GRRC Technical Report 2010-02

Dynamic Simulation and Collision Modeling of the Packbot Manipulator

Author: **John Hall**
Supervisor: **Joshua Langsfeld**
Advisors: **Prof. Dawn Tilbury**
                 Prof. Ella Atkins

Date of completion: August 11, 2010
UNCLASSIFIED: Dist. A. Approved for public release

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **11 AUG 2010**	2. REPORT TYPE **Journal Article**	3. DATES COVERED **11-08-2010 to 11-08-2010**

4. TITLE AND SUBTITLE **DYNAMIC SIMULATION AND COLLISION MODELING OF THE PACKBOT MANIPULATOR**	5a. CONTRACT NUMBER **w56h2v-04-2-0001**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **John Hall**	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Universtiy of Michigan | Automotive Research Center,Ground Robotics Research Center,2350 Hayward Street,Ann Arbor ,MI,48109-2125**	8. PERFORMING ORGANIZATION REPORT NUMBER **; #21387**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **U.S. Army TARDEC, 6501 E.11 Mile Rd, Warren, MI, 48397-5000**	10. SPONSOR/MONITOR'S ACRONYM(S) **TARDEC**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) **#21387**

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Packbot is a ground robot commonly used by the US military for explosive ordnance disposal. It is prone to failure in the field for various reasons [1]. When a single joint of the Packbot EOD manipulator fails, the entire arm is rendered useless despite the fact that it has redundant degrees of freedom. We are designing reconfigurable control systems and strategies to regain manipulator function in the face of joint failure, but we lack any way to test these strategies. Either a copy of the arm or a software model is needed, and this report describes the software route.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**22**	

Table of Contents

1 Introduction

The Packbot is a ground robot commonly used by the US military for explosive ordnance disposal. It is prone to failure in the field for various reasons [1]. When a single joint of the Packbot EOD manipulator fails, the entire arm is rendered useless despite the fact that it has redundant degrees of freedom. We are designing reconfigurable control systems and strategies to regain manipulator function in the face of joint failure, but we lack any way to test these strategies. Either a copy of the arm or a software model is needed, and this report describes the software route.

2 The Goal

The purpose of this research project is to model of the physical behavior of the Packbot EOD arm in order to enable the design of reconfigurable control systems. The behavior described in the robot dynamic equations is important, as are the contact forces between the links of the robot and its static environment. A complete model will include kinematics, dynamics, and collision physics. Such a model will allow us to study failures in the Packbot arm without a physical example of the arm. This report describes efforts to study and model the Packbot arm and the development of virtual tools to aid this and future studies of the Packbot.

3 Modeling

Initial work on control reconfiguration of the Packbot manipulator can be divided into three tasks: describing the geometry and kinematics of the real Packbot arm with software tools, creating dynamic models to predict the behavior of the arm, and using these dynamic models to simulate failures and new control schemes.
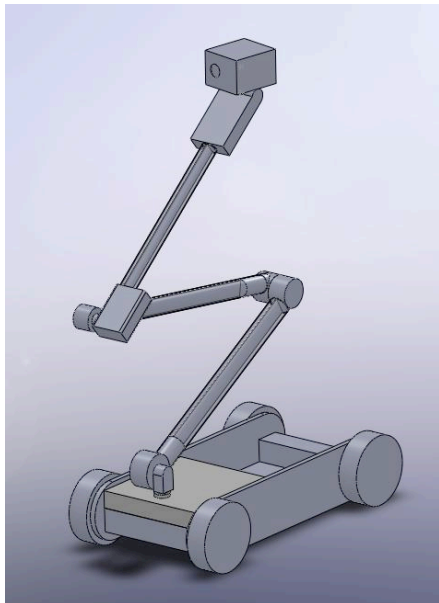
Figure 1: SolidWorks model

3.1 Geometry and Kinematics

SolidWorks was used to create a visual and geometric model of the Packbot, shown here. This model is based on length measurements taken by hand. Because the measurements were taken by hand, the accuracy uncertainty is noticeable. Detailed notes on the model's fidelity may be found in Appendix A. The focus in creating this model was to define the manipulator enough to accurately determine its Denavit-Hartenberg (DH) parameters. These parameters can be used to fully describe the kinematic behavior of the manipulator.

Each link of the arm is also designed to have inertial properties similar to the actual link. Each of the first two major links is represented by a hollow tube encasing a computer chip and a

motor. The ends are capped with solid parts meant to represent the joint casing and gearings. The third major link has, like the first two, a tube containing a chip. The gripper wrist and the camera wrist are described as solid objects. The motors in this link are assumed to be contained in the wrist parts. Each CAD part is considered to be uniform in density, but since the mass and location of each object in the link is true to reality, the overall inertia and center of mass for each link should be relatively accurate.

Since SolidWorks is ill-suited to kinematic analysis, a MATLAB model was created using the geometry of the first model as a reference. Specifically, we used the MATLAB Robotics Toolbox [2] created by Peter Corke to make this model. In addition to describing the kinematics of serial-link manipulators, the toolbox also includes features for dynamic analysis.

3.2 Dynamic Modeling

While kinematics is useful for studying the workspace of the manipulator and for mathematically describing its movement, a dynamic model is needed to simulate mechanical behavior, like forces and inertia. We used SolidWorks to calculate the inertia and center of mass of each link. Information about the mass data may be found in Appendix A. iRobot provided the gear ratios for each of the arm joints.

