Санкт-Петербургский государственный университет

Кафедра механики управляемого движения

Белошапко Алексей Геннадьевич

Выпускная квалификационная работа бакалавра

Управление бегом антропоморфного механизма

Направление 010400

Прикладная математика и информатика

Научный руководитель,
кандидат физ.-мат. наук,
доцент
Латыпов В. Н.

Санкт-Петербург

2016

Содержание

Введение

В данной работе рассматривается один из подходов к управлению бегом антропоморфного механизма.

Антропоморфные механизмы с давних времён будоражили умы человечества. Прежде всего, их рассматривали как замена человеку в тяжёлых рабочих условиях, а также для развлекательных целей. С построением таких многофункциональных механизмов связано множество проблем, и одна из них – это проблема их перемещения в пространстве.

Ключевой особенностью таких механизмов является то, что для перемещения они используют опорные конечности (ноги), поочерёдно перенося на них свой вес. Антропоморфные механизмы используют две такие конечности.

Несмотря на то, что проблема перемещения такого механизма исследовалась практически всё последнее столетие, она до сих пор не была решена до конца. Основными проблемами стали гибридная природа человеческой походки, большое число степеней свободы антропоморфных механизмов и их неполная управляемость в общем случае. Проблема динамического удержания равновесия во время ходьбы и бега остаётся краеугольным камнем в решении задачи перемещения таких механизмов. Однако в некоторых частных случаях проблема поддаётся анализу и решению, о чём свидетельствует успех некоторых исследовательских и коммерческих робототехнических проектов.

Отличительной особенностью перемещения многоногих механизмов часто бывает периодичность траектории движения в некоторых системах координат. Имеет смысл искать такое управление механизмом, которое приводит к устойчивой и периодической траектории его движения. Другой отличительной особенностью является наличие опорной поверхности, вдоль которой происходит движение. Контакты звеньев механизма с опорной поверхностью накладывают и снимают некоторые ограничения на движение этого механизма, что и указывает его на гибридную природу.

Удобно каждый повторяющийся период движения многоногого механизма разделять на некоторые характерные фазы, отличающиеся друг от друга разными наборами ограничений. Количество фаз зависит от характера движения и от конструкции робота. Различают два фундаментально разных стиля перемещения роботов – это бег и ходьба. Бег отличается от ходьбы наличие фазы движения, свободной от контактов с опорной поверхностью.

В данной работе рассматривается задача бега двуногого многозвенного антропоморфного механизма. Задача разделена на две меньшие подзадачи: нахождение опорной траектории модели механизма с меньшим числом степеней свободы и динамическая стабилизация полного механизма около этой опорной траектории. В качестве меньшей модели выступает двухзвенный одноногий механизм, каждая нога которого является перевёрнутым маятником с пружиной. Для нахождения опорной траектории этой модели используется метод отображений Пуанкаре. В качестве полной модели робота используется плоский пятизвенный двуногий механизм. Стабилизация движения этого механизма около опорной траектории также осуществлена методом динамической оптимизации, работающего в режиме реального времени. Таким образом, достигается устойчивый к возмущающим факторам бег механизма.

Постановка задачи

Рассмотрим плоский пятизвенный двуногий механизм, изображённый на рис. 1. Этот механизм состоит из торса, соединённого с двумя ногами вращательными соединениями. Обе ноги состоят из бедра и голени, также соединённые вращательным соединением. Обозначим - положение тазобедренного соединения, - ориентации звеньев в абсолютной системе координат, – длина и масса торса, - длина и масса бедренного звена, – длина и масса голени, - масса и длина стопы. Особенность соединения стопы рассмотрена на рис.2. Для простоты предположим, что центр массы звена всегда совпадает с его серединой.

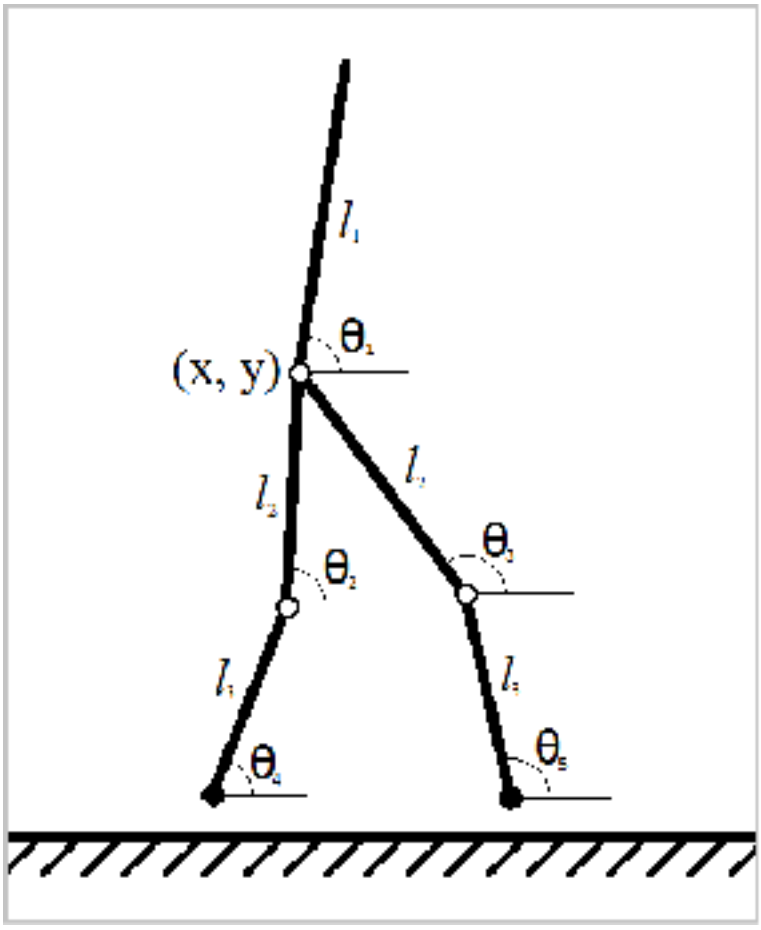

Рис. 1: Пятизвенный двуногий механизм

В соединениях между звеньями предполагаются двигатели, способные создавать момент силы между этими звеньями. Обозначим моменты этих сил как – моменты сил между торсом и правым бедром, между торсом и левым

бедром, между правым бедром и правой голенью, между левым бедром и левой голенью соответственно.

Нашей задачей является нахождение такого , который приведёт к динамически устойчивому бегу этого механизма. В дальнейших главах подробно описан подход к подбору требуемого управления.

В общих чертах он разбит на два этапа. Первым является вычисление траектории центра масс и положений ступней с помощью модели меньшего порядка, а именно с помощью расширенной модели перевёрнутого маятника с пружиной. Описывается как подход к подбору управления этой моделью с удержанием горизонтальной скорости центра масс, так и подход к смене горизонтальной скорости центра масс. Все вычисления проводятся заранее для всего ожидаемого диапазона скоростей.

Второй этап использует полученную опорную траекторию центра масс, а также некоторые другие эвристические характеристики для формулировки задачи численной оптимизации управления, с целью стабилизацию заданных характеристик движения около заданных значений. Решение этой задачи происходит в режиме реального времени. Одной из дополнительных эвристических характеристик может являться удержание центрального момента робота около нулевого значения.

Обзор литературы

Применительно к бегу двуногого антропоморфного механизма, революционной в этой области стала работа М. Райберта[1], опубликованная в 1986 году. Райберт рассмотрел и решил задачу перемещения одноногого прыгающего механизма, а затем предложил решать задачу бега многоногих механизмов, сводя её к перемещению набора одноногих механизмов, что существенно упрощало анализ проблемы. Этот подход доказал свою эффективность на прототипе, способном устойчиво бежать в плоскости, свободно меняя скорость передвижения. Ключевой особенностью управления являлось создание такого усилия в опорной ноге, какое могла бы создать некоторая жёсткая пружина. Управляя жёсткостью этой виртуальной пружины, а так же динамически выбирая точку опоры робота, оказалось возможным добиться эффективного и устойчивого перемещения механизма в пространстве.

