of the center of gravity relative to the origin of the geometry. Example:

```
#cog 10.0 0.0 100.0
```

**inertia_matrix** the value is composed of the 9 entries in the body's inertia matrix, on three separate lines. Example:

```
 #inertia_matrix
#4853.0 -1.1196 -6.5156
#-1.1196 4853.0 47.542
#-6.5156 0.0 2357.6
```

For more examples, see the body files included with the distribution, in `$GRASPIT/models/objects`.

In the future, we would like to separate the GraspIt! parameters from the geometry, and get rid of this crude parsing method. We are considering an XML-like language, and parser, for specifying any body parameters inside a file. This might be implemented in the next release of GraspIt!.

## 4.2 Robots

A Robot is made up of multiple links, connected into kinematic chains. A link is simply a dynamic body, as described above. A Robot always has a base link (called "palm" for hands) and one or more kinematic chains attached to it. Each chain is in turn made up of a succession of links, connected by joints. In order to define a robot, two things are needed: the Body files for all the links that are part of the Robot, plus an overall Robot configuration file, which has all the kinematic information and references the appropriate body files for the links. Here we describe the structure of the Robot configuration file.

Robot configuration files can seem daunting at first, and they are a bit annoying to get used to. However, you can start from one of the many robots that are included with this distribution, and use it as a starting block for your own robot that you are trying to build. For the moment, a Robot configuration file is a simple text file which must be parsed in the strict order given below. In the future, we hope to move to more flexible XML-like format. In Robot configuration file, any line that begins with the # character is considered a comment. Blank lines are ignored.

A Robot configuration file has three sections. The first one has the overall information about the Robot (type, number of kinematic chains, number of DOF's, etc). The second part contains the information for each kinematic chain. The third part contains some optional additional information. Most Robot files included with this GraspIt! distribution are also commented to show what the purpose of each line is.

In this example, we will walk through the file for the Robonaut hand. The first item in the file tells GraspIt! whether this is a generic robot or hand or a particular subclass of one of those. A hand should use the "Hand" class type. In some cases, if a robot has special features or its own inverse kinematics algorithm, it is necessary to use a subclass of these generic types, such as "Barrett", or "Puma560":

```
#Robot ClassType
Robonaut
```

The next item is the filename for the palm link:

```
#Palm filename (considered a link)
right_palm.iv
```

This is followed by information on the DOF's. First comes the number of DOF's. Then, for each DOF, a line with the information for initializing that DOF. Note that a DOF can be connected to one or more joints in the kinematic chains, this information will be supplied later in the Robot configuration file. For more details about DOF's and joints in GraspIt!, see the Joint Coupling and Underactuated Hands chapter.

For each DOF, the line with initialization information contains the following:

- a letter showing the DOF type. For the Robonaut hand, all DOF's are of the type "rigid", depicted by the letter "r". This is the most common type of DOF in GraspIt!. Unless you are building robots with coupled joints (multiple joints connected to a single DOF), you can always use this type of DOF.
- the default velocity for that DOF during an autograsp operation. This is used to pre-define the "closing" motion of a hand. For anthropomorphic hands, these pre-defined directions tell GraspIt! how to move each DOF in order to "autograsp", or how to "close the hand". This generally means moving the DOF's of the MCP, PIP and DIP joints in the "flexing" direction, and no movement for the abduction - adduction DOFs.
- the max force the DOF can apply to each joint it is connected to. The unit is N * 1.0e6 * mm for torques and N * 1.0e6 for forces.
- the Kp and Kv coefficients for a PD force controller built into the DOF
- the visual scale of the dragger that allows the user to control this DOF through the GraspIt! GUI.
- a number of optional parameters, depending on the DOF type. For the "rigid" DOF, no more parameters are needed.

```
#number of DOFs
14

#for each DOF:
#type default_velocities_for_auto_grasp max_effort_in_N_or_Nm dragger_scale
```

```
r 0.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
r 0.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
r 0.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
r 1.0  5.0e+9 1.0e+10 1.0e+7 20
```

The next piece of information contained in the Robot config file is the number of kinematic chains (also called "fingers" for hands):

```
#Number of fingers
5
```

We then move on to the next section, which contains information for each of the kinematic chains. We will only show here one of the five chains in the Robonaut hand, all the other chains follow the same convention. The first entries are the number of joints and links in the chain. Note that these are not always equal, as you can have up to 3 joints between two links.

```
#----------------f1 - index --------------------
#number of joints
4

#number of links
4
```

Next comes the transform from the origin of the palm (which is also the origin of the robot) to the base of this chain, which is where the first joint in the chain is places. Any chain starts with a joint.

```
#Transforms from palm origin to base of the finger
t 87.883873 -27.044726 -16.619068
R
0.997205 -0.074709 0.000000
0.074425 0.993411 -0.087156
0.006511 0.086912 0.996195
```

The next block is dedicated to the joints in the chain: one line for each joint, with the Denavit-Hartenberg parameters and some additional joint information. For each joint, the following entries are present:

- the four D-H parameters, in the order theta d a alpha. Either theta or d must be linked to a DOF number, as shown below.
- the next two values are the joint limits.
- the next values are optional (not present in the Robonaut file) and can contain things like joint friction coefficients, spring stiffness etc.

```
#Joint Descriptions (1 joint per line)
#(joints are ordered from closest to palm outward)
#(d# indicates DOF that this joint is connected to)
#(linear equations are of the form: d#*k+c [no spaces!])
#theta d a     alpha min max
d3      0 8.0518 90      -20 10
d4+-5.9 0 45.72   0      -10 85
d5      0 26.67   0        0  85
d5+5.9  0 0       0        0  85
```

Finally, we have the block that puts links and joints together. This block consists of a list of connection types and links. The connection type tells us how each link is connected to the one before. It can be one of the following: "Revolute", "Prismatic", "Universal", "Ball", or "Fixed". Depending on which one is used, the chain will take some of the joints from the joint list above, and put them together to create a connection of that type. For example:

- Revolute: the link is connected to the one before by through a single revolute joint
- Universal: the link is connected to the one before by two revolute joints, usually with perpendicular axes
- Ball: the link is connected to the one before by three revolute joints, usually with perpendicular axes

The link entry is simply a Body file that that link can be loaded from. This is followed by the `lastJointAxis` number which tells GraspIt! the index of that last joint axis that affects that position of that link. The joint axes are numbered from 0. We are hoping to get away from this convoluted way of specifying connections pretty soon.

```
#Link Descriptions (1 link per line)
#filename lastJoint
#(links are ordered from closest to palm outward)
#(lastJoint is the last joint in the chain which can affect this link)
Revolute
yoke2.iv 0
Revolute
```

```
pph3l.iv 1
Revolute
mph4.iv 2
Revolute
bph3.iv 3
```

The last part of the Robot file contains optional information. This usually includes things such as Eigengrasp information, connection to a Flock of Birds sensor, etc. These are described in more detail in the dedicated chapters of this manual.

## 4.3 Worlds

In GraspIt!, robots and bodies populate a simulation world. This document describes how these elements can be added or deleted from a world and describes the format of a world (.wld) file, which stores the current state of the world.

When GraspIt! begins the world is empty. The user may either load a previously saved world by choosing File → Open, or populate the new world. To import an obstacle (a static body) or an object (a dynamic body), File → Import Obstacle or File → Import Object, and then choose the Body file (see the previous section on bodies). Note that any Body file (regardless of whether it's meant for a static or dynamic body) can be loaded as an obstacle (GraspIt! will just ignore the dynamic parameters). However, when a body file is imported as an Object, GraspIt! will automatically instantiate it as a dynamic body. It will also try to find the dynamic parameters in the body file and, if it can not find them, assign default values. Be aware that the default values occasionally have unpredictable results.