All this data applies to the behavior of the normal arm, but we are also interested in the behavior of a failed arm. Two common types of electromechanical failure in the manipulator are a frozen joint and a loose joint (that cannot exert any torque). A loose joint is modeled easily by restricting the torque exerted by that joint to zero. We modeled a frozen joint by redefining the arm as a five-link manipulator, with the two links connected by the frozen joint considered to be one link. For the frozen joint case, the new DH parameters are geometrically calculated, but the inertial properties of the combined link are interpolated across all possible joint angles. We used this more complicated method – instead of simply restricting the proper joint variable – in order to simplify the design of the reconfigurable control system. For kinematic analysis this extra calculation is not needed.

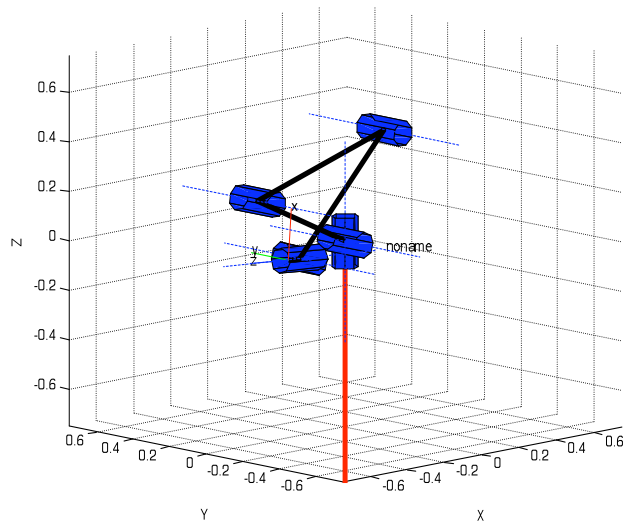Figure 2: MATLAB dynamic model with colliding joints

3.3 Collision Detection

The normal interactions of the links (at the joints) are described by the dynamic and kinematic models, but it is also possible for the movement of the links to be restricted by interference with each other, the Packbot chassis, or the environment. To prevent physical impossibilities like intersecting solid bodies and

4

to model the reaction forces due to any collision between the solid bodies in the model, collision detection software is needed.

Our collision detection algorithm approximates each solid body as a combination of convex polyhedra, as shown in Figure 3 below. This approach not only follows the real geometry of the Packbot very closely, but also allows the algorithm to use simpler and more robust geometric computations. The algorithm uses the Gilbert-Johnson-Keerthi (GJK) algorithm as a basic primitive test for collision between two polyhedra, and this method would not be valid for concave polyhedra. The GJK algorithm is described in Christer Ericson's book [3], as well as in his presentation to SIGGRAPH 2004 [5]. To prune out unnecessary primitive tests and streamline the collision detection code as much as possible, we also implemented a bounding volume hierarchy. This works by enclosing certain combinations of polyhedra (all those in a given link, in this case) in a virtual box. By testing all these boxes against each other before testing the more complicated geometries, the number of primitive tests required decreases dramatically.
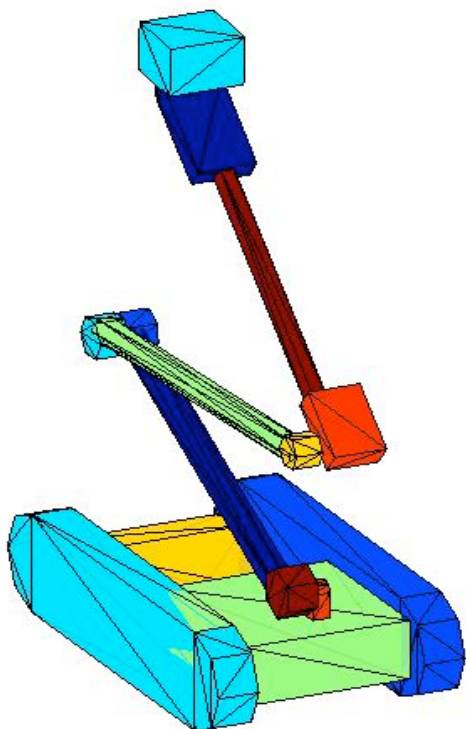
Most of this code is written in MATLAB so that it meshes well with the other components of the models and is easy to edit. MATLAB is an interpreted language, however, and as such can be fairly slow. Speed is a major problem because the geometric tests include many if-statements and loops, both of which run slowly in MATLAB. In order to speed things up, as many loops and conditional statements as possible were coded in C++ and compiled as MATLAB executables (MEX files). These are called from the MATLAB command line, but run much faster than an equivalent m-file. More details concerning the collision detection software may be found in Appendix B.

3.4 Collision Behavior and Simulation

But detecting the collisions is only solves half the problem; a useful simulation will not only detect collisions but react to them. Our dynamic simulator (collisionDynamics.m) implements ode45 with event detection. An event function runs at every solver step and

Figure 3: Packbot approximated by polyhedra

halts execution if any of the polyhedra described in the previous section intersect. Based on which bodies are colliding, the simulation resets certain joint velocities to reflect the impact and limits the acceleration of those joints as long as the colliding bodies are in contact. The joints that represent degrees of freedom between two colliding polyhedra will be called the 'relevant joints' in this report.

The main drawback to this joint-based algorithm is that if there is more than one degree of freedom between the colliding bodies the algorithm cannot tell which combination of those degrees of freedom caused the collision. Mathematically speaking, these positions of the relevant joints represent a space of

5

joint positions, and our current algorithm cannot calculate which subset of that space to avoid. Therefore it constrains every joint in that space in one direction based on joint velocities at the time of the collision. This works in most cases, but a more rigorous algorithm would limit movement in the joint space only along the basis vector normal to the surface (in the joint space) defined by the collision.