Идея перемещения прыжками на одной ноге сама по себе не являлась новой: аналогичные задачи рассматривал Ларин[2](1978). Этот советский математик рассмотрел простую трёхзвенную модель робота с ногами переменной длины. Задача управления разделялась на две подзадачи: нахождение периодической опорной траектории движения и стабилизация относительно неё. Опорное управление, приводящее к бегу по опорной траектории, было выбрано специального вида и зависело только от текущей длины опорной ноги. Для такого управления оказалось возможным построение траектории в явном виде. Параметры управления выбирались исходя из соображений периодичности траектории. Для стабилизации движения был использован метод оптимальной стабилизации около опорной траектории и опорного управления. Данный подход был проверен только численно.

Эти две работы рассматривали управление достаточно простыми двуногими механизмами с призматическими коленями. Дальнейшие работы были адресованы проблеме управления двуногими многозвенными

механизмами с большим количеством степеней свободы, имеющими также вращательные соединения.

Одну из методик подбора контроллера предложили Томас Гейчтенберг и его коллеги [3]. Управление осуществлялось с помощью пропорционально-дифференциального контроллера, параметры которого находились с помощью генетических алгоритмов оптимизации. Подход является достаточно общим для различных топологий механизма, а также для разных способов перемещения – ходьба и бег, прыжки на двух ногах. Желаемые особенности походки задавались в виде желаемых поз каждой фазы движения. Подход был проверен численно на механизмах различной топологии. Подбор параметров управления мог занять от 2 до 12 часов на персональном компьютере.

Другой подход к решению этой проблемы предложила группа исследователей [4]. В их работе для уменьшения степеней свободы задачи предлагалось накладывать на движение робота так называемые *виртуальные ограничения*. Учёт этих ограничений происходил при подборе управления. Таким образом, задача перемещения робота в пространстве сводилась к дизайну контроллера, асимптотически стабилизирующего степень нарушения этих ограничений около нуля. Для поиска периодической траектории механизма был использован метод отображений Пуанкаре (Poincaré return map). Данный подход к управлению был реализован на пятизвенном плоском роботе RABBIT. Численные эксперименты показали способность этого робота устойчиво бежать при заданной скорости бега.

Несколько по-другому была решена эта задача у Венсинга[5]. Как и в работе Ларина, задача была разложена на две меньшие: построение опорной траектории и стабилизация около неё. Однако, опорная траектория задавалась для определённых характеристик робота, таких как положение центра масс, положение стоп, значение момента сил вокруг центра масс робота. Задача поиска опорной траектории для центра масс была решена с помощью более простого механизма с двухзвенными ногами переменной длины. Управление

в суставе опорной ноги было таким, как если бы в ноге была жёсткая пружина. Такой механизм мог перемещаться прыжками, имитируя бег двуногого механизма. Далее поиск параметров этой пружины осуществлялся с помощью методов численной оптимизации. Стабилизация движения робота около опорной траектории также была осуществлена методом многомерной оптимизации, что помогло уйти от проблем, связанных с большим числом степеней свободы в многозвенной ноге. Метод способен работать в режиме реального времени

Глава 1. Подбор опорной траектории центра масс робота

§1.1. Описание упрощённой модели.

Далее излагается подход, предложенный Венсингом [6]. Необходимо рассмотреть упрощённую модель робота – четырёхзвенного робота с призматическими коленями, в которые встроена виртуальная механическая пружина (рис. 2).

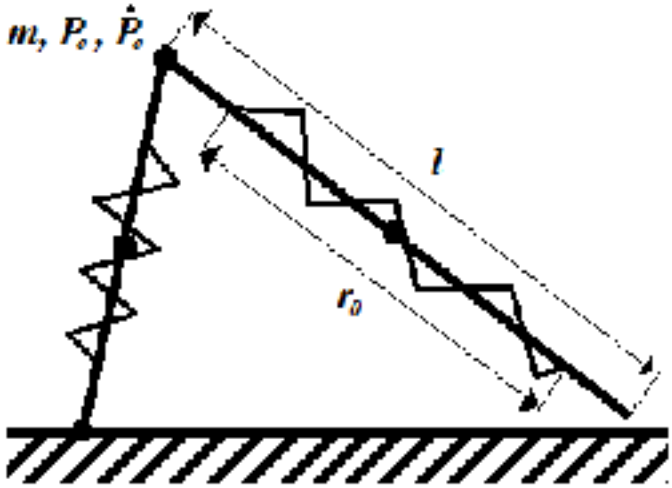

Рис. 2: Упрощённая двуногая модель

Ноги робота предполагаются безмассовыми, таким образом, вся масса робота сконцентрирована в точке, соединяющей ноги вращательным соединением. Вектором обозначим координаты этой точки, а вектором - вектор производных этих координат. Каждая из виртуальных пружин добавляет силу Гука между соседними звеньями ног, и имеет жёсткость и длину в нерастянутом состоянии. При этом сама нога имеет длину .

Данный механизм разработан исключительно для перемещения прыжками на каждой ноге поочерёдно, что моделирует бег двуногого робота. При этом робот может находиться в двух состояниях: либо он стоит на одной ноге, либо испытывает период чистого полёта. Эти состояния чередуются в

таком порядке: фаза полёта – стойка на одной из ног – фаза полёта – стойка на второй ноге. Всех остальных состояний – стойки на двух ногах и касания центра масс робота опорной поверхности – мы намеренно избегаем.

Так как в каждой фазе движения этого робота нас интересует только одна нога: либо опорная, либо следующая по очереди для опоры, а так же потому, что ноги не имеют массы, то можно предполагать механизм одноногим (рис. 3).

Рис. 3: Упрощённая одноногая модель

Обозначим координаты точки конца ноги робота (ступни) как , а угол наклона ноги относительно горизонта - . Таким образом, во время фазы полёта

Во время стойки конец ноги неподвижен и находится на опорной поверхности. Обозначим

Тогда уравнение центра масс принимает вид

Переключение фаз происходит при пересечении состояния модели поверхностей переключения

-
- ,

где – орт второй оси координат.

§1.2. Отображения Пуанкаре

Управление этой моделью осуществляется путём выбора угла наклона ноги в момент приземления робота, а также путём выбора жёсткости пружины. Цель управления механизмом – управление горизонтальной скоростью и высотой полёта в момент времени фазы полёта, соответствующий наивысшему положению центра масс. Обозначим этот момент как , а за обозначим интересующие нас величины состояния

где ***, j*** – орты координатных осей, - горизонтальная скорость центра масс маятника в момент времени , - вторая координата положения центра масс (высота) в момент . Обозначим также управление

где - угол наклона ноги в момент перехода в фазу стойки, , - жёсткость пружины до и после момента максимального сжатия пружины.

Таким образом, можно выделить 4 циклические фазы движения маятника: фаза падения, фаза сжатия пружины в положении стойки, фаза расслабления пружины в фазе стойки и фаза взлёта (рис. 4). Первая фаза начинается в наивысшей точке полёта маятника, этой точке соответствует нулевой момент времени . В этот момент происходит управление углом наклона ноги. Переход во вторую фазу движения осуществляется в момент касания ноги опорной поверхности, и продолжается до момента максимального сжатия пружины. В момент максимального сжатия пружины происходит смена коэффициента жёсткости пружины. Третья фаза

завершается моментом отрыва ноги от опорной поверхности. Четвёртая фаза завершается в момент равенства вертикальной скорости центра масс нулю.