To import a robot, use File → Import Robot, open the correct robot folder, and select the robot configuration (.cfg) file.

To delete a body, select it, and then press the ¡DELETE¿ key. To remove a robot, first select the entire robot (by double-clicking one of the links when the selection tool is active) and press the ¡DELETE¿ key.

Note: newly imported bodies or robots always appear at the world origin. You can move existing bodies out of the way before importing a new one. If you do not, than the newly imported body will overlap with an old one, and you will have to temporarily toggle collisions in order to move one of them out of the way.

When the user selects "Save" in the file menu, GraspIt! saves the current world state in a text world (.wld) file. In the future this will be stored in an XML format. The first line of the file (e.g. `#GraspIt! version 2.0beta`) identifies this file as a GraspIt! file. Each block of lines following that describes on element of the world. The end of a block is marked with a blank line. Each block begins a line describing the type of element. This must be one of: Obstacle, GraspableBody, Robot, Connection or CameraFull.

**Obstacle:** the next line identifies the file name associated with the body. The filename is relative to `$GRASPIT`. Next line should be a material type, and finally there is a transform that locates the body with respect to the world coordinates. The transform is stored by using a "T" followed by 4 numbers surrounded by parenthesis to represent the body orientation as a quaternion, and 3 numbers surrounded by brackets to represent the body position.

**GraspableBody:** the block is the same except the material line is omitted (the material is found in the body file).

**Robot:** the second line after this keyword identifies the robot configuration file. The third line contains values for DOF in the robot. Note that this might mean a single number per DOF or more information, depending on the DOF type. The last line is the transform described above.

**Connection:** indicates a connection between two robots. The second line of the block contains 3 numbers. The first is robot index of the parent robot, starting at 0. The second number is the kinematic chain number on the parent robot that the other robot is attached to. The third number is the index of the child robot. The next line (can be empty) specifies a body that is optionally used as a mount piece between the two robots. The last line of the block is the constant offset transform between the last link of the parent's kinematic chain and the base link of the child robot.

**CameraFull:** specifies the position, orientation and focal point of the camera. The next line contains 8 numbers: the first 3 are the camera position in World coordinates, the next 4 are the camera orientation specified as a quaternion, and the last one is the focal distance of the camera.

For an example, take a look at the `barrettGlassDyn.wld` file supplied with this GraspIt! distribution.

# 5 Collision and Contact Detection

A key component of the GraspIt! engine is the collision and contact detection. When collisions are enabled, anytime you interact with a GraspIt! world, either by moving an object or using the joint draggers on a robot, the engine will prevent any movement that brings two objects in collision and stop it at the moment of contact. Points of contact are then marked with contact indicators.

Important note: although GraspIt! (through the Inventor scene graph) can display all of these geometry types, the collision detection system only works with triangles. When a body geometry is loaded, GraspIt! will use the Inventor scene graph to facet it into triangles, which are then used to build the body model for the collision detection system.

In GraspIt! there is a very important difference between contact and collision. We define collision as two bodies that are interpenetrating, no matter by how much. In general, most algorithms in GraspIt! consider collision to be an **illegal** state and will attempt to find collision-free states. Contact is defined as two bodies that are closer together than a given threshold, but are **not** interpenetrating. In general, this threshold is set to be $0.1mm$. If you would like to change this threshold, you will have to go inside the code: it is the `THRESHOLD` static member of the `Contact` class.

You can also enable or disable collisions, for either the entire simulation world, one particular body, or a pair of bodies. This is all done using the `Toggle Collision` switch in the main GraspIt! interface. Depending on what is selected when you press the button, the following will happen:

- if nothing is selected, collisions are toggled for the entire simulation world
- if a single body is selected, collisions are toggled for that particular body (it can / can not collide with any other body in the world)
- if a pair of bodies is selected, collisions are toggled for that particular pair (they can / can not collide with each other; collisions with all the other bodies in the world are unaffected).

Most GraspIt! functionality that involves moving bodies around (including user interaction with joint draggers) works as follows:

- move the bodies freely as long as there is no collision;
- when collision is detected, attempt to interpolate between the collision state and the last known collision-free state until the bodies are no longer colliding, but are separated by less than the contact threshold;
- find all points where two bodies are separated by less than the contact threshold and mark them as contacts.

## 5.1 Contacts

In GraspIt! a contact is defined as any point where two bodies are separated by less than the contact threshold, but **not** interpenetrating. The collision detection engine will find these points for you, and also do some pruning, as explained below. A GraspIt! contact, defined in the `Contact` class, encapsulates the following information:

- the location of the contact on each of the two bodies (the points on the two bodies that are separated by less than the contact threshold)
- the contact normal (defined as the normalized vector between the two points mentioned above)
- the Contact Wrench Space: the space of forces and torques that can be applied at the contact. This is a crucial concept, which is at the base of most grasp quality computations and dynamic simulations. In the simplest case, that of Coulomb friction, this space is a 3D cone. See the Publications section of this document for more details.
- a visual marker showing the location of the contact in the GraspIt! GUI. In the case of rigid body contacts, this is actually a rendering of the friction cone, aligned with the contact normal.

Note that everywhere in GraspIt! we refer to contacts as **points**. The reason is that geometry in GraspIt! never deforms, so we can never explicitly compute **areas** of contact. There are two important aspects though. First, two rigid bodies might be locally similar, so that more than one point is within the contact threshold. If that happens, the collision detection just returns many point contacts in a small area. All these point contacts are then pruned, keeping only those contacts that are on the contour of the area (the boundary of the convex hull of contact locations). According to the theory on contacts that we rely on, this will have no net effect on any grasp quality computations, but will reduce computation time by reducing the number of contacts).

Second, GraspIt! can simulate some of the effects of soft bodies in contact without explicitly computing the deformation. This is done by using a different version of the Contact Wrench Space. See the Soft Contacts chapter of this manual for details.

## 5.2 Software implementation

For many algorithms, the collision engine is the computational bottleneck. It is very important to have an efficient engine, but we also require this engine to be very robust and work well on triangle meshes (as opposed to analytical primitives). The GPL version of GraspIt! comes with its own collision detection implementation, using a number of common bounding box hierarchy methods. However, fast collision detection is a research area in itself, and we are sure there are many ways to improve our implementation. Any bug reports, patches or optimizations for the collision detection engine are highly appreciated.

From a software perspective, we have built the collision and contact detection libraries to be as modular as possible. This allows the complete replacement of our current collision detection with the library of your choice. If you are interested in doing this, check out the `CollisionInterface` class and its subclasses. If you know of a good collision detection library that is fast and robust, works with triangle meshes, and has a GPL-compatible license, we would love to hear about it.

## 5.3 Under development: Multi-threading

We have also implemented a crude multi-threaded support for the collision detection mechanism. This is still very much work in progress, both from a design and implementation standpoint. The overall concept is as follows: if you have multiple threads in your GraspIt! code, each thread can add its own bodies to the simulation world. The rule we have implemented is that bodies from different threads never collide with each other. The only exception is that all bodies collide with bodies from the main thread.

The reason for this implementation is to allow you to test multiple scenarios in parallel, without worrying about collision detection. As an example, suppose you have a hand and a glass sitting on a table, and you want to evaluate many grasps quickly. In this case, you would populate the world with the table and the glass in the main thread. Then you would fire up many threads, each loading its own copy of the hand. Each thread can then tests its own grasps independently, without needing to synchronize with the other threads or worrying about colliding with the hands from other threads. Of course, if your code is single-threaded, you can just ignore all of this and pretend it does not exist.