In rare cases, using joint velocities to decide the direction in which to limit acceleration also gives an erroneous result. For instance, consider a collision with two relevant joints. The net movement from these two degrees of freedom is obviously towards collision, but it is possible that one joint was not contributing to this movement; the movement of the second joint was simply overwhelming the effect of the first. Using joint velocities to determine a direction in which to limit acceleration would result in constraining the first joint in the wrong direction.

Another method we tried before using the above solution was to calculate joint torques that affect the arm in the same way Cartesian impact forces would. This method does not use the joint velocities at the time of collision, but it requires a minimum norm vector between the two colliding polyhedra. This method circumvents both major difficulties associated a purely joint space method while introducing three more problems, each of which is arguably larger.

First, the using torques to create collision behavior means using large, discontinuous torque inputs to mimic impacts, and the solver will drastically decrease its step size to deal with the new stiffness in the problem. Detailed explanations for this behavior may be found in the first two sections of Appendix B. Second, the torques would be calculated from an impact force that should depend on many factors, including gravity, coriolis and centripetal forces, inertia and any disturbances or control inputs. These calculations are so complex that understanding and checking one force at one step by hand could take hours, and any mistake could cause the objects to pass through each other or rebound with excessive velocity.

The third major problem with involves the direction of the force rather than its magnitude. The direction should be found from a normal vector at a point of tangency, where the polyhedra intersect but do not overlap. It is impossible to find that extremely specific time and state with no numerical error. Instead, the algorithm would use a vector of minimum norm in a non-intersecting state. The polyhedra that make up the arm model are geometric approximations that could lead to large, chaotic discrepancies in this vector between similar time steps, as shown below.
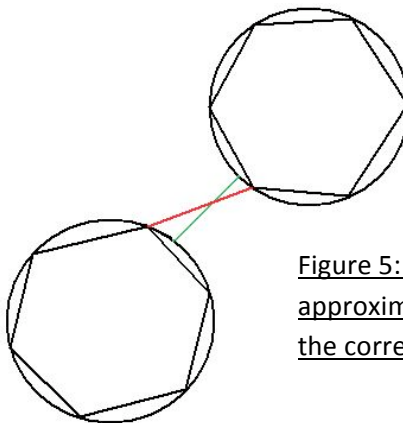
Figure 5: Error in vector of minimum norm due to approximation of smooth geometry (green line is the correct vector, red line is the erroneous one)

6

All these issues considered along with development time, using the relatively simple method described in the first paragraph provides the most realistic behavior in a reasonable time frame.

4 Results

This section describes one of the scenarios used to test our simulation. All joint torques are set to zero, simulating a total collapse of the arm. The initial position is the standard drive position, which is shown in Figure 6, and all initial joint velocities are zero. Using only the dynamic model and the simulator provided in the Robotics Toolbox, the arm falls through the chassis and continues to swing chaotically. A snapshot is shown in Figure 7.
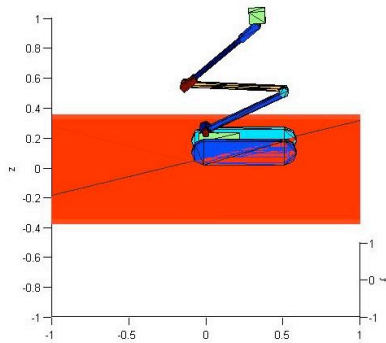
Figure 6: Drive Position
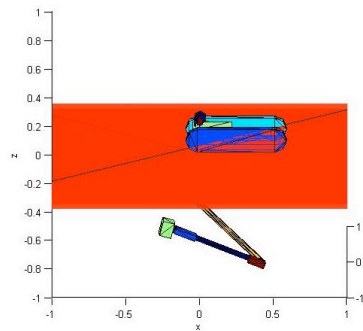
Figure 7: Result without collision
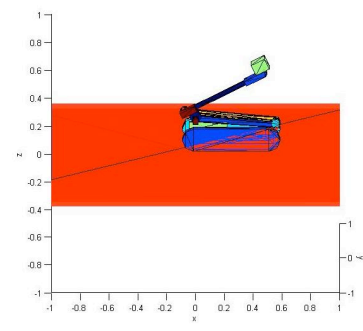
Figure 8: Static result with collision detection

In contrast, using the simulator we have written for collision dynamics yields an intuitive static result, pictured in Figure 8. The third link remains upright because it is longer than the second link and thus becomes propped up on the shoulder joint casing. Graphs of joint position and velocity over time verify our result is static; these graphs may be found below in Figures 9 and 10.
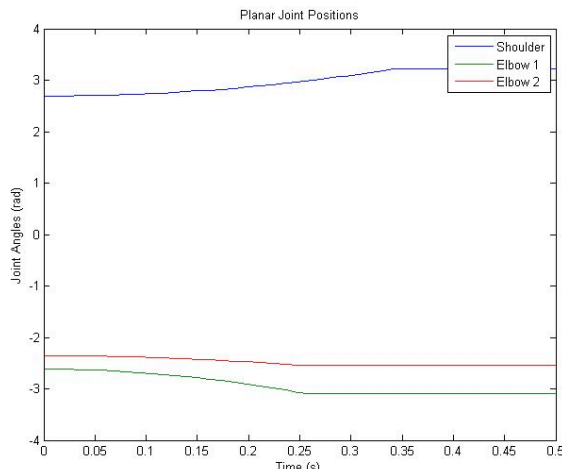
Figure 9: Planar joint positions
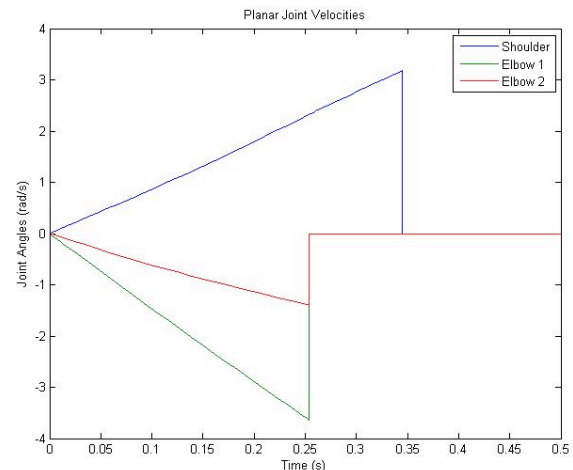
Figure 10: Planar joint velocities