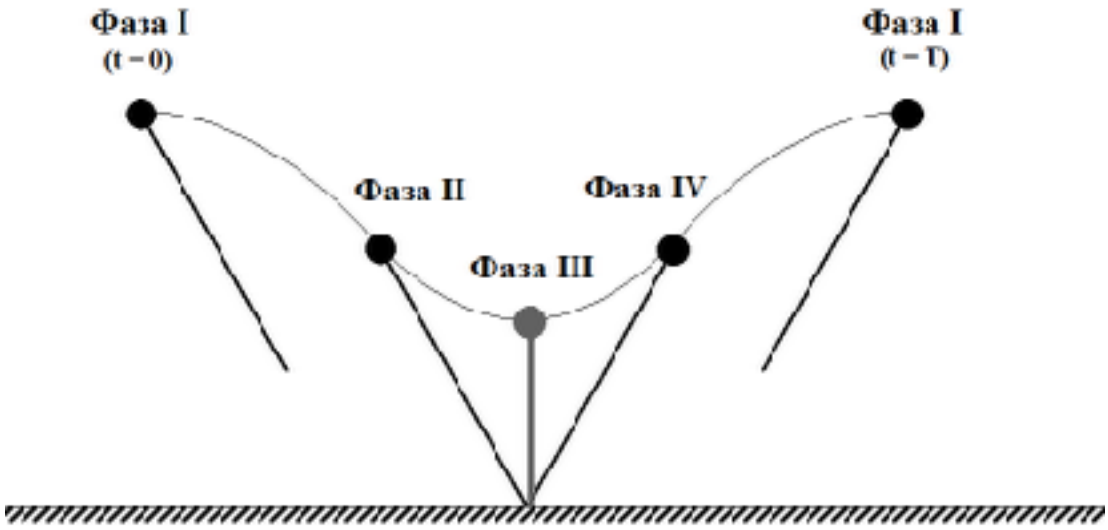

Рис. 4: Фазы движения перевёрнутого маятника

Теперь можно ввести отображение состояний маятника в моменты его наивысшего подъёма. Пусть - состояние механизма в момент перехода в фазу I -ного цикла. Тогда следующая формула связывает следующее начальное состояние с предыдущим

Где - управление маятником в цикле . Функция в этом случае называется отображением Пуанкаре.

В дальнейшем, нашей задачей станет нахождение для данного начального состояния такого управления , чтобы

что будет означать нахождение периодической траектории.

§1.3. Формулировка задачи оптимизации

В целом, требование периодичности траектории данной модели маятника оставляет некоторую свободу в выборе управления. Поэтому

введём ещё одно требование, характеризующее ритм бега. Исследование экспериментальных данных о человеческом беге, предоставленных Роуландсом[6] позволило сделать следующий вывод: с увеличением скорости количество шагов в минуту растёт, а среднее время стойки на опорной ноге уменьшается. Эти величины могут быть приблизительно вычислены по следующим формулам

(1.3.1)

Последнее уравнение будет использовано как дополнительное ограничение: желаемое время стойки на опорной ноге будет определено горизонтальной скоростью центра масс

Моменты начала стойки и окончания стойки могут быть путём интегрирования уравнений соответствующей механической системы. Этот способ вычисления обозначим функцией Таким образом, нашей задачей становится подбор такого управления для состояния , чтобы

Данная задача может быть переформулирована для решения методом наименьших квадратов

(1.3.2)

§1.4. Результаты численных экспериментов.

Для получения описанных опорных траекторий был реализован простой интегратор перевёрнутого маятника. Для решения задачи (1.3.2) был использован метод оптимизации Левенберга-Марквардта, реализация которого была адаптирована из математической библиотеки dlib. Вычисления проводились на домашнем ноутбуке ASUS Eee с процессором AMD 450 APU 1.65 Ггц и заняли около 10 минут.

Величина точечной массы перевёрнутого маятника была равна суммарной массе пятизвенного робота (63.25 кг), длина опорного стержня – 0.9м, длина нерастянутой пружины – 0.7м.

Подбор параметров управления осуществлялся последовательно по всему диапазону горизонтальных скоростей от 0 до 8 м/с с шагом в 0.2 м/с. Параметры управления, полученные как результат для предыдущей скорости, были использованы как начальное предположение для следующей. Начальная жёсткость пружины равна , начальная высота центра масс – 0.95м, угол наклона стержня – 0 градусов.

Модель времени стойки из уравнения (1.3.1) была признана адекватной для горизонтальных скоростей, начиная с 3 м/с. Для скоростей ниже этой желаемое время стойки на опорной ноге было приблизительно оценено в 0.3 секунды.

Графики отражают процесс изменения параметров движения маятника от его горизонтальной скорости во время экспериментов. Начальная высота центра масс меняется, увеличиваясь до 1.2 м (гр. 2). Угол наклона опорного стержня также равномерно растёт, достигая (гр. 1). Несмотря на то, что точно следовать желаемому времени стойки робота на земле не удалось, эта величина варьируется в допустимом интервале (гр. 3). Практически экспоненциальное уменьшение жёсткости пружины (гр. 4) отражает тот факт, что с увеличением скорости взаимодействие с опорной поверхностью носит всё более мягкий характер.

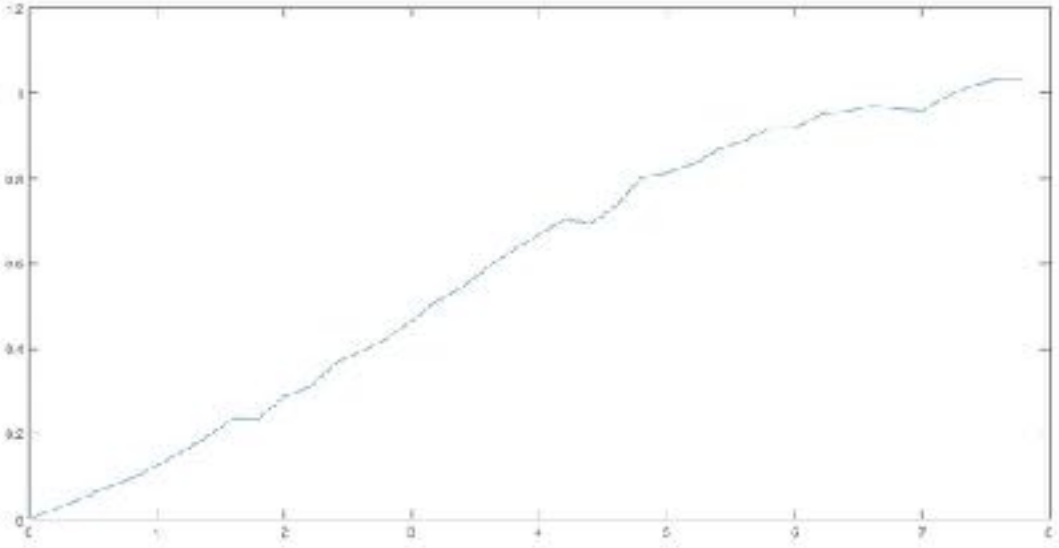

График 1: Зависимость начального угла наклона стержня от горизонтальной скорости модели

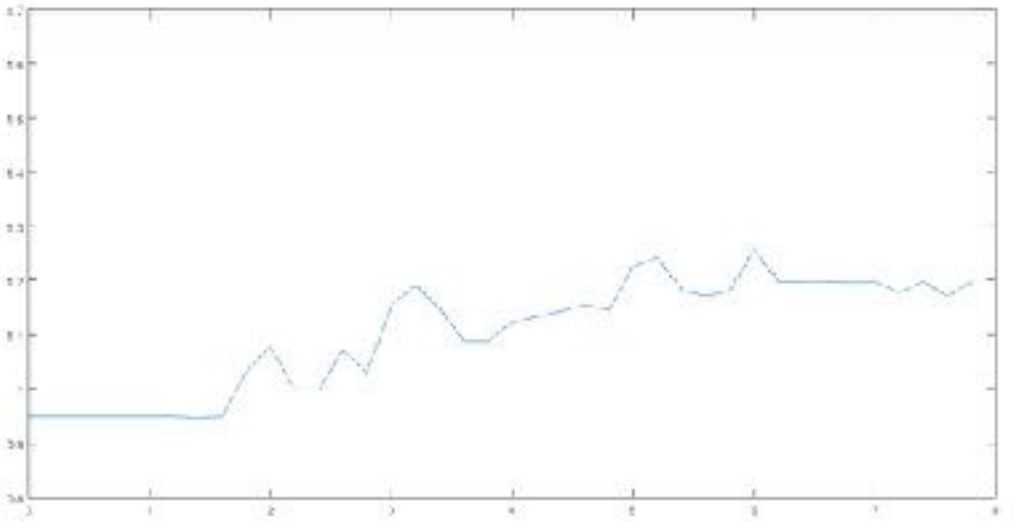

График 2: Зависимость начальной высоты центра масс от горизонтальной скорости модели