The general steps for GraspIt! multi-threading are:

- in the main thread, load all the objects that will be shared between threads
- fire up all of your computation threads. In each thread, inform the collision detection mechanism that it is now servicing a new thread (see the CollisionInterface class for

what method to call for this)

- in each thread, load all the objects or robots that are specific to that thread
- in each thread, run any computation like you normally would.

For examples, see the `EGPlanner` class which has support for running in its own thread.

### 5.4 Under development: Object cloning

We also have a mechanism for "cloning" objects in GraspIt! so you can easily create multiple copies of a body or robot without having to load it from a file multiple times and without wasting memory. As far as any GraspIt! algorithm is concerned, a clone is a totally independent body with its own position in the world, contacts, etc. However, under the hood, a clone will share all the scene graph geometry and collision detection bounding box hierarchy with the original. See the `Body::cloneFrom(...)` and `Robot::cloneFrom(...)` methods for details. This mechanism works well, with one exception: we never implemented a nice cleanup phase that handles the case where the original is deleted and the clone still lingers in the world. This situation is almost guaranteed to cause a crash.

In practice, cloning and multi-threading often go together. If you have lots of computation that you might parallelize, you can create clones of your moving objects (usually the robots and hands) and pass them to different threads, where each thread will do its own work. See again the `EGPlanner` class for details - it can run in its own thread using a cloned hand for computations.

## 6 Soft Fingers

The GraspIt! engine never computes geometry deformation explicitly, therefore can not find exact contact areas between soft bodies. However, the frictional implications of soft fingers in contact are too important to be completely ignored for grasp quality computations. The most important effect is that contacts over an area (as opposed to point contacts) can also apply **torsional friction**. The soft finger model in GraspIt! attempts to capture at least an approximation of this effect, without explicitly computing the contact deformation. See the Publications section for complete details on the theoretical aspects of our soft finger contact computation.

In order to designate a body as "soft", specify it's Young's modulus in the properties section of the Body file (see the Data Files - Bodies section of this manual for a description of the properties section). The entry should contains the keyword "youngs", followed by the value in MPa. For example, an entry can have the following form:

`#youngs 1500000`

Any body that has such an entry in the properties section, including robot links, is treated by GraspIt! as a soft body. When a soft body is found to contact another body (irrespective of whether the second body is also soft or not), the contact engine does the following:

- find a set of vertices on the surface of each body in a small area around the contact points. These vertices define the "soft neighborhood" of the contact

- if multiple point contacts are found close to each other, only one of them is kept, and their soft neighborhoods are merged
- fit an analytical surface to the soft neighborhoods. This is done by fitting a surface of the form $z = ax^2 + by^2 + cxy$ to the vertices in the neighborhoods
- the radii of curvature of the analytical surfaces on each body are used to approximate the amount of torsional friction that can be applied at the contact
- the Contact Wrench Space is built in order to also contain the effect of torsional friction. This wrench space is 4D (unlike the Coulomb friction cone which is 3D). Therefore, it can not be displayed as a contact marker. Instead, soft contacts are indicated by displaying a small patch of the analytical surface fit to each body around the contact.
- the resulting Contact Wrench Space affect both grasp quality computations and the behavior of the contact in dynamic simulations.

All of this functionality is encapsulated in the `SoftContact` class; see the code and documentation of this class for details.

Intuitively, this entire computation has the following effect: if the bodies are locally "flat", or if their curvatures match in a small region around the contact, they will produce a larger area of contact for a given normal force. This will in turn lead to larger torsional friction. Conversely, sharp edges in contact, even on soft bodies, will create small torsional friction. The amount of torsional friction is also influenced by the value of Young's modulus specified for each body.

This method captures much of the effects of soft contacts on the kind of simulations that are of primarily interest in GraspIt!. It is important to note though that it is only an **approximation** of the real-life phenomenon. It relies on fitting analytical surfaces to each of the bodies in a small region around the contact. On bodies with very complex or degenerate geometry the fitting procedure can fail leading to incorrect amounts of torsional friction applied. The fitting procedure also is not very good at handling very sharp features such as corners or edges.

# 7 Grasp Quality Metrics

This section is intended as a practical guide on how to access the grasp quality tools in GraspIt!. Grasp quality metrics are an active area of research; for discussions of their theoretical aspects please see the Publications section.

In GraspIt!, a grasp is completely defined by an object and a set of contacts, presumably created by a hand. In the context of grasp quality computations, the posture of the hand, or it's position relative to the object are not important; all that matters is the location of the contacts. The interface to all of the quality computations is the `Grasp` class, which collects the contact between a hand and an object. However, an instance of this class never needs to know about joint angles, etc (with a few exceptions listed below).

In order to create a grasp, you must load a hand and an object into a simulation world, then place them relative to each other and change the joint values of the hand until multiple contacts are formed. To facilitate this process, load the `dlr_flask.wld` world file provided with the GraspIt! distribution. Then use the Grasp → Auto Grasp menu command. This should result in multiple contacts, using all the fingers as well as the palm. We will investigate the quality of the grasp in the following steps.

## 7.1 Grasp Wrench Space metrics

All the grasp quality metrics computed in GraspIt! rely on building the Grasp Wrench Space (GWS) from the individual Contact Wrench Spaces of all contacts (see Publications for theoretical details). There are multiple ways to build a GWS, and then multiple ways to compute a quality metric on one. In GraspIt! a Quality Measure (QM) is more than just a number - it is an object that hangs around from the moment you create it until you dismiss it. At any point, you can ask a QM to update itself and provide you the computed quality value. You can access this functionality through the Grasp → Quality measures menu. The Quality measures window allows you to:

- create a new Quality Measure, and assign it:
  - a name
  - a Quality Measure type
  - a GWS type
- edit (change the name or type) of an already built Quality Measure

Once you have constructed a QM, the main GraspIt! window will display it, along with the most recently computed value, in the space provided in the lower left corner of the GUI.

Note that this dialog said nothing of GWS construction. The GraspIt! GUI does all that behind the scenes. Whenever a QM needs a particular type of GWS, one is constructed. However, multiple QM's can share a GWS: if a QM requests a GWS type that exists already, it will be redirected to the existing GWS rather than building a new one. GraspIt! does the reference counting for you, so when a GWS is no longer needed by any QM, it is deleted. In general, it is building the GWS that is the computationally expensive part of the QM computation.

From a code standpoint, quality measures are organized as follows:

- the `Grasp` class keeps track of the hand, the object and the contacts between them
- the implementations of the `QualityMeasure` interface can compute quality metrics and provide you with the results. Any implementation of QualityMeasure needs an instance of a `GWS` class (but the same instance of the `GWS` class may be shared by multiple `QualityMeasures`).
- the `GWS` implementations actually build the GWS from the set of contacts. This is usually done by computing the convex hull of individual Contact Wrench Spaces using QHull, an excellent library for building high-dimensional convex hulls. Building a GWS is a fairly expensive process (especially when many contacts are involved) and a bottleneck for many GraspIt! algorithms.

In general, a GWS is a 6-dimensional convex polyhedron. As such, it can not be visualized directly. However, GraspIt! allows you to view 3D projections of a GWS. Use the Grasp → Create GWS Projection menu for that. You will need to choose which 3D are fixed (used for projection). Note that a GWS (and its projections) are independent from Quality Measures - you do not need a QM to visualize a GWS. However, if the GWS does not contain the origin (the grasp does not have form-closure), or the GWS is a degenerate 6D object, the GWS construction process might be aborted and the visualization might not show anything. In general, form-closed grasps with a true 6D GWS create correct 3D projections.

### 7.2 Under development: Grasp Force Optimization

All of the QM's discussed above share a common trait: they assume that the amount of normal force that can be applied at each contact is identical, and normalized to magnitude 1. In practice this is not true: a given contact can only apply the level of normal force that is possible given the actuation of the hand. A great amount of theoretical work exists on taking hand actuation into account when evaluating a grasp. In GraspIt! we have two tools for this, but one of them has never been extensively used or tested, and the other is still under development.