5 Future Work

Before control simulations can be run, however, the models need to be refined to more accurately reflect real-world behavior. For instance, the effects of motor inertia and frictional forces (both static and viscous) should be added to the dynamic model. These parameters will be deduced from publicly available specifications of the maximum weight the manipulator can support. Torque limits would also serve to make simulations more realistic, although torque can be checked with a post-processor. Unfortunately, these torque limits are not publicly available, and so we will need to either negotiate the information from iRobot or deduce approximations from what specifications are publically available.

The next step in making the model realistic is to design a more intelligent joint constraint algorithm. One option is to keep the current algorithm but give it the ability to self-diagnosis faults. Misbehavior of the collision code can be detected by watching for secondary intersections – intersections that occur between two bodies that are already constrained. Another option is to calculate the Jacobian matrix that describes the relationship between Cartesian space and the space of relevant joints. Taking the inverse of the Jacobian and identifying the subset of joint space that is responsible for the collision would allow us to limit motion in that direction without completely constraining every relevant joint.

Whether or not we implement a more intelligent joint constraint algorithm, we will simulate dynamic transitions between failed equilibrium poses and other control challenges. The results of this research should allow us to design a reconfigurable controller that can recover function in a damaged manipulator arm. The implementation of such a controller would improve the reliability of the Packbot by mitigating the effect of failure in the field. This is the most important work that remains to be done.

Another possible use of the collision software is to detect camera occlusion. If the function to be recovered from the arm is the camera's ability to see a target, creating long, thin polytopes between the camera and its target could allow us to test whether the camera's view is blocked.

6 Summary

In order to improve the reliability of the Packbot manipulator through control reconfiguration, we have created various computer models of the manipulator. The kinematic models created in SolidWorks and MATLAB describe the geometry, position, and velocity of the arm. The kinematic model forms the basis for a dynamic model that can describe forces and collisions – the real behavior of the arm. This dynamic model will be used to formulate and test new control strategies with the goal of recovering function from a failed arm. This reconfigurable control system could improve the reliability of the Packbot by preventing minor failures from affecting the success of a mission.

But there is still more to do in refining the models should more realistic behavior prove necessary. Torque limits and the effects of motor inertia and friction should be represented in the models. Also, the algorithms that determine collision behavior could be refined to work in all cases instead of just most of

them. These changes, especially the second one, would take time but could vastly improve the model's usefulness in control design.

7 References

[1] Phuoc-Nguyen Nguyen-Huu, Joshua Titus, "Reliability and Failure in Unmanned Ground Vehicle (UGV)", *GRRC Technical Report 2009-01,* 2009.
[2] Peter I. Corke, "A Robotics Toolbox for MATLAB", *IEEE Robotics and Automation Magazine,* Vol. 3, March 1996.
[3] Christer Ericson, *Real-Time Collision Detection*. Morgan Kaufmann, 2005.
[4] John Nagle. "Re: Collision points with SOLID." *comp.graphics.algorithms* usenet newsgroup article, Message-ID <3C4089BC.80907@animats.com>. January 12, 2002.
   http://groups.google.com/groups?selm=3C4089BC.80907%40animats.com
[5] Christer Ericson (2004). "The Gilbert-Johnson-Keerthi (GJK) Algorithm", *SIGGRAPH*.
[6] "MATLAB – Documentation", The Mathworks, 2010.
   http://www.mathworks.com/access/helpdesk/help/techdoc/

8 APPENDIX A: Details and limitations of the SolidWorks model

The following dimensions are known to within 5 mm:
- full length definition of the chassis
- wheel width, radius, and location
- location of the arm with respect to the chassis
- camera dimensions
- manipulator link lengths, offsets, and diameters
- joint casing diameters
- wrist lengths and widths
- full length dimension of the arm attachment plate

These dimensions are based on estimates and memory:
- motor length
- computer chip width and thickness
- hollow tube thickness
- wrist thickness

The masses of these parts are known to within 100 g:
- motors
- joint casing/gearings
- overall arm
- camera
- camera wrist
- end effector wrist
- arm attachment plate
- overall robot

The masses of these parts are based on estimates and memory:
- computer chips
- hollow tubes
- wheels
- chassis

-

9 APPENDIX B: Notes on Collision Detection Implementation

This Appendix is meant to provide notes on the collision detection software described in Sections 3.3 and 3.4. The collision detection software is a collection of MATLAB and C++ code meant to work with the MATLAB Robotics Toolbox [2]. Currently the code is specific to the Packbot, though I may generalize it at some later date.

This version of the collision software only models the interactions between the links of the packbot and between the arm and its static environment. The simulation cannot yet handle external moving objects, though there is a possibility that such functionality will be added in the future.

9.1 Basic Concepts

Primitive Tests
The basic way to tell if objects in an environment are colliding is to test each shape in the model against every other shape for collision status and distance. Whatever test that is used is called the primitive test. In this package, the shape of the manipulator is approximated by a collection of convex polygons, and the primitive test used is the Gilbert-Johnson-Keerthi (GJK) algorithm.