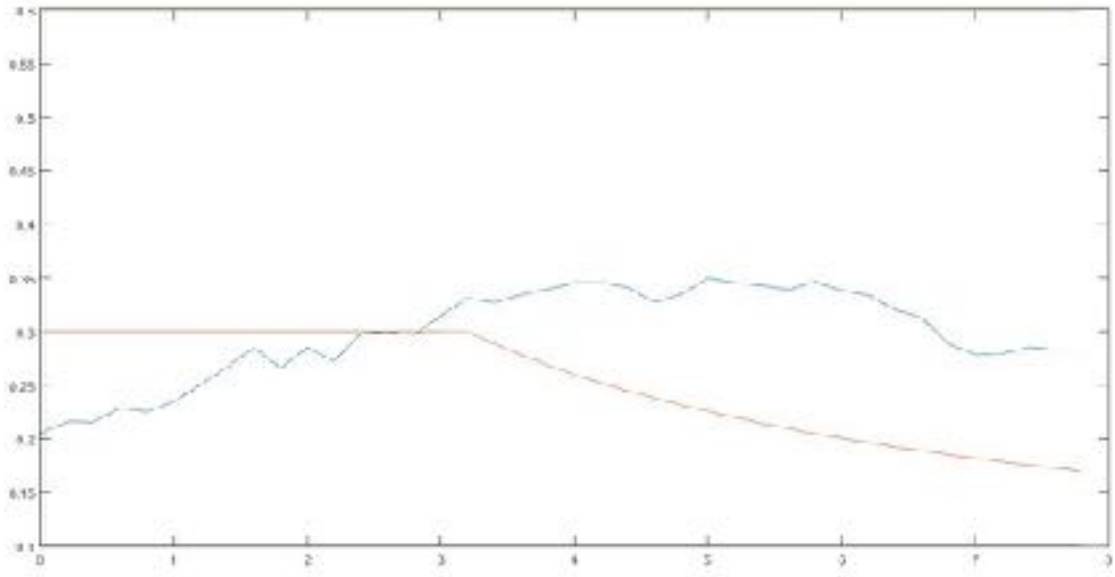

График 3: Синий график: зависимость времени стойки модели от горизонтальной скорости.

Красный: желаемая зависимость.

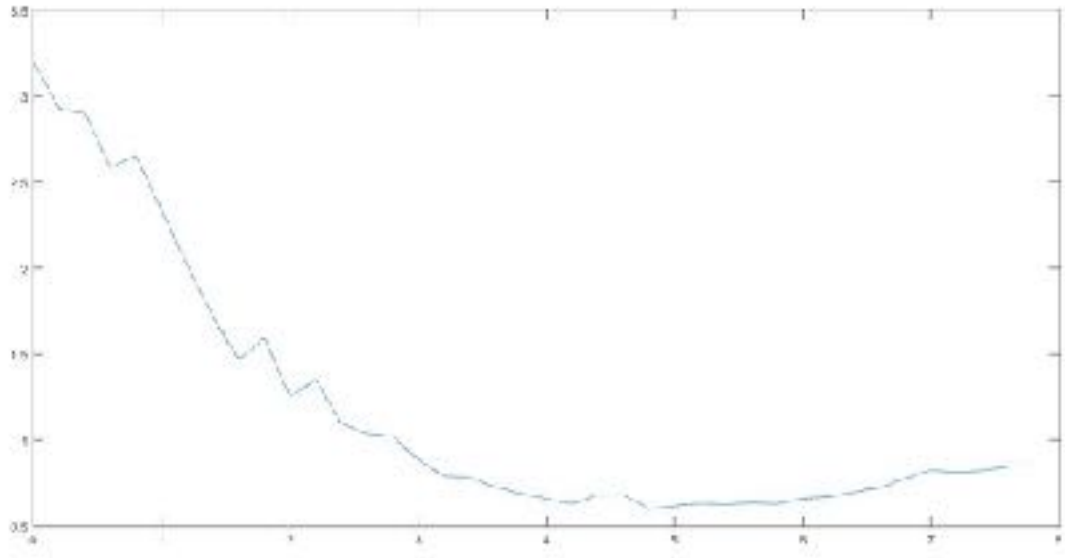

График 4: Зависимость жёсткости виртуальной пружины от горизонтальной скорости

§1.5. Стабилизация модели около опорной траектории

В этой главе будут описаны подходы к управлению скоростью рассмотренной модели. Имея для определённого начального состояния робота управление, приводящее к периодической траектории движения, необходимо выработать правило подбора управления гашения отклонений. Два подхода будут рассмотрены, одно из которых оперирует с горизонтальной скоростью механизма, а второй – с состоянием робота.

Подход 1.

В отчёте MIT Leg Laboratory от 1989 года [2] был предложен подход к управлению скоростью одноногого прыгающего робота с призматической ногой, основанный на оценке времени стойки. По своей сути, он является модификацией ПИД-регулятора. Зная время стойки , текущую горизонтальную скорость , и желаемую горизонтальную скорость , положение стопы робота относительно центра масс может быть вычислено следующим образом

Где - небольшая положительная константа. Данный подход подходит для управления скоростью при не очень больших отклонениях. Кроме того, аналогичный контроллер может контролировать скорость и по второй горизонтальной оси в случае трёхмерного механизма.

Подход 2.

Этот подход, предложенный Венсингом[6], также является модификацией пропорционального контроллера, однако пригоден также и к управлению жёсткостью пружины. Пусть – начальное состояние и управление, приводящее к периодической траектории. Тогда

.

Имея отклонение в начальном состоянии робота (в верхней точке полёта), поправка к управлению может быть принята такой, что

Учитывая, что якобиан может иметь полный ранг, поэтому добавим ещё одно ограничение, приняв поправки к жёсткости пружины противоположными

Получаем выражение для поправки управления

(1.5.1)

В уравнении (1.5.1) матрица фиксирует вышеуказанное правило выбора изменения жёсткости пружины.

Глава 2. Управление механизмом в пространстве задач

В данной главе описан подход к управлению антропоморфным механизмом в пространстве задач. Отличительной особенностью выбранного подхода является относительная простота применения и способность адаптироваться к изменчивой среде движения. Далее приводится пример применения этого метода к упрощённому антропоморфному механизму.

Подход к управлению роботом в пространстве задач призван упростить задачу описания желаемого движения робота по сравнению с методами обратной динамики. Например, задача бега двуногого механизма может быть описана как траектории центра масс робота и его ступней. Также он легко адаптируется к бегу по неровной или неточно известной опорной поверхности [6].

Решение задачи управления роботом производится методом численного решения соответствующей задачи квадратичного программирования. Задача квадратичного программирования формулируется таким образом, чтобы избегать нарушения ограничений на силы реакции опоры, т. е., чтобы избегать незапланированного скольжения или отрыва опорной ноги. Поиск решения производится среди класса кусочно-постоянных управлений с постоянным временем дискретизации.