The first tool is an implementation of the Grasp Force Optimization approach proposed by L. Han, J. Trinkle, and Z. Li, *Grasp analysis as linear matrix inequality problems*, IEEE Trans. on Robotics and Automation, vol. 16, 2000. GraspIt! has the code necessary for all the concepts found in that paper, however, this part of the code has not been tested or used lately. It can all be found in the `Grasp` class, marked as Grasp Force Optimization (`GFO`) code.

The second tool is somewhat similar, but employs a different mathematical model (Quadratic Programming instead of Linear Matrix Inequalities) and is geared towards underactuated compliant hands. It performs a Quasi-static (QS) force equilibrium computation to see whether contacts are stable given the actuation scheme of the robot. All the details are in the Publications chapter of this manual. The code itself is also in the `Grasp` class, marked as quasi-static analysis routines. However, this is still work in progress in our lab, so use with care. Also see the Joint coupling chapter of this manual for more details on DOF force generation capabilities in GraspIt!.

## 8 Grasp Planning

### 8.1 General concepts and planner types

Grasp planning is one the most useful (and most widely used) tools in GraspIt!. The core of this process is the ability of the system to evaluate many hand postures quickly, and from a functional point of view (i.e. through grasp quality measures). Using a simulated environment allows us to test grasps much much faster than in real life, and also at a lower cost. The quality metrics give us feedback on the grasps, often more than just a binary success / fail outcome. This is therefore the general concept: try out lots of grasps really fast and see which work. Of course, there is an infinite number of possible implementations, optimizations, refinements, etc. that can be played starting from this simplified idea. GraspIt! comes with a couple of grasp planners, each of them different in its own way, but all have roots in the same concept presented above.

The grasp planners within GraspIt! are grouped in three families:

- the Primitive-based Planner, primitive not in the sense that it is ancient but rather in the sense that it uses primitive decompositions for the grasped object.
- the Eigengrasp Planner family, which relies on hand posture space dimensionality reduction.
- the Database Planner family, which relies on a huge database of pre-computed grasps to plan grasps for novel objects.

This section only presents the Primitive-based Planner; the other two families have their own sections in this manual.

All the types of grasp planners have been extensively described in various publications. If you are interested in the machinery behind the scenes and the theory of the planners, the Publications section has many more details than presented here.

## 8.2 The Primitive-based Grasp Planner

The Primitive-based Planner is accessible via the Grasp → Planner menu. It has a couple of restrictions: it only works on the Barrett hand, and only if the user also supplies a primitive approximation of the object to be grasped. When the Planner dialog is opened, GraspIt! will attempt to load the primitive version of the current object from the `$GRASPIT/models/objects/primitives` directory. In order to create a primitive file, see the examples in the primitives directory that are included with the distribution. Note that a primitive file may only include spheres, cubes, cylinders and cones. For more details, see the relevant publication.

The Planner itself goes through 2 stages. The Planner dialog window has two groups, one for each stage. The first stage (accessed through the button group on the left) is to generate many pre-grasps for your object. Pre-grasps are generated based on the primitive version of your object. You can generate as many pre-grasps as your computational resources and allocated time will allow you to test. The number of pre-grasps generated is controlled by the density factors. You can either choose a master density factor (Automated sampling) and allow GraspIt! to do the rest, or choose sampling densities along different dimensions separately. Alternatively, you can pre-specify the exact pre-grasps to be tested by loading them from a file, which is useful for debug purposes. Once you have set the desired parameters, click the Generate button to generate your pre-grasps.

The second stage is to compute the grasps that result from the chosen pre-grasps. Note that grasp execution is done on the actual object (even though pre-grasps are generated on the simplified primitive version). You can also choose which Quality Measure should be used to rank these grasps. If a usable QM exists already, you can select it from the drop-down list. If not, use the New button to fire up the QM creation dialog and create a new one. Once you have set the desired metric, you are ready to test all the pre-grasps by clicking the Test button.

After testing is finished, the hand will be set back to its initial position and the Show button will become enabled. Use the Show button to cycle through the list of found grasps, sorted in order of the Quality Metric.

**IMPORTANT**: you can choose to visualize the testing process by checking the Visualize process box. This means that the process of executing all the grasps will be rendered, and you can see the hand trying out each of them. You must check the Visualize box **before** clicking the Generate button for this to work. Visualization makes for a more compelling demo, but rendering slows down the planning process considerably. For time-sensitive applications, we recommend disabling the visualization.

When rendering is disabled, we have found that the computational bottleneck for the Primitive-based planner is collision and contact detection.

# 9 The Dynamics Engine

GraspIt! has two main modes of operation. In the first one, which has been assumed in all of the previous chapters, the user directly interacts with the simulation world by moving objects around or moving robot joint draggers. The collision detection system disallows any movement that would bring two bodies in collision, and stops any such movement at the point of contact. We refer to this operation mode as "static operation".

In real life however, objects move around as a result of contact and external forces. In GraspIt! these phenomena are taken into account when using the dynamics engine. This engine computes the contact forces that prevent interpenetration and the velocities and accelerations of all the bodies in response to contact forces, joint constraints and external forces. The resulting motion of the bodies is integrated over time in a time-stepping procedure resulting in a full simulation of a grasping process (as opposed to just assessing the quality of a final grasping posture, as we do in static operation).

## 9.1 User interaction in dynamics mode

The key difference between static and dynamic operation is in the way the user interacts with the simulation world. In statics, the user directly moves bodies around, and sets robot joint values. None of these are allowed in statics. The only way the user can affect the simulation world is by setting desired values for the robot DOF's. Built-in controllers for the robot DOF's take care of the rest, simulating true dynamic operation. DOF controllers generate motor forces; as a result of these robot joints move and potentially touch other objects. Bodies are also affected by gravity, and will fall off the world if they are not supported by an obstacle (such as a floor or table). Obstacles are only part of the dynamics engine in the sense that they provide contacts for other bodies. They never move in dynamics.

Use the `Dynamics start` button in the toolbar to start or pause the dynamics engine. While the engine is running, you should see the dynamics timer advancing. During this time, you will see bodies move as a result of gravity, or DOF motor forces if the desired DOF values have been set. For a quick demo, see the Getting started chapter of this manual.

Currently, the only way for a user to specify a desired position for a robot in dynamics mode is to use the AutoGrasp feature. Use the Grasp → Auto Grasp menu just as you would in statics. When dynamics are on, this will set the robot desired DOF values at either the DOF max or min, depending on the default velocity specified in the robot configuration file. In the future, we intend to provide an interface for a user to set desired robot DOF values during dynamics execution.

## 9.2 DOF controllers

The simulated DOF controllers have proven to be very difficult to calibrate and tune. Currently, we are implementing PD controllers and attempt to generate trajectories with smooth acceleration and velocity profiles from current DOF values to desired DOF values. See the `DOF::PDPositionController(...)` function for details. These controllers are **very** sensitive to the proportional and derivative gains used, and we have no better method of setting these than empirical tuning. Even like this, for complex dynamic simulations, we often see erratic behavior. We are hoping to improve the DOF control mechanism in the future.

## 9.3 Implementation

For all the details on the dynamics engine implementation, see the Publications chapter. Here we provide just a quick overview.

The dynamics engine works in time steps. At each time step, two main tasks are performed:

- move all dynamic bodies according to the velocities computed at the previous time step. If any collision occurs, interpolate back in time until the exact moment of contact. Mark all the contacts in the world.
- formulate solve a Linear Complementarity Problem (LCP) that gives us:
  - the contact forces that prevent interpenetrations
  - the joint forces that keep the joints in place
  - the velocities of all the bodies in the world in response to the forces above, as well as external forces (motors, gravity, etc.)
- see the dynamics-related functions in the `World` class for more details.