GJK works by iterating through different simplicies formed from the vertices of the Minkowski difference of the two polyhedra. The Minkowski difference is a set of points with the property that a convex hull of these points will include the origin if and only if the polyhedra are colliding. At each recursive step, GJK calculates the point on the current simplex that is closest to the origin. Any points in the simplex on which this closest point does not depend are removed from the simplex. Then GJK multiplies the position vector of the closest point by negative one, and determines the supporting vertex of the Minkowski difference polytope in that direction. The supporting vertex is then added to the current simplex, and the process repeats. The recursion ends if the closest point on the simplex is the origin (the polyhedra intersect), or if the supporting vertex of the Minkowski polytope is no closer to the origin than the closest point found in that recursive step (the polyhedra don't intersect). This method converges asymptotically.

Bounding Volume Hierarchy
Even as fast as GJK converges, any collision detection code that simply tests each convex polyhedron against every other will be fairly slow. There are just too many primitive tests. One solution to this problem is to prune tests from the algorithm using a bounding volume hierarchy. The basic idea is to create simple shapes that contain certain combinations of the main set of polyhedra. With the Packbot manipulator, boxes fit well over the polyhedra defining each link. If a given pair of these bounding volumes do not collide, then none of the polyhedra contained in the first box collide with any of the polyhedra in the second box. This concept is used to prune tests from the collision detection algorithm.

Variable-step Numerical Solvers

All of the ordinary differential equation (ode) solvers that come with Matlab are variable-step. This means that instead of holding a time step constant, the solver controls error by varying step size. At each step the solver estimates both the absolute error accumulated and the error relative to the last step. If either is above a specified tolerance, the solver throws out that step and calculates a new one with a smaller time increment. This continues until the error is within tolerance. Using a variable-step solver usually guarantees that the solver has not missed any high-frequency behavior, but it does cause the step size to become very small during such behavior.

Stiffness
Stiffness is a property of differential equations that has never been clearly defined. A problem or set of equations is considered stiff if either numerical errors or high-frequency behavior cause a numerical solver to diverge or become unstable. An example would be a trigonometric function operating at 10kHz if the solver used has a fixed step greater than around 2e-6 to 5e-6 seconds. Even such a small fixed step would ignore the function's high-frequency behavior, causing the solver solution to diverge in most cases. A variable-step solver applied to the same problem would be forced to use a very small step size that may be undesirable if low-frequency behavior is also of interest.

9.2 Speed Concerns

Computational geometry is not MATLAB's strong suit, but it does provide excellent high-level programming features. In this package as many computing bottlenecks as possible have been replaced by C++ MEX files. In this way we hope to balance speed of computation with MATLAB's interface. The most important use of MEX files is in gjksubmex.m. We attempted to implement the GJK algorithm purely in C++, but the main WHILE loop (GJK is recursive) caused memory problems. So instead of pure C++ code, gjksubmex.m is a MATLAB function that calls MEX files to do most of its heavy lifting. With this MEX code gjksubmex.m can run in a fraction of a millisecond.

The collision physics of a robotic arm represent a stiff problem. The standard dynamics operate on a much longer time scale than the collisions, which can require bodies to change velocity very quickly. When the solver encounters an impact it drastically reduces the step size so as to stay within its error tolerance. Even a relatively large error tolerance will cause this reduction in step size; an error of 10% is still small compared to a change in sign (a change of at least -100%). The result is that a simulation scaled in seconds is run with microsecond step sizes, and even a streamlined code can take hours to finish.

The solution is to separate the high- and low-frequency behavior with an event detector. Matlab solvers allow a user to specify an event function that runs at each step and can halt the solver's integration if it detects high-frequency behavior. In this software package, the event function halts execution when it encounters a new collision state. The pertinent velocities are changed, and the solver is restarted with the new initial values. The solver will still decrease the step size immediately before the collision so that it can pinpoint the time and state at which the collision occurs, but resetting the integration also resets the step size. Thus event detection preserves both accuracy and a normal step size for this stiff problem.

10: APPENDIX C: Code Documentation

This appendix is meant as a reference guide for the Matlab and MEX code I have written. It explains how critical data is stored and how each function works, outlines the parent/child relationships between functions, and documents known issues or problems with the current version of the code.

Anyone using this software package should have MATLAB 7.6 or later and the MATLAB Robotics Toolbox Release 8 or later. The C++ source files are meant to be compiled as MEX files using the MATLAB command line. There is a script – described below – that is meant to automate this setup procedure. It is available on the GRRC-RA4 C-Tools site in the Resources section as a ZIP file.

There is very little exception handling in the code as it stands now. This means that the MATLAB scripts will throw random errors if given the wrong input, and the MEX files will probably throw segmentation faults. These are bad, so check this Appendix, the release notes in the ZIP file, or the comments at the top of most functions before using them.

10.1 Data Storage and Indexing

This is a description of each important data variable used as an input or output to one of the functions in Section 9.3 above. A '.mat' suffix indicates a variable that is included in the collision detection package. Each of the others is the result of a function.

polyhedra.mat – This is a data structure containing cross-referenced and indexed data on the verticies of polyhedra, specifically those used to represent physical bodies in this software package. The collision package includes a version of this variable for the Packbot EOD arm. Different instances of polyhedra can be made using the createCollisionBodies function.

.points – This field is an nx3 matrix containing the vertices of all the polyhedra in the model. The vertices are listed first by order of the reference frame to which that polyhedron is attached and second by the individual polyhedron. All points are given with respect to the local frame of reference – that is, with respect to the frame they are attached to.