§2.1. Пространство задач

Далее будем предполагать, что рассматриваемый робот подчиняется следующим уравнениям

Эти уравнения являются уравнениями Лагранжа II-го рода. – обобщённые положение, скорость и ускорение соответственно, Матрица – якобиан поддержки, отображающий обобщённую скорость в матрицу линейных скоростей точек поддержки робота, а – вектор сил реакции опоры.

Пусть – произвольная характеристика движения робота, зависящая только от обобщённого положения робота, т. е.

(2.1.1)

Тогда первая производная данной характеристики может быть вычислена

Где . С другой стороны, характеристика может быть задана в виде

(2.1.2)

Уравнение (2.1.2) определяет характеристику в более общем виде, нежели уравнение (2.1.1), так как в общем случае - может не быть якобианом какой-либо функции.

Управление рассматриваются из класса кусочно-постоянных функций. Пусть стоит задача управления механизмом на промежутке времени . Тогда управление имеет вид

Пусть нам дана желаемая характеристика движения в виде . Так как поиск управления происходит в реальном времени, мы заинтересованы в вычислении управления по времени не далее, чем на вперёд.

Тогда задачей управления в момент времени назовём поиск управления , соответствующего наиболее близкому удержанию выбранной характеристики в момент времени

где

либо

Выбранная характеристика может не полностью определять движение робота. Тогда допустимо определить несколько желаемых характеристик движения с различными приоритетами. Решение задачи управления для

удержания набора характеристик называется *управлением в пространстве задач*.

§2.2. Формулировка задачи квадратичного программирования для поиска управления

Задача поиск управления, приводящего к удержанию выбранной характеристики робота, эквивалентна следующей задаче квадратичного программирования

Где , – верхняя и нижняя границы для компонент вектора управляющих моментов, – количество звеньев робота, способных касаться опорной поверхности, – область допустимых сил реакции опоры. Для плоского двуногого бегающего робота .

Для удержания сразу нескольких характеристик, два подхода применимы: это решение цепочки квадратичных задач, начиная с задачи с максимальным приоритетом, или решение квадратичной задачи со сложной целевой функцией, являющейся линейной комбинацией целевых функций каждой характеристики в отдельности.

В этой работе используется второй способ. Общая задача квадратичного программирования для поиска подходящего управления примет вид

Где – ошибка удержания характеристики .

§2.3. Пример применения управления простым механизмом в пространстве задач.

Рассмотрим простой механизм (рис 5), представляющий собой ногу робота и состоящий из трёх звеньев: бедра, голени и стопы.

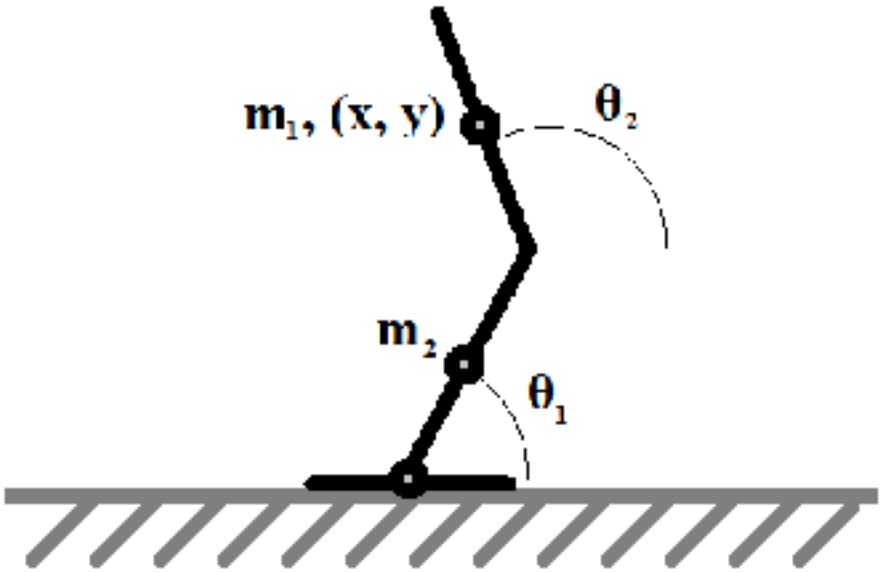

Рис. 5: Простой трёхзвенный механизм

Предположим, в данном примере что стопа плоско стоит на опорной поверхности и не отрывается. Тогда можно ввести обобщённые координаты , где - угол отклонения бедра от горизонтали, - угол отклонения голени от горизонтали. Начало системы координат поместим в точку соединения стопы и голени, ось направим вдоль опорной поверхности, ось направим ортогонально оси и вверх.

Пусть нашей задачей является перемещение центра масс голени точно в определённое место. Данную задачу можно решить и более простыми способами, однако, чтобы наша цель здесь – продемонстрировать пример применения управления в пространстве задач.

Задачей в этот раз является перевод центра масс голени с координатами из точки в точку по заданной траектории. Траекторию выберем так, чтобы в начальной и конечной точке скорость центра масс бедра была нулевой

Выразим положение точки интереса через обобщённые координаты

Обозначим

Тогда задача может быть сформулирована следующим образом

Целевая функция примет вид

Глава 3. Управление пятизвенным антропоморфным роботом в пространстве задач

§3.1 Описание задач

В этой главе будут рассмотрены детали управления пятизвенным механизмом в пространстве задач. В связи с большим числом степеней свободы такого механизма, для эффективного управления его бегом необходимо задать достаточно много характеристик движения, которые ему следует удерживать. Набор удерживаемых характеристик должен однозначно определять движение робота как бег.

Для пятизвенного механизма необходимо обозначить:

- траекторию центра масс робота
- траекторию движения стоп робота
- задачу на удержание углового момента скорости около нуля

Траектория центра масс робота может быть получена способом, описанным в главе 1. Одновременно, движение перевёрнутого маятника определяет положение стопы опорной ноги робота, как и моменты времени удара опорной ноги о землю и её отрыва.

Так как стопы у рассмотренного механизма отсутствуют, будем задавать траектории для точек, находящихся на концах ног. Необходимо рассмотреть два разных случая: случай, когда робот находится в воздухе, и когда опирается одной ногой на поверхность.

В случае, когда робот опирается на землю, опорная нога должна лишь сохранять положение до момента перехода в фазу полёта. Переносимая же нога должна прийти в положение, симметричное с тем, в котором она оказалась в момент перехода в фазу стойки, то есть должна быть вынесена вперёд. Траекторию переноса конца ноги можно задать кубическим сплайном.

Если же робот находится в воздухе, то ему следует убедиться, что следующая для опоры нога находится в заданном положении. Траекторию перемещения следующей ноги для опоры в нужное положение относительно центра масс можно также можно определить сплайном.

Удержание момента скорости вокруг центра масс около нуля является важной задачей во время стойки, так как необходимо избежать ненужного вращения во время полёта.

§3.2. Эксперименты

Для проведения экспериментов была разработана тестовая среда, состоящая из интегратора, визуализатора и оптимизатора. В качестве интегратора был адаптирован физический движок ODE (Open Dynamics Engine), а в качестве механизма для решения задач минимизации – алгоритм Левенберга-Марквардта из библиотеки dlib. Так как этот алгоритм не может самостоятельно учитывать ограничения в явном виде, ограничения включались в неявном виде в функционалы качества, как методом суперпозиции, так и методом штрафов.

Было проведено моделирование ключевого элемента шага – момента приземления на опорную поверхность и отскока от неё (рис. 6). В качестве опорной траектории для центра массы робота была выбрана траектория

движения перевёрнутого маятника, летящего с горизонтальной скоростью в 2.5 м/с. Недостаток управляемости роботом, обусловленный отсутствием у него стоп, вызвал некоторое отклонение в конечном состоянии робота от желаемого значения. Однако данная погрешность может быть устранена во время последующих шагов методами стабилизации, описанными в §1.5.