The core of the dynamics engine, and also its most important computational effort, is the LCP solver. We have implemented a version of Lemke's algorithm for this, found in the `dynamics.cpp` file.

## 9.4 Under development: dynamics improvements

Unfortunately, the dynamics engine exhibits occasional instability, which we would like to address in the future. These might be caused by the LCP solver, the time step integration, or both. It has been suggested by experts in the field that the LCP solver is the primary suspect. We are hoping at some point to replace the current solver with a better one, based on newer methods proposed in the literature.

We would also like to cut down on computation time by providing the LCP with a "warm start" using the solution from the previous time step. This is made more difficult by the fact that contacts are always broken and then re-computed between time steps, making it harder to keep track of previous time step solutions. We have put in place a mechanism for "inheriting" contact parameters between time steps, you will find this in the `Contact` class. However, with the current LCP solver, we have not been able to obtain an improvement using this information. Contact "inheritance" though has been left in place (although disabled for now) in the hope that future LCP solvers might be able to take advantage of it.

# 10 Eigengrasps

Eigengrasps define a subspace of a given hand's DOF space. Assuming the hand has $d$ DOF's, each eigengrasp is a $d$-dimensional vector. A basis comprising $b$ orthogonal eigengrasps can define a $b$-dimensional subspace. Additionally, this subspace needs an origin, which is also a $d$-dimensional vector. This gives the following options:

- define a hand posture using $b$ eigengrasp amplitudes (as opposed to $d$ DOFs).
- find the Eigengrasp subspace projection of a given hand posture which is not necessarily in this subspace.

The term "Eigengrasp" is of our own creation; in the literature (particularly regarding the human hand), the same concept is often referred to as "hand synergies".

For a complete discussion on Eigengrasps and their application in practice, please see the Publications section. This is just a brief overview of Eigengrasps in practice, in this version of GraspIt!.

This distribution of GraspIt! include Eigengrasp information for many dexterous hands. For the human hand, we are providing eigengrasp directions matching those discovered through user studies by Santello et al. (see M. Santello, M. Flanders, and J. F. Soechting, *Postural hand synergies for tool use*, Journal of Neuroscience, vol. 18, no. 23, 1998 for details). The hand model used in that study had 16 DOF. Therefore, the only the 16-DOF version of the human hand included with GraspIt! has all 6 eigengrasps discovered in the study. The 20-DOF version has only the 2 dominant eigengrasps, and since an empirical mapping was done between 16 DOF and 20 DOF, they might not be as accurate as the ones provided with the 16-DOF version.

3 more dexterous hands have eigengrasp information pre-defined in this version of GraspIt!: the Robonaut hand, the DLR hand and the Barrett hand. For the anthropomorphic models (Robonaut and DLR) we have performed an empirical mapping of the 2 dominant eigengrasps from the human hand to adapt them to the robotic hand kinematics. The Barrett hand natively only has 4 DOF (it can be considered an eigengrasp hand by construction, since it has 7 joints). We have still defined 2 eigengrasps empirically, this is the simplest case of using eigengrasps and it will be used as an example in the rest of this chapter.

## 10.1 Loading Eigengrasp Information

For any hand model, eigengrasp information can be defined using a text file which is loaded together with the hand configuration file. Usually, eigengrasp information files are placed together with the rest of the information that defines a robot, such as the configuration file or link geometry files. Here is an example file defining a 2-dimensional ($b = 2$) eigengrasp subspace for the Barrett hand ($d = 4$). The example shown here is found in the file `barrett_eigen.egr` included in this distribution.

- the first line contains the keyword `DIMENSIONS` followed by the number of DOF's of the hand
- each eigengrasp begins with the keyword `EG`. On the next line, a single value containing the eigenvalue associated with this particular eigengrasp. For now this number is not used anywhere in the code, but it might have uses in the future. Finally, on the next line, the $d$-dimensional vector that defines the eigengrasp
- an arbitrary number of eigengrasps can be defined, as long as each is formatted as described above. In this example, we define a 2-dimensional subspace with two eigengrasps.
- the origin of the subspace is defined exactly like an eigengrasp, but it is preceded by the keyword `ORIGIN`. If no origin is found in the file, the system will use a pre-defined subspace origin: for each DOF, the subspace origin is assumed to be at ( maxVal - minVal ) / 2
- the normalization information is optional. If desired, it can be defined like an eigengrasp, preceded by the keyword `NORM`. If this information is not present in the file, no normalization is used

```
DIMENSIONS 4

EG
0.51
1.0  0.0  0.0  0.0

EG
0.25
0.0  1.0  1.0  1.0

ORIGIN
0.0000
1.13 0.79 0.79 0.79

NORM
0.0000
1.57 1.22 1.22 1.22
```

When loading a robot, GraspIt! will look in the robot configuration file for information on what eigengrasps file to load (if any). First comes the keyword `EigenGrasps`, followed on the next line by the eigengrasp file to be loaded. The path to the Eigengrasp file is relative to the robot configuration file. For example, all this information can be supplied by placing the following lines anywhere in the hand configuration file (in this example, `Barrett.cfg`):

```
...
EigenGrasps
eiegn/barrett_eigen.egr
...
```

## 10.2 Using eigengrasps

With the desired hand highlighted in the hand drop-down list, use the Grasp → EigenGrasp Interface menu. Two windows will show up: the eigengrasp amplitude sliders window and the eigengrasp options window. If no eigengrasp information has been loaded from a file, the system will display the trivial eigengrasp set, where each eigengrasp corresponds to exactly one DOF and the eigengrasp subspace is identical to the DOF space.

- at launch, the hand is placed in a neutral position (all eigengrasp amplitudes are 0). By moving each slider you control the amplitude of the respective eigengrasp. You can also move individual DOF's like you normally would in GraspIt. The result depends on the mode of operation, defined by the `Rigid` checkbox as described below.

- rigid mode: only movement in Eigengrasp space is allowed. If an individual DOF is changed, had posture is projected back into eigengrasp space.

- non-rigid mode: free movement is allowed. When an individual DOF is changed, the eigengrasp sliders still show the eigengrasp amplitudes of the projection in eigengrasp space. However, hand posture is allowed to leave eigengrasp space.

- collision detection is **not performed** while moving using the eigengrasp interface. This is temporary, we will probably add it in soon.

- when moving the eigengrasp sliders, motion is stopped as soon as any DOF reaches its joint limit. Alternatively, it could be possible to just block the stopped DOF and allow others to move, but this would take the hand out of eigengrasp space. See code (`eigengrasp.h`) for details.

# 11 Grasp Planning II - Eigengrasp planning

This chapter described the grasp planners that run in Eigengrasp space. Unfortunately, they do not share the same framework as the Primitive-based planner described earlier, although from a software engineering viewpoint they probably should, as they are still rooted in the same try-many-grasps concept.

Most of the ideas behind these planners are explained in a lot of detail in the Publications. We strongly recommend reading the Eigengrasp related papers that can be found there before proceeding through this chapter, they will make most of the concepts shown here much clearer.

## 11.1 Grasp planning in Eigengrasp space

In this family of planners, the task of grasp planning is seen as a search over multiple variables, or as an optimization problem for a high-dimensional function. The variables define the grasp and the optimized function is the quality of the grasp. In general, in order to define a grasp we need two sets of variables: the intrinsic variables (to define hand DOF's) and the extrinsic variables, to define the position of the hand relative to the target object.