.pointBody – This field is a vector length n that gives the body number of each point in the points field. Bodies are numbered within each frame, so both Frame 1 and Frame 2 have a Body 1. A number independent of frame can be obtained from the location of the body's index in the field bodyIndex.

.pointFrame – An n length vector giving the frame number each row in the points field is expressed in and attached to.

.bodyIndex – An m length vector in which each element gives the row index for the beginning of a polyhedron in the points field. All indices are given in the order they appear in that

field. The last element is the index for the first empty row (where the next polyhedron would be if there was one).

.bodyFrame – An m length vector giving the frame number for each entry of bodyIndex.

.frameIndex – A vector length p that gives the indices for the beginning of each reference frame in bodyIndex, in the same way that bodyIndex gives the beginning of each polyhedron in the points field. As before, the indices are sorted in ascending order and the last index gives the first empty element in bodyIndex.

.boundingBox – An $(8(p-1))$x3 matrix containing the vertices of all the bounding boxes calculated by boundBox.m. The format is identical to that of the points field. The fact that each box has the same number vertices, and each reference frame contains only one box, means that the extensive cross-referencing used for the original polyhedra is not necessary here.

.boundingBoxIndex – This field serves the same purpose for boundingBox as bodyIndex serves for the points field, and works in the same way. It is a vector length $8(p-1)$.

.boundingBoxFrame – An analog of bodyFrame for boundingBoxIndex that gives the frame number of each bounding box. It has the same length as boundingBoxIndex.

collisions – This data structure is a global variable internal to collisionDynamics and its child functions cdyn2 and intersection. It contains data about each collision detected by the simulation that is still affecting the behavior of the arm. These are considered the active collisions. Data is written by intersection and read mostly by intersection and cydn2.

.c – A vector of Boolean values, one for each active collision. An entry reads true if the two polyhedra involved are currently colliding.

.bodies – A matrix with a row for each active collision. Each row is four elements – two pairs. Each pair is associated with one of the colliding polyhedra, giving the first and last body indicies into the points field from polyhedra.mat.

.frames – A matrix containing the frame numbers of each polyhedron involved in an active collision. Each active collision has a row length 2, with each entry giving the frame of one of the polyhedra.

.proximity – A vector with an entry for each collision that gives the smallest distance between the two polyhedra.

.time – A vector the contains the simulation time at which each collision was recorded as active.

.joints – A vector of variable length that lists the joints that should have their accelerations restricted by cdyn2.

14

.direction – A vector the same length as the joints field that dictates the direction in which acceleration is not allowed for each restricted joint. Each entry is -1 or 1.

10.2 Calling Structure

This outline shows which functions call which others. It is only intended to give an idea of how each function relates to the others. Full descriptions are provided in the next section.

- makeCollision.m
- createCollisionBodies.m
  - boundBox.m
- initializeCollision
  - define_packbotv3.m
- collisionDynamics.m
  - cdyn2.m
    - taufun.m
  - intersection.m
    - collisionCheck.m
      - ptfkin.m
      - gjksubmex.m
        - PtMinNorm.mexw32
        - IndexandSupport.mexw32
    - findRelevantJoints.m
- wireAnimate.m
  - wireplot.m
- Rmatrix.m

10.3 Function Overviews

Below are detailed descriptions and notes on each finished function from Section 9.3 in the order they appeared in that section. Variable names are given in the Input and Output fields; sometimes these are figures created in MATLAB. Figures numbers do not refer to figures in this report; rather they give the figure window in which the output will appear.

makeCollision.m
Script – no input or output arguments

Algorithm: This script compiles the all the MEX code needed to run collisionDynamics. It also rearranges several files for convenience after you first download the collision package.

Notes: Place this script in the same directory as both the collision and robot directories, and then run it. It will ask you to choose a compiler first. You may have to try one or two of these because certain compilers cannot handle MEX code.

createCollisionBodies.m
Input: Robot object
Output: polyhedra

Algorithm: This function iterates through each frame, asking the user to identify which workspace variables hold the points to define the polyhedron for each body. The user should create these variables beforehand. As it goes, it cross-references the data and stores this indexing data in the proper fields of polyhedra. When all bodies are created, createCollisionBodies will run boundBox to create bounding volumes around the bodies of each reference frame.

Notes: I highly recommend using this function to create polyhedra data structures rather than assigning everything manually. It will save time and ensure that all indexing and cross-referencing is done correctly.

boundBox.m
Input: polyhedra
Output: polyhedra

Algorithm: This function iterates through each reference frame. At each frame, the function finds the maximum and minimum values in each Cartesian dimension. It uses these to determine the vertices of a box containing all bodies attached to that reference frame.

Notes: This function is not very streamlined, but doesn't need to be. It only runs once: at the end of createCollisionBodies.

initializeCollision.m
Script – no input or output arguments

Algorithm: Adds paths to the collision and robot directories, as well as the simulink and mex directories within robot. It will then run define_packbotv3 to create a dynamic model of the Packbot EOD arm, load polyhedra.mat, and set the joint variables q to put the EOD arm in the standard drive pose.

Notes: This script was meant solely for my convenience and I encourage users to alter it for theirs. It's stored in the collision directory until you run makeCollision, which moves it outside to the parent directory.

define_packbotv3.m
Script – no input or output arguments

Algorithm: This script creates the robot and link objects for a complete dynamic model of the
                 Packbot EOD manipulator. Most parameters in this version are approximate, but close
                 enough to give good general results. See the Robot Toolbox guide for further
                 information.