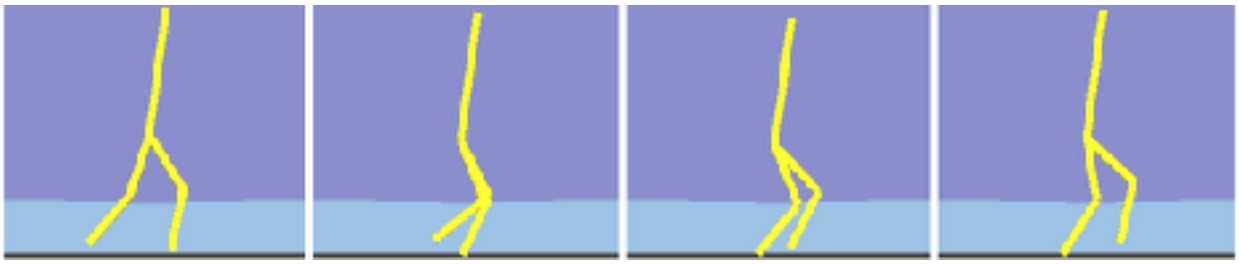

Рис. 6: Моделирование периодической части движения робота в указанном окружении.

Заключение

В данной работе рассмотрена задача бега пятизвенного антропоморфного механизма. Для решения этой задачи был выбран подход к управлению в пространстве задач. Была решена вспомогательная задача получения опорной траектории для центра масс, необходимой для работы выбранного алгоритма управления. Проведены эксперименты, подтверждающие применимость данного метода к решению выбранной задачи.

Несмотря на простоту выбранной модели механизма, подход достаточно общий как для плоских моделей с большим количеством степеней свободы, так и для трёхмерных механизмов.

Решение задачи движения антропоморфного робота крайне важно для развития механизмов такого рода в целом. Так как в перспективе человекоподобные роботы могут заменить человека в сложных рабочих условиях, и даже существенно расширить границы этих рабочих условий, то создание прототипа, способного устойчиво перемещаться в пространстве, является весьма актуальной задачей. Постоянное увеличение вычислительной мощности бортовых компьютеров, которыми можно оснащать роботов для управления, открывает дорогу всё новым алгоритмам и подходам к управлению. Описанный в этой работе подход является одним из них.

Список литературы

1. Raibert M. H., Brown B. H., Chepponis M., Koechling J., Hodgins J. K., Dustman D., Brennan K. W., Barrett D. S., Thompson C. M., Hebert J. D., Lee W., Borvansky L. Technical Report 1179 LL-6: Dynamically Stable Legged Locomotion. MIT Cambridge, Massachusetts, 1989. 203 p.
2. В. Б. Ларин. Управление локомоционными системами. Институт математики АН УССРб 1978. 52 с.
3. Geijtenbeek T., van der Panne M., van der Stappen A. F. Flexible Muscle-Based Locomotion for Bipedal Creatures. ACM Transactions on Graphics, 2013. Vol. 32, No. 6.
4. Westervelt E. R., Grizzle J. W., Chevallereau C., Choi J. H., Morris B. Feedback Control of Dynamic Bipedal Robot Locomotion. CRC PRESS, 2007. 503 p.
5. Wensing P. M. Optimization and Control of Dynamic Humanoid Running and Jumping: phD dissertation. The Ohio State University, 2014.
6. Rowlands A., Stone M., Eston R. Influence of speed and step frequency during walking and running on motion sensor output. Med. And Science in Sports and Exercise, 2007. Vol 39, No 4. P. 716-727.

Приложение

Исходный код программы на C++:

RunningRobot.h:

```
#ifndef _RUNNING_ROBOT_H_
#define _RUNNING_ROBOT_H_

#include "RobotState.h"
#include "MechanismIntegrator.h"
#include "OdeIntegrator.h"
#include "MechanismTask.h"
#include "RobotFeatureDescriptor.h"
#include "Optimizer.h"
#include "RobotVisualizer.h"
#include "CubicSplines.h"

#include <Eigen/Dense>
#include "MatrixOperations.h"
#include "CommonOperations.h"
#include <math.h>
#include <process.h>

class RunningRobot
{
public:
	RunningRobot(Visualizer* visualizer);
	void buildDescriptor();
	void initializeOptimization();
	void adjustTasks(std::vector<pReal>& q);

	void debugInfo(std::vector<pReal>& control);
	void runOptimization(std::vector<pReal>& control);
	void runControl();

	void run();
	void startTest();

private:
	MechanismDescription::MechanismDescriptor*
mMotherDescriptor;
	MechanismIntegrator* mOptIntegrator;
	MechanismIntegrator* mTestIntegrator;

	Visualizer* mVisualizer;
	MechanismVisualizer* mRobotVisualizer;

	TaskObjective* mTasks;
	LevenbergOptimizer<pReal>* mOptimizer;
```

```
	int mControlDim;
	pReal mTimeStep;
	pReal mCurrTime;

	//debug options
	HopperFeature* mDesiredComVel;
	MechanismFeature* mRealComVel;

	HopperFeature* mDesiredComPos;
	MechanismFeature* mRealComPos;

	MechanismFeature* mRealTorsoVel;

	RelativeBackFootTrajectory* mDesiredBF;
	MechanismFeature* mRealBF;

	HopperFeature* mDesiredSFPos;
	MechanismFeature* mRealSFPos;

	HopperFeature* mDesiredSFVel;
	MechanismFeature* mRealSFVel;

	MechanismTask* velocityDamp;
};

#endif
```