For dexterous hands, if we consider one variable for each DOF, the space is too high-dimensional to search efficiently. That is why most of these algorithms only work well in eigengrasp space, where the intrinsic variables are the eigengrasp amplitudes. However, there is nothing in the structure of these planners that prevents them from running in DOF space. In fact, if you load a hand model that does not have Eigengrasps defined, GraspIt! will automatically define the "identity" eigengrasp set, with one eigengrasp corresponding to each DOF. You can then run all Eigengrasp-based planners exactly as you would if "real" Eigengrasps were defined.

An Eigengrasp-based planner is composed of three things:

- a set of variables to search over
- a quality function to optimize over these variables (usually related in some way to grasp quality)
- an optimization method (usually Simulated Annealing)

To start, load a hand and an object model, then use the Grasp → EigenGrasp Planner menu. The Eigengrasp Planning dialog window appears. The left side of the window is dedicated to the variables that are searched. The right hand side is dedicated to the planning process itself, allowing you to choose from multiple types of planners, see details about the current planning execution, etc.

## 11.2 Variables

All variables that are being searched are shown in the left panel of the dialog. Variables serve two main purposes: they define hand position and hand posture.

- **hand posture** is defined using Eigengrasp amplitudes. As stated above, if the hand you are currently using does not have other Eigengrasps defined, the planner will just use the "identity" eigengrasp set. All Eigengrasp variables are shown in the dialog with the prefix `EG`.
- **hand position** can be parameterized in many different ways. The drop down list `Type` offers a number of different parameterization options. When the selection in this

drop-down list changes, you will see the list of variables change as well. For complete exhaustive searches, the default option is `Axis-angle`. In this case hand position is parameterized using 6 variables: 3 for translation and 3 for rotation. The other ways of specifying hand position are more specialized; see code and Publications for more details.

In the GraspIt! code, a state comprising all the variables in search is contained in the `SearchState` class. Please consult the documentation of this class for many details about these variables, different ways to encode hand position, etc.

Next to each variable, there is an `On` checkbox, which can be used to enable / disable search over a particular variable. If a variable is disabled (its box is unchecked), it is no longer part of the search, and will maintain its current value in all generated solutions. For each variable, there are also other options marked `Input`, `Target` and `Confidence`. These options are used for a very specialized planning operation, using external input. This type of planning is only described in one of the papers in the Publications section, and not in this manual. For more information, please contact us. In general, just ignore these options and leave them in the default configuration.

## 11.3 Search energy

The search energy is a way of assessing how good a state encountered during a search really is. It can also be considered a function of the search variables that needs to be optimized. The most straightforward search energy is the one that we have already seen used by the Primitive-based Grasp Planner: use a grasp Quality Measure. However, a traditional QM is only defined when the hand is actually in contact with the object. When planning in Eigengrasp space, many hand states will be very close, but not in perfect contact with the object. One possibility would be to use the AutoGrasp function and close the hand, and compute a QM for the resulting grasp. However, that is a slow process, and we would like to be able to evaluate more hand postures quickly. As a result, we have tried to define a number of implementations of the "Search Energy" concept that do not require hand-object contacts.

The default energy function (pre-selected in the `Energy` formulation drop-down list) is `Hand Contacts`. This formulation attempts to bring all links of the hand as close as possible to the target object. The computation of this energy function can be greatly sped-up by pre-specifying "desired" contact locations on each link (see Publications for details). This is the role of the checkbox `Preset contacts`. Such pre-defined contacts can be loaded from a file. For the Human and Barrett hands, pre-specified contact files are supplied with the distribution. If no contact file has already been loaded, when the `Preset contacts` checkbox is checked the system will ask the user to select a file to be loaded. Note that it is also possible to specify a pre-defined contacts file in a robot configuration file, so that it is always loaded together with the Robot. From a code standpoint, a pre-defined contact location is loaded into an instance of the `VirtualContact` class; see that class and its documentation for details.

Other formulations for the search energy are also available, but many of them also incorporate the `Hand Contacts` formulation in one way or another. For the moment, the code implementing these formulations (the `SearchEnergy` class) is some of the ugliest in the GraspIt! codebase. We hope to change it soon, at which point we will also provide more documentation. For now, the most tested and trusted energy formulation is Hand Contacts described above.

## 11.4 Optimization method and planner types

The core optimization methods, that is at the base of all Eigengrasp planners, is Simulated Annealing. The most basic type of planning is to run a Simulated Annealing search using the Hand Contacts energy function, over all Eiegengrasp variables plus hand position parameterized as axis-angle. This is shown in the practical example below. From this, we have tried quite a number of variations on this theme.

One in particular might prove useful, so it is also exemplified below: the MultiThreaded Planner. One of the shortcomings of the Hand Contacts energy function is that is does not guarantee form-closure: it will find states where the hand conforms to the shape of the object, but not necessarily form-closed. To fix this, the MultiThreaded planner runs an optimization exactly as before; however, whenever it finds a state that has a good value for the Hand Contacts energy function, it fires off a child thread which will search that area in more detail, by applying the `AutoGrasp` function and computing the exact Quality Measure that results. The MultiThreaded planner is your best bet if you need to find form-closed grasps of an arbitrary object with one of the dexterous hands in GraspIt!.

From a code standpoint, the `EGPlanner` pure abstract class is the base for all of these planners: it contains the functionality for looping, computing a given hand energy formulation, saving the best solutions etc. The one thing that it does not know how to do is run an optimization algorithm. Its direct offspring, the `SimAnnPlanner` combines that framework with a Simulated Annealing optimization. To get started, check out the `EGPlanner` class for most details, and the `SimAnnPlanner` class for an example implementation of it. All other planners inherit from `EGPlanner`. The setup for this code is OK, but could still take quite a bit of improvement, which we hope to get to at some point.

## 11.5 Practical example: the Simulated Annealing planner

- load the `HumanHand20DOF`
- load the `flask.iv`
- fire up the EigenGrasp planner dialog (Grasp → EigenGrasp Planner menu)
- select `Axis-angle` as `Space Search Type`
- select `Hand Contacts` as `Energy formulation`
- check the `Preset contacts` box. If prompted to select a contact file, select `all_18_contacts.vgr` ( should be found in `$GRASPIT/models/robots/humanHand/virtual` )
- select `Sim. Ann.` as `Planner Type`

- set 70000 as `Max Steps`
- click `Init`
- click `>`
- allow the planner to run until finished.
- use the `Show Results` buttons ( `< Best >` ) to see the results.

### 11.6 Practical example: the Multithreaded planner

Not yet written... Coming soon...

## 12 Joint Coupling and Underactuated Hands

In the chapters so far we have used the terms "joint" and "DOF" as if they were the same thing, and one DOF always correspond to one and only one joint. This is not always true in real hand models, and it is not always true in GraspIt! either. GraspIt! allows you to connect multiple joints to a single DOF (joint coupling) and allows support for multiple coupling methods.

The best way to think of a DOF in GraspIt! is as a motor. It is responsible for moving the joints of a robot, and it is the only external interface to the robot available for doing that. In the GraspIt! GUI, joint draggers are not actually "joint" draggers but rather "DOF" dragger: only one dragger is presented to the user for each DOF. In the case of the Barrett hand for example, you will notice that there is only one dragger per finger, even though each finger has two joints. Moving the one dragger however affects both joints, as they are coupled to the same DOF.

In a fully actuated robot, there is one motor for each joint. In many cases however, we have joint coupling, that is multiple joints connected to a single DOF. The key aspect here is how exactly this coupling is implemented from a hardware standpoint, and whether we can replicate that in software.

A GraspIt! DOF is responsible for telling us how all of the joints that are attached to it respond to changes in the DOF. This intelligence is built into the DOF class and its implementations. Depending on how the joint coupling is achieved, DOF behave differently, which is why there are multiple implementations. The most important question is: when one of the joints of a DOF can not move (presumably due to some contact on its links), what happens to the other joints connected to the same DOF?