Notes: none

collisionDynamics.m
Inputs: Robot object, Initial time (t0), Final time (t1), Function handle for a user-defined torque
                 function (torqfun), Initial joint positions and velocities (q0, qd0), polyhedra, any other
                 variables the user would like passed to the torque function
Outputs: Time vector (tout), Joint position and velocity results arranged in rows by time step (qout,
                 qdout), Event times (tcout), Joint positions and velocities at event times (qcout, qdcout)

Algorithm: This function is the main simulation loop for a collision dynamics simulation.
                 Initialization includes creating the data structure collisions as a global variable and
                 creating an ode options structure that alters error tolerances and sets intersection.m as the
                 events function.

                 The simulation loop runs ode45 to solve the robot dynamic equations. The simulation
                 will terminate early if a collision event is detected. If this happens, the initial values will
                 reset to the time step before the collision, but with the velocities of all joints relevant to
                 the collision set to zero. The solver will then be called again to continue solving the robot
                 equations. This loop will continue until the simulation time has reached the final value
                 given by the user. The global variable collisions is cleared before collisionDynamics
                 terminates.

Notes: This function replaces fdyn from the Robot Toolbox as a dynamic simulation function
                 and wrapper for ode45. The torque function serves the same purpose as before, and the
                 outputs are formatted similarly. The main differences are the event detection features and
                 the addition of polyhedra as a required input. As in fdyn, extra arguments will be passed
                 through to the torque function.

                 When an event occurs and the solver terminates, the joint velocities relevant to the
                 collision do not have to be set to zero. This represents only the perfectly inelastic case.
                 Multiplying the velocities by a coefficient of elasticity will yield any sort of collisions the
                 user desires. However, the simulation will break if the sign of the velocity is not reversed
                 in the more elastic case.

                 The user is encouraged to tweak the error tolerances to ensure a reasonable run time. This
                 function reads from collisions, but does not write to it.

cdyn2.m

Inputs: Current time and joint states (t, x), Robot object, Torque function (torqfun), polyhedra,
             and undetermined extra arguments for the torque function

Outputs: Joint state derivatives for the next time step of ode45 (xd)

Algorithm: This function represents the robot dynamic equations for the solver. It takes in the current
             state, runs the torque function to decide what joint torque to apply and runs accel from
             the Robot Toolbox to calculate the state derivatives for the next time step. Before ending,
             however, it restricts the acceleration of certain joints based on information stored in the
             global variable collisions. These restrictions – in theory – should prevent further
             intersections from a previously recognized collision event. The restrictions are decided by
             findRelevantJoints.m and in their current version should work in most cases.

Notes: If the functioning of the acceleration restrictions is not acceptable, feel free to alter the
             way these restrictions are decided in findRelevantJoints.m. See the guide entry for that
             function for more details.

             This function reads from collisions, but does not write to it.

taufun.m

Inputs: Current time (t), Current joint positions and velocities (q, qd), Robot object

Outputs: Joint torques (tau)

Algorithm: This is a sample function used to set torques as a function of time and the current state of
             a robot. Any code written to control a robot belongs here. This sample simply sets all
             joint torques to zero, resulting in a collapse of the arm if run in a simulation.

Notes: User-written.

intersection.m

Inputs: Current time and joint states (t, x), Robot object, Torque function (torqfun), polyhedra,
             and other undetermined inputs intended for the torque function

Outputs: Event function value (value), Boolean command on whether to terminate the solver for a
             given event (isterminal), and the direction the Event function crossed zero (direction)

Algorithms: This function, as the event function for ode45, is run at each solver step. If the event
             function crosses zero, an event is said to have occurred. See Matlab documentation for
             more details.

             collisionCheck is run after initialization, followed by findRelevantJoints. These two
             functions together completely define the global variable collisions. If any polyhedra are
             intersecting, the sign of the event function value is changed from negative to positive and
             isterminal is set to one, halting execution of the solver.

Notes: This function writes to collisions.

collisionCheck.m
Inputs: Robot object, Current joint positions (q), polyhedra, [current time, collisions]
Outputs: Boolean value reading true if any polyhedra intersect, [or an updated copy of the global variable collisions]

Algorithm: This is essentially two functions combined. The first function simply checks for any polyhedron intersections. The second is meant to be run from intersection.m. This second version is run if it is given the arguments in brackets above, and outputs the global data structure collisions instead of just a Boolean value.

                Both versions of code run forward kinematics on the polyhedra, use the bounding volume hierarchy to filter out most primitive tests, and then run those primitive tests. The more basic version will terminate as soon as a pair of polyhedron primitives are found to be intersecting.

                The advanced version called by intersection.m has more steps. Initialization includes reading the optional variables from varargin and slightly different initial sizes for the primitive check matricies, as well as setting a proximity tolerance for deactivating collision states. The bounding hierarchy is the same, but at the primitive tests the algorithm diverges significantly.

                Old collisions states are automatically tested again for a new proximity value if nothing else. Tests are sorted by whether the collision has or has not already been detected and whether the polyhedra involved are colliding or not. In each case, the collisions data structure is altered in different ways.

                The advanced version of collisionCheck ends by deleting any collisions that have left become inactive by leaving the proximity tolerance (the polyhedra are no longer close to each other).

Notes: In the primitive test case of an old collision that has intersected again, the simulation is probably broken. findRelevantJoints should be setting joint constraints so to prevent this. The current version of the code does nothing special here, but the algorithm could be made to self-diagnose faults by altering collisions in a new way when it encounters this case.