RunningRobot.cpp:

```
#include "RunningRobot.h"

using namespace MechanismDescription;
using namespace Eigen;

void printVector(std::vector<pReal>& v)
{
	std::cout << "[";
	for(int i = 0; i < v.size(); i++)
		std::cout << v[i] << ", ";

	std::cout << "]\n";
}


pReal adjustAngle(std::vector<pReal> real, std::vector<pReal>
desired)
{
	pReal realAngle = std::atan2(real[1], real[0]);
	pReal desiredAngle = std::atan2(desired[1], desired[0]);
```

```
     return realAngle - desiredAngle;   //remember about
different angle convention
}

RunningRobot::RunningRobot(Visualizer* visualizer)
{
     mVisualizer = visualizer;
}

void runTest(void* lpParams)
{
     RunningRobot* test = (RunningRobot*) lpParams;

     test->run();
}

void getInitialCoords(std::vector<pReal>& q);

void RunningRobot::buildDescriptor()
{
     mMotherDescriptor = new MechanismDescriptor(3);

     Matrix<pReal, 3, 3> inertia = Matrix<pReal, 3, 3>::Zero(3,
3);
    inertia(0, 0) = 1.0;
     inertia(1, 1) = 1.0;
     inertia(2, 2) = 1.0;

     Matrix<pReal, 3, 1> center = Matrix<pReal, 3, 1>::Zero(3,
1);
     center(0, 0) = 1.0;

     Matrix<pReal, 3, 1> anchor = Matrix<pReal, 3, 1>::Zero(3,
1);
     anchor(0, 0) = -0.4;

     Body* torso = new Body("torso", 42.0, center, inertia);
     Body* left_leg = new Body("left_leg", 7.25, center,
inertia);
     Body* right_leg = new Body("right_leg", 7.25, center,
inertia);
     Body* left_foot = new Body("left_ankle", 3.375, center,
inertia);
     Body* right_foot = new Body("right_ankle", 3.375, center,
inertia);

     std::vector<pReal> dims;
     dims.resize(3);
     dims[0] = 0.8; dims[1] = 0.05; dims[2] = 0.05;

     torso->setDimentions(dims);
```

```
	dims[0] = 0.4;
	left_leg->setDimentions(dims);
	right_leg->setDimentions(dims);


	left_foot->setDimentions(dims);
	right_foot->setDimentions(dims);

	MechanismDescription::Joint* hip1 = new
MechanismDescription::Joint("hip1",
		new RevoluteJoint(0.2),
		*torso,
		*left_leg,
		anchor,
		rotate_clockwise_z(0)
		);
	MechanismDescription::Joint* hip2 = new
MechanismDescription::Joint("hip2",
		new RevoluteJoint(0.2),
		*torso,
		*right_leg,
		anchor,
		rotate_clockwise_z(0)
		);

	anchor(0, 0) = -0.2;

	MechanismDescription::Joint* lKnee = new
MechanismDescription::Joint("lKnee",
		new RevoluteJoint(0.2),
		*left_leg,
		*left_foot,
		anchor,
		rotate_clockwise_z(0)
		);

	MechanismDescription::Joint* rKnee = new
MechanismDescription::Joint("rKnee",
		new RevoluteJoint(0.2),
		*right_leg,
		*right_foot,
		anchor,
		rotate_clockwise_z(0)
		);

	mMotherDescriptor->addBody(torso);
	mMotherDescriptor->addBody(left_leg);
	mMotherDescriptor->addBody(right_leg);
	mMotherDescriptor->addBody(right_foot);
	mMotherDescriptor->addBody(left_foot);
```

```
      mMotherDescriptor->addJoint(hip1);
      mMotherDescriptor->addJoint(hip2);
      mMotherDescriptor->addJoint(rKnee);
      mMotherDescriptor->addJoint(lKnee);

      mControlDim = 7;
}

void RunningRobot::initializeOptimization()
{
/*integrator initialization*/
      mTimeStep = 0.01;
      mCurrTime = 0;

      AbstractIntegrator* integrator = new Ode::OdeIntegrator();
      mOptIntegrator = new MechanismIntegrator(integrator,
mMotherDescriptor);

      std::vector<pReal> q;
      q.resize(7);
      getInitialCoords(q);

      mOptIntegrator->build(q);
/*end*/

/*features initialization*/
      HopperLoader loader("2.5.log");
      loader.load();

      //com pos
      mDesiredComPos = new HopperFeature(&loader, 0);
      mRealComPos = new MechanismCOMPos();

      //com vel
      mDesiredComVel = new HopperFeature(&loader, 1);
      mRealComVel = new MechanismCOMVel();

      //foot pos
      mDesiredSFPos = new HopperFeature(&loader, 2);
      mRealSFPos = new MechanismInternalPointPos("right_ankle",
-0.2, -0.025);

      //foot vel
      mDesiredSFVel = new HopperFeature(&loader, 3);
      mRealSFVel = new MechanismInternalPointVel("right_ankle",
-0.2, -0.025);

      //back foot pos
      BackFootTrajectory* cuboid = new BackFootTrajectory(
            0.35,
            0.1,
```

```
            0.4,
            0.1,
            0.4,
            0.2
            );
     mDesiredBF = new RelativeBackFootTrajectory(cuboid);  //
cubic spline
     mDesiredBF->setRelativeFeature(mDesiredComPos);

      mRealBF = new MechanismInternalPointPos("right_ankle",
-0.2, 0.025);

     FeatureDescriptor* bfTrajVel = new
FeatureDerivative(mDesiredBF, 1e-3);    //spline derivative
     MechanismFeature* backFootVel = new
MechanismInternalPointVel("right_ankle", -0.2, 0.025);//back
leg end's velocity
/*end*/

/*tasks creation*/
     MechanismTask* torsoPosTask = new
MechanismTask(mDesiredComPos, mRealComPos); //com pos
     MechanismTask* torsoVelTask = new
MechanismTask(mDesiredComVel, mRealComVel); //com vel
     MechanismTask* bFPosTask = new MechanismTask(mDesiredBF,
mRealBF);                //back foot pos task
     MechanismTask* bFVelTask = new MechanismTask(bfTrajVel,
backFootVel);            //back foot vel task
     MechanismTask* sFPosTask = new MechanismTask(mDesiredSFPos,
mRealSFPos);        //stance foot pos task
     MechanismTask* sFVelTask = new MechanismTask(mDesiredSFVel,
mRealSFVel);        //stance foot vel task
/*end*/

/*task collection creation*/
     mTasks = new TaskObjective(mOptIntegrator, mControlDim - 3,
mTimeStep);
     mTasks->addTask(torsoVelTask, 1e-2);
     mTasks->addTask(torsoPosTask, 1e-2);
     mTasks->addTask(sFPosTask, 1.0);
     mTasks->addTask(sFVelTask, 1.0);
     mTasks->addTask(bFPosTask, 1e-2);
     mTasks->addTask(bFVelTask, 1e-5);
/*end*/

/*other initialization actions*/
     //reinit mech according to task init conditions
     adjustTasks(q);

     //prepare optimizer
     mOptimizer = new LevenbergOptimizer<pReal>(1e-6);
```

```
	mOptimizer->addMinimizable(mTasks);

	//init visualizer
	mRobotVisualizer = new MechanismVisualizer(mVisualizer,
mOptIntegrator->getEndDescriptorP());
	mRobotVisualizer->build();
/*end*/
}

void RunningRobot::runOptimization(std::vector<pReal>& control)
{

	TaskOptimizerWrapper optWrapper(mOptimizer, 1.0);
	optWrapper.solve(control);
}

void RunningRobot::runControl()
{
	//initialize control params
	std::vector<pReal> control;
	control.resize(mControlDim);
	std::fill(control.begin(), control.end(), pReal(0));

	while(mCurrTime < 30)
	{
		mTasks->setCurrentTime(mCurrTime);	//setting time
to tasks
		runOptimization(control);			//getting
control values

		mOptIntegrator->step(control, this->mTimeStep);  //
stepping mech with accepted control
		mCurrTime += mTimeStep;
		mOptIntegrator->acceptStep();

		debugInfo(control);

		while(!mVisualizer->isInitialized());	//waiting
for visualizer
		mRobotVisualizer->update();			//
visualizer update
	}
}

void RunningRobot::run()
{
	buildDescriptor();
	initializeOptimization();
	runControl();
}
```

```
void RunningRobot::startTest()
{
	_beginthread( &runTest, 0, this);
}
```

MechanismTask.cpp:

```
#ifndef _MECHANISM_TASK_H_
#define _MECHANISM_TASK_H_

#include "RobotState.h"
#include "RobotFeatureDescriptor.h"
#include "Optimizer.h"
#include "MechanismIntegrator.h"

#include <math.h>

using namespace MechanismDescription;

class MechanismTask
{
public:
	MechanismTask(FeatureDescriptor* desired, MechanismFeature*
real);

	pReal getDifference();

	void
fetchFeature(MechanismDescription::MechanismDescriptor*
realDescr);
	void setScaling(std::vector<pReal>& scaling);
	void setTime(pReal time);

private:
	pReal weightedSquareDifference();

	FeatureDescriptor* mDesired;
	MechanismFeature* mReal;

	int mDim;
	std::vector<pReal> mDesiredVal;
	std::vector<pReal> mRealVal;
	std::vector<pReal> mScaling;

	pReal mCurrentTime;
};

class TaskObjective : public ObjectiveComponent<pReal>
{
public:
```

```
        TaskObjective(
                MechanismIntegrator* integrator,
                int controlSize,
                pReal integrationTime
                );

        void addTask(MechanismTask* task, pReal priorityScale);
        void setCurrentTime(pReal time);

        void evaluate(
                dlib::matrix<pReal>* result,
                const dlib::matrix<pReal, 0, 1>* arg,
                const dlib::matrix<pReal, 0, 1>* param
                ) const;

private:
        int mControlSize;
        pReal mCurrentTime;
        pReal mIntegrationTime;

        MechanismIntegrator* mIntegrator;
        std::vector<MechanismTask*> mTasks;
        std::vector<pReal> mPriorityScales;
};

class TaskOptimizerWrapper
{
public:
        TaskOptimizerWrapper(LevenbergOptimizer<pReal>* optimizer,
pReal scaling);
        pReal solve(std::vector<pReal>& control);

private:
        LevenbergOptimizer<pReal>* mOptimizer;
        pReal mScaling;
};

#endif
```