In dynamics mode, a DOF is also responsible for enforcing the same mode of operation. This is done via "DOF constraints" which must be built into the LCP solved by the dynamics engine. For instance, a rigidly coupled DOF must ensure that all of its joints always move at the same rate. Finally, the DOF is also responsible for applying forces to its joints (which then pass them on to the links) in dynamics mode. This is the only way of moving a Robot in dynamics. Force generation capabilities also depend on the implementation of the DOF, see the classes described below for details.

## 12.1 Rigid coupling

The simplest type of coupling is rigid, implemented in practice through gear transmission of steel cables. This means that whenever one joint from a DOF stops moving, all the other joints from the same DOF stop as well. In code, this is implemented in the RigidDOF class. Most hands in GraspIt! use this kind of underactuation (Robonaut, DLR, the 16 DOF version of the HumanHand). See the `RigidDOF` class for details.

## 12.2 Breakaway transmission

This kind of coupling allows a distal joint to "break away" and continue to close if a proximal link has been stopped. The break away point is marked, and the proximal joint is only re-engaged if the DOF goes back to the break away point. This is the type of transmission built into the Barrett hand. In code, this is the `BreakAwayDOF`, see this class for details.

## 12.3 Under development: compliant joints

This is the transmission achieved in practice through tendon networks and compliant (spring-like) joints. It is more difficult to simulate that the BreakawayDOF, as stopped joints are always connected to the DOF. In the code, this is the `CompliantDOF`. However, this is still an active area of research for us, and this kind of DOF has proven difficult to simulate in static mode.

For the compliant DOF we have also built a mechanism for hand quasistatic analysis. This mechanism exists for any kind of DOF, but is only relevant for the compliant one: due to its nature, the Compliant DOF will apply some unbalanced forces through existing contacts during the hand closing process. The Compliant DOF is thus responsible for computing these forces, which are then passed on to the quasi-static computation. This computation is then performed by the `Grasp` class; also see the Grasp quality metrics chapter for details. A complete description of this engine and its applications can be found in the Publications chapter.

# 13 Matlab interface

**Warning**: The Matlab interface has not been been maintained or tested recently. We hope to bring it up to date at some point, but for now you might have to fix it yourself to get it working again.

When GraspIt! begins, it starts up a TCP server that accepts connections. A text protocol, which will be described in future documentation, allows external programs to issue commands and collect data from the simulation. These programs could run on the same machine that GraspIt! is running on or on a separate machine. The functions described in this document allow MATLAB to communicate with GraspIt. The functions are implemented as C language MEX files. Each one opens a connection to the GraspIt! server (which for now must be running on the same machine), issues a command to either change or query the world state, and closes the connection.

To use these functions, be sure to set your MATLAB working directory to the "matlab" folder within the GraspIt root installation folder. Type "help matlab" to get a summary of the available functions, and type "help" and the name of a function to find more detailed information.

## 13.1 Commands

- `computeNewVelocities` - solves for the new velocity of each dynamic body
- `getAverageContacts` - returns the average contact force and position for a body or bodies.
- `getBodyName` - returns the name of a body or bodies.
- `getContacts` - returns all contact forces and positions for a body or bodies.
- `getDOFVals` - returns the current positions of the degrees of freedom of a robot or robots.
- `getRobotName` - returns the name of a robot or robots.
- `moveDOFs` - kinematically moves the DOFs of one robot to desired positions.
- `moveDynamicBodies` - moves every dynamic body by one timestep or until contact occurs.
- `render` - instructs GraspIt! to render the scene.
- `setDOFForces` - sets the forces acting on the degrees of freedom of a robot or robots.
- **Body related commands**: `getAverageContacts, getBodyName, getContacts` - These commands can be used on a specific set of bodies if they are given a body index vector, or if none is supplied they operate on every body currently defined in the GraspIt! world. The `getBodyName` function can be used without arguments to see which body corresponds to which index. The position of the body name in the returned array is the index of the body in the world.
- **Robot related commands**: `getDOFVals, getRobotName, setDOFForces` - These commands can be used on a specific set of robots if they are given a robot index vector, or if none is supplied they operate on every robot currently defined in the GraspIt! world. The `getRobotName` function can be used without arguments to see which robot corresponds to which index. The position of the robot name in the returned array is the index of the robot in the world.

  The moveDOFs function requires a single robot index, because GraspIt! is not setup to kinematically move multiple robots at once.

- **Dynamics related commands**: `moveDynamicBodies, computeNewVelocities` - These two functions are used to advance the dynamic simulation by one time step. The user supplies moveDynamicBodies with a maximum time step, and the fuction moves each body along its current trajectory for that amount of time. However, if a contact occurs during that period, the function moves the bodies up to that point, adds the new contact, and returns the amount of time that passed until the event occurred. A similar process occurrs a joint limit is reached. The time value returned should be passed to the `computeNewVelocities` command when it is called. This command solves for the impulses acting on each body and the resulting velocity of each body given the current constraints.

## 13.2 One time step

A script that performs a dynamic simulation will contain a loop, where each iteration of the loop advances the simulation by one time step. The following is a possible example script:

```
%the maximum time step is commonly 2.5 milliseconds
maxdt = 0.0025;

while 1
  % advance the simulation
  h = moveDynamicBodies(maxdt);
  computeNewVelocities(h);

  % compute the control forces
  %where 6 9 12 are the indices of the fingertip bodies
  [CF,CP] = getAverageContacts([6; 9; 12]);

  DP = getDOFVals;
  DF = computeControlForces(DP,CF,CP);
  %This would be a user supplied function that computes the new
  %DOF forces based on the current DOF positions, the contact forces,
  %and positions.

  setDOFForces(DF);
  %the computed forces are applied to the bodies and will affect
  %the next velocity computation.
end
```

# 14 Hardware connections

We have found two main reasons for connecting GraspIt! to hardware devices. The first one if for GraspIt! to provide output: allow a real robot to be controlled from within the simulator. Usually, this is done by having a virtual model of the robot inside GraspIt! that uses an algorithm running in the simulated environment. The real robot must then match the pose of its virtual replica. The second application is to provide input to GraspIt!, in the form of object geometry from a scanner, object location from a tracker, robot pose, etc.

For the moment we do not have a unified architecture for connecting GraspIt! to real world devices. This means that when you need such a connection, you will probably need to write some interface code yourself. In the future, we might write a general interface for the virtual robot to real robot paradigm.

In our work, we have connected GraspIt! to the following external devices:

- a real Barrett hand
- a Flock of Birds tracker that can be used to move objects or robots in the simulation world
- a Cyberglove which can be used to provide hand pose input.

All the code for these connections is included with the current distribution. However, it has two shortcomings: first it is Windows-only. The only reason for that is serial port communication which he have not yet made cross-platform. The second is that the code needs a good overhaul to improve its design and robustness.

All the code that is specific to the hardware is offered as a separate Visual Studio project called `hardware`. It can be found in `$GRASPIT/hardware`. You will need to compile this project separately into a static library. Then, inside the main GraspIt! project file (`graspit.pro`), indicate that you want GraspIt! linked against it and its features accessible. This project contains a simple Serial Port interface that is used by all hardware interfaces, and interfaces for each of the three pieces of hardware mentioned above.

The second part of the interface is code that lives within GraspIt! itself. All of this code is guarded by pre-processor definitions so that it is only compiled if the `hardware` project had been built and linked against. We are really hoping to improve this design at some point. Most of this functionality is accessible from the GraspIt! GUI via the Sensors menu.