                Although this code is very long, most of the runtime is spent on gjksubmex. The rest of the function is well vectorized. The user should feel free to alter the value set to proximityTolerance during initialization if it results in unsatisfactory behavior.

ptfkin.m
Input: Robot object, Current joint positions, polyhedra
Output: Polyhedron vertices in current position (transformedPoints), Bounding box vertices in current position (transformedBoundingBoxes), Frame origins in current position (frameOrigins)

Algorithm: The algorithm initializes by retrieving the manipulator's DH parameters from the robot object and creating the index vectors for bb. The function then iterates through each frame of reference in reverse, transforming all points from polypts and bb into the base frame to match the current position of the arm.

Notes: This function may be a slight bottleneck in the collision functions. But the algorithm, though computationally intensive, is well vectorized. The benefits of translating this function to a MEX file are unclear. There may be a bug of some sort that causes all objects to be rotated 180 degrees around the local z axis.

gjksubmex.m
Input: Vertices of two convex polyhedra (A, B)
Output: Collision status (c), Indices of points in A and B used in final simplex Q (I), Barycentric coordinates in terms of Q for the final point (Bary)

Algorithm: This function applies the Gilbert-Johnson-Keerthi (GJK) algorithm [5] to a pair of convex polyhedra. For details of the main algorithm, see that reference. PtMinNorm.mexw32 is used to find the point on Q closest to the origin, and IndexandSupport.mexw32 is used to remove old points from Q and find a new supporting vertex.

Notes: This implementation of GJK is basically a MATLAB skeleton for the MEX files it calls. Since most of the heavy lifting is done in C, this MATLAB code will run relatively quickly. It is used as the basic primitive test for the intersection of the polyhedra composing the model and any bounding volumes.

PtMinNorm.mexw32
Input: Vertices of the current simplex (Q)
Output: Cartesian (Cart) and barycentric (Bary) coordinates of the point of minimum norm

Algorithm: This code finds the closest point on a simplex to the origin by determining the order of the simplex and running the appropriate subroutine.

Notes: This function is designed for use in gjksubmex.m, but it may be useful elsewhere.

IndexandSupport.mexw32
Input: Vertices of the current simplex (Q), Vertices of the two convex polyhedra (A, B), the Cartesian (Cart) and barycentric (Bary) coordinates of the point of minimum norm

Output: Vertices of the new simplex (Q), The new supporting vertex (V), Indices into A and B of the points whose difference resulted in the new supporting vertex (i, j)

Algorithm: This function removes unnecessary points from the simplex Q. A point is unnecessary if the barycentric coordinate for that point is zero. It then calculates the next supporting vertex by sampling the Minkowski difference of the two tested polyhedra. It does not calculate the Minkowski sum outright.

Notes: This function is also designed for use in gjksubmex.m. Unlike PtMinNorm.mexw32, there is probably no other use for it.

findRelevantJoints.m
Input: Current joint velocities (qd), collisions, polyhedra
Output: collisions

Algorithm: This function determines which joints actively affect a collision and decides how to constrain them. This version considers only planar joints and ignores the effects of the waist and camera pan. For each active collision, all the planar degrees of freedom between the two colliding bodies are restricted. There are several exceptions written explicitly into the code where under certain conditions a link may be discounted or an extra one considered.

                When a new collision is detected, the direction of constraint is determined by the joint velocity at the time of the event. Old collision states keep their directions. A constraint set here causes cdyn2 to disallow any joint acceleration in that direction as long as the collision is active.

Notes: This function writes to collisions.

                While this simple algorithm will prevent intersection in most cases, certain special cases will break it. A case is considered to be broken if an active collision intersects after its collision event. By coordinating this function with collisionCheck it is possible for the simulation to self-diagnose these faults and adjust its behavior.

wireAnimate.m
Input: Robot object, Time vector (t), Joint position matrix with each row corresponding to a time step (q), polyhedra, Mode keyword (mode)
Output: none

Algorithm: This algorithm preprocesses an animation of the polyhedra representing a robot, then runs it. In the normal mode, every time step will be used. In the even mode, the largest step size will determine the step size of the animation, and in the discrete mode the animation step size is always set to 0.01. Preprocessing involves trimming the time vector and running forward kinematics on polyhedra for all time steps. To run the animation,

wireAnimate calls wireplot at each time step and waits as long as the new time vector dictates.

Notes: The animations run very slow compared to real time because wireplot is not fast enough, but in the even or discrete modes all time steps are at least even.

wireplot.m

Input: Vertices of polyhedra (v), Vector with indices indicating the beginning of each polyhedron in v (e)

Output: A solid wireframe plot of all polyhedra in v (Figure 1)

Algorithm: The function iterates through each polyhedron. If four or more vertices are given, the vertices are tessellated and plotted with the tetramesh function. If three or fewer vertices are given, lines are plotted connecting each vertex with the others. The colors of the polyhedrons rotate through a colormap. The axes for the figure are manually scaled to the Packbot.

Notes: This function runs relatively slowly, but gives a high-quality visualization of the Packbot model (or any other set of polyhedrons). The plot function included in the MATLAB Robotics Toolbox [2] is a better choice for animations.

Rmatrix.m

Input: Three angles (a1, a2, a3), A string argument identifying the rotation system (system = 'e' for Euler angles or 'rpy' for roll-pitch-yaw)

Output: Rotation matrix (R)

Algorithm: This subroutine computes each element of the 3x3 rotation matrix described by the inputs.

Notes: none

10.4 Known Issues

The primitive test function gjksubmex.m is still a bottleneck in the collision simulations, taking up approximately half of the computation time. This problem does not have to be permanent because although most of the computation involved in the GJK algorithm is done in MEX files, the recursive while loop is still coded in Matlab. If the entire function was converted to C MEX it would cease to be a bottleneck. That said, the simulation as it stands runs in a reasonable time unless the error tolerances are extremely small. This bottleneck should not be a major problem.