## 14.1 Barrett Hand

A virtual Barrett hand can be linked to a real Barrett hand. Then, the pose of the virtual hand can be replicated by the real hand, or vice versa. The GraspIt! GUI also provide a crude dialog window for doing this. The Barrett class is a good starting point to check out this implementation.

## 14.2 Flock of Birds

A Flock of Birds tracker can be used to set the position of any element (body or robot) in the GraspIt! simulation world. The following steps must be followed:

- at load time, the Robot configuration file, or the Body file, must specify that the Robot or Body is to be controlled by the Flock of Birds. In order to do this, you must also specify where on the Robot (or Body) the Flock of Birds sensor is to be mounted. Remember that the origin of a body's geometry in GraspIt! is often arbitrary, and the location of the sensor makes a huge difference. For examples, look at the `FlockSensor.iv` body file (in `$GRASPIT/models/objects`) and the `HumanHand20DOF` robot configuration file.

- using the GraspIt! GUI, you must turn on Flock of Birds tracking. When that happens, the GraspIt! world (in the `World` class) will periodically monitor the Flock of Birds and ask for an update of its position. Then, it will update the position of all the elements in the simulated world that are controlled by the flock. See the relevant functions in the `World` class for details.

## 14.3 Cyberglove

A Cyberglove can be used to set the pose of a hand. However, hand models in GraspIt! do not necessarily have a perfect correspondence between their DOF's and glove sensors. Therefore, some translation is necessary, telling the hand which DOF's correspond to which glove sensors. This functionality is built into the `GloveInterface` class. Furthermore, some form of calibration is also needed to map raw sensor values to DOF values. This turned out to be a very delicate thing to achieve in practice. The `GloveInterface` can also perform calibration for you, then save the calibration to a file. Similar steps to the Flock of Birds must then be taken:

- at load time, the Robot configuration file must indicate the name of the Cyberglove calibration file that is to be used. See the `HumanHand20DOF` config file for an example.
- using the GraspIt! GUI, you must turn on Cyberglove tracking. When that happens, the GraspIt! world (in the `World` class) will periodically monitor the Cyberglove and ask for an update of its sensor readings. Then, it will update the pose of all the robots in the simulated world that are controlled by the glove. See the relevant functions in the `World` class for details.

A calibration file is provided with the `HumanHand20DOF` model. We have done our best to calibrate it, but it is a difficult task, especially for the thumb joints. We have implemented a version of the algorithm presented by Weston B. Griffin, Ryan P. Findley, Michael L. Turner and Mark R. Cutkosky, *Calibration and Mapping of a Human Hand for Dexterous Telemanipulation*, Haptics Symposium 2000. However, the calibration code needs a major overhaul and the calibration itself probably could be improved.

# 15 Publications and References



All of the publication below are available at http://grasping.cs.columbia.edu.

For an introduction to GraspIt!, the most complete overview of its core features is in:

- Andrew Miller and Peter K. Allen. “Graspit!: A Versatile Simulator for Robotic Grasping”. IEEE Robotics and Automation Magazine, V. 11, No.4, Dec. 2004, pp. 110-122.

We recommend starting with that paper for the best introduction to the system. Most of the papers below address individual features of the simulator, you can read those that are relevant to the particular project you are working on. The list of publication is presented in chronological order, from oldest to newest. For each publication, we also provide a short description of the parts of GraspIt! that it is most relevant for.

- Andrew T. Miller and Peter K. Allen. ”Examples of 3D Grasp Quality Computations”. In Proceedings IEEE International Conference on Robotics and Automation, Detroit, MI, pp. 1240-1246, May 1999.

  Introductory theory on the grasp quality metrics used by GraspIt!. Discussed topics such as the Grasp Wrench Space, L1 and LInf norms, epsilon and volume quality metrics, etc.

- Andrew T. Miller. ”GraspIt!: A Versatile Simulator for Robotic Grasping”, Ph.D. Thesis, Department of Computer Science, Columbia University, June 2001.

  Extremely detailed presentation of the GraspIt! core. The presentation is mostly from a theoretical and research standpoint, but also covers a number of practical implementation issues.

- Danica Kragic, Andrew T. Miller, Peter K. Allen. ”Real-time tracking meets online grasp planning”. In Proceedings IEEE International Conference on Robotics and Automation, Seoul, Republic of Korea, pp. 2460-2465, May 2001.

  Application of GraspIt! to execute a grasping task with a real robot. A real-life object is tracked using a camera, its position is replicated in GraspIt! where a grasp is planned using a virtual Barrett hand. The grasp is then executed using a real Barrett hand.

- Andrew T. Miller, Steffen Knoop, Peter K. Allen, Henrik I. Christensen. ”Automatic Grasp Planning Using Shape Primitives,” In Proceedings of the IEEE International Conference on Robotics and Automation, pp. 1824-1829, September 2003.

  Detailed discussion of the Primitive-based grasp planner.

- Andrew T. Miller, Henrik I. Christensen. ”Implementation of Multi-rigid-body Dynamics within a Robotic Grasping Simulator” In Proceedings of the IEEE International Conference on Robotics and Automation, pp. 2262 - 2268, September 2003.

  Presents the theoretical framework between the dynamics engine in GraspIt!. Covers topics such as time step integration, formulation of contact and joint constraints as Linear Complementarity constraints, etc. Shows how the full Linear Complementarity problem is assembled and solved at each time step of the dynamic engine. A must-read for understanding GraspIt! dynamics.

- Rafael Pelossof, Andrew Miller, Peter Allen and Tony Jebara. "An SVM Learning Approach to Robotic Grasping". In IEEE Int. Conf. on Robotics and Automation, New Orleans, April 29, 2004, pp. 3512-3518.

  Proposed the use of GraspIt! to generate large amounts labeled grasping data that can be used to apply machine learning algorithms. This code is not included in the current GraspIt! distribution.

- Matei Ciocarlie, Claire Lackner and Peter Allen. "Soft finger model with adaptive contact geometry for grasping and manipulation tasks". In IEEE Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, Tsukuba, JP, March 19-21, 2007.

  Discusses the Soft Finger contact as implemented in GraspIt!, covering the analytical surface approximation, soft finger grasp wrench space and formulation as linear complementarity constraints.

- Corey Goldfeder, Peter K. Allen, Claire Lackner, Raphael Pelossof. "Grasp Planning via Decomposition Trees". In IEEE Int. Conference on Robotics and Automation, April 13, 2007, Rome.

  Proposes an automatic method of decomposing an object into primitives (superquadrics) to fully automate the task of primitive-based grasp planning. This code is not included in the current GraspIt! distribution.

- Matei Ciocarlie, Corey Goldfeder and Peter Allen. "Dimensionality reduction for hand-independent dexterous robotic grasping". In IEEE / RSJ Conference on Intelligent Robots and Systems (IROS) 2007, San Diego, Oct. 29 - Nov. 2

  Introduces the eigengrasp concept and grasp planning in eigengrasp space as an optimization problem solved through Simulated Annealing. This is the recommended starting point if you are interested in eigengrasps.

- Matei T. Ciocarlie and Peter K. Allen. "On-Line Interactive Dexterous Grasping". In Eurohaptics 2008, Madrid, June 10-13, 2008.

  Present an application of eigengrasp planning for on-line interaction with a human user. This is the theory behind the OnLinePlanner class included in the distribution.

- Corey Goldfeder, Matei Ciocarlie, Hao Dang and Peter K. Allen. "The Columbia Grasp Database". In IEEE Int. Conf. on Robotics and Automation 2009, Kobe.

  Shows how GraspIt! can be used to generate a huge database of labeled grasp data, and how this database can be used for data-driven grasp planning algorithms. The database is publicly available. We are currently working on releasing all the interface code that you need for using both GraspIt! and the Columbia Grasp Database together. We hope that this feature will be available in the summer of 2009 at latest.