MechanismTask.cpp:

```
#include "MechanismTask.h"

using namespace MechanismDescription;

MechanismTask::MechanismTask(FeatureDescriptor* desired,
MechanismFeature* real)
{
        mDesired = desired;
        mReal = real;
```

```
    int dDim = mDesired->valueLength();
    int rDim = mReal->valueLength();

    (dDim < rDim)? mDim = dDim : mDim = rDim;

    mScaling.resize(mDim);
    std::fill(mScaling.begin(), mScaling.end(), pReal(1.0));

    mDesiredVal.resize(mDim);
    mRealVal.resize(mDim);
}

void
MechanismTask::fetchFeature(MechanismDescription::MechanismDescr
iptor* realDescr)
{
    mReal->fetchFeature(realDescr);
}

pReal MechanismTask::getDifference()
{
    mDesired->getValue(mDesiredVal, mCurrentTime);
    mReal->getValue(mRealVal);

    return weightedSquareDifference();
}

void MechanismTask::setScaling(std::vector<pReal>& scaling)
{
    mScaling = scaling;
}

void MechanismTask::setTime(pReal time)
{
    mCurrentTime = time;
}

pReal MechanismTask::weightedSquareDifference()
{
    pReal result = pReal(0);
    pReal value;

    for(int i = 0; i < mDim; i++)
    {
        value = (mDesiredVal[i] - mRealVal[i]);
        result += value*value;
    }

    return pReal(std::sqrt(result));
}
```

```
TaskObjective::TaskObjective(
     MechanismIntegrator* integrator,
     int controlSize,
     pReal itegrationTime
     ) : ObjectiveComponent<pReal>(1, 0, controlSize)
{
     mIntegrator = integrator;
     mIntegrationTime = itegrationTime;
     mControlSize = controlSize;
}

void TaskObjective::addTask(MechanismTask* task, pReal
priorityScale)
{
     task->fetchFeature(mIntegrator->getEndDescriptorP());
     mTasks.push_back(task);
     mPriorityScales.push_back(priorityScale);
}

void TaskObjective::setCurrentTime(pReal time)
{
     this->mCurrentTime = time;
}

void TaskObjective::evaluate(
     dlib::matrix<pReal>* result,
     const dlib::matrix<pReal, 0, 1>* arg,
     const dlib::matrix<pReal, 0, 1>* param
     ) const
{
     std::vector<pReal> u;
     u.resize(mControlSize + 3);

     pReal objectiveValue = pReal(0);

     for(int i = 0; i < mControlSize; i++)
     {
          u[i + 3] = 1.0*(*param)(i, 0);
     }

     mIntegrator->step(u, mIntegrationTime);

     objectiveValue = pReal(0);
     for(int i = 0; i < mTasks.size(); i++)
     {
          mTasks[i]->setTime(mCurrentTime);
          pReal taskObj = mTasks[i]->getDifference();
          objectiveValue += mPriorityScales[i]*taskObj*taskObj;
     }
```

```
     (*result)(0, 0) = objectiveValue;
}

TaskOptimizerWrapper::TaskOptimizerWrapper(LevenbergOptimizer<pR
eal>* optimizer, pReal scaling)
{
     mOptimizer = optimizer;
     mScaling = scaling;
}

pReal TaskOptimizerWrapper::solve(std::vector<pReal>& control)
{
     int controlDim = control.size();
     dlib::matrix<pReal, 0, 1> x = dlib::matrix<pReal, 0, 1>(1,
1);
     dlib::matrix<pReal, 0, 1> params = dlib::matrix<pReal, 0,
1>(controlDim - 3, 1);

     for(int i = 3; i < controlDim; i++)
     {
          params(i - 3, 0) = mScaling*control[i];
     }

     pReal objectiveVal = mOptimizer->solve(x, params, 40.0);

     for(int i = 3; i < controlDim; i++)
     {
          control[i] = params(i - 3, 0) / mScaling;
     }

     return objectiveVal;
}
```

RobotFeatureDescriptor.h:

```
#ifndef _ROBOT_FEATURE_DESCRIPTOR_H_
#define _ROBOT_FEATURE_DESCRIPTOR_H_

#include "ProjectTypes.h"

#include <vector>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <stdlib.h>

class TimeFeatureTable
{
public:
```

```
        typedef std::pair<pReal, std::vector<pReal>> Data;

        class DataCompare
        {
        public:
            bool operator()(const Data& lhs, const Data& rhs);
            bool operator()(const Data& lhs, const
Data::first_type& k) const;
            bool operator()(const Data::first_type& k, const Data&
rhs) const;
        private:
            bool keyLess(const Data::first_type& k1, const
Data::first_type& k2) const;
        };

        TimeFeatureTable();
        TimeFeatureTable(const TimeFeatureTable&);
        ~TimeFeatureTable();
        void operator=(const TimeFeatureTable& rhs);


        void addData(Data& d);
        void rearrange();
        void getData(pReal key, std::vector<pReal>& data);

        bool empty();
        std::pair<pReal, pReal> getTimeInterval();

    private:
        std::vector<Data> mData;
        std::vector<Data>::iterator mLastGivenOut;
    };

    class FeatureDescriptor
    {
    public:
        FeatureDescriptor();
        virtual int valueLength() = 0;

        void getValue(std::vector<pReal>& dest, pReal time);
        void setOrigin(pReal x, pReal y);
        void setStartTime(pReal time);

    protected:
        virtual void getRawValue(std::vector<pReal>& dest, pReal
time) = 0;

    private:
        pReal xOffset;
        pReal yOffset;
```

```
        pReal timeOffset;
};

class ExternalFeature : public FeatureDescriptor
{
public:
        ExternalFeature(int valLength);
        void getRawValue(std::vector<pReal>& dest, pReal time);
        int valueLength();

        std::pair<pReal, pReal> getTimeInterval();

protected:
        //vector sorted by time
        TimeFeatureTable mValues;
        int mLength;
};

class FeatureDerivative : public FeatureDescriptor
{
public:
        FeatureDerivative(FeatureDescriptor* f, pReal dt);
        void getRawValue(std::vector<pReal>& dest, pReal time);
        int valueLength();

private:
        FeatureDescriptor* mIntegralFeature;
        pReal dt;
};

class HopperLoader
{
public:
        HopperLoader(std::string filepath);
        void load();

        void getData(TimeFeatureTable& table, int index);

private:
        std::string mFilepath;

        TimeFeatureTable* mData;
};

class HopperFeature : public ExternalFeature
{
public:
        HopperFeature(HopperLoader* data, int index);
};
```

```
#endif
```

RobotFeatureDescriptor.cpp

```
#include "RobotFeatureDescriptor.h"

bool TimeFeatureTable::DataCompare::operator()(const Data& lhs,
const Data& rhs)
{
     return keyLess(lhs.first, rhs.first);
}

bool TimeFeatureTable::DataCompare::operator()(const Data& lhs,
const Data::first_type& k) const
{
     return keyLess(lhs.first, k);
}

bool TimeFeatureTable::DataCompare::operator()(const
Data::first_type& k, const Data& rhs) const
{
     return keyLess(k, rhs.first);
}

bool TimeFeatureTable::DataCompare::keyLess(const
Data::first_type& k1, const Data::first_type& k2) const
{
     return k1 < k2;
}

TimeFeatureTable::TimeFeatureTable()
{

}

TimeFeatureTable::TimeFeatureTable(const TimeFeatureTable& rhs)
{
     this->mData = rhs.mData;
     this->mLastGivenOut = mData.end();
}

TimeFeatureTable::~TimeFeatureTable()
{
     //do nothing
}

void TimeFeatureTable::operator=(const TimeFeatureTable& rhs)
{
     this->mData = rhs.mData;
     this->mLastGivenOut = mData.end();
}
```

```
void TimeFeatureTable::addData(Data& d)
{
     mData.push_back(d);
     mLastGivenOut = mData.end();
}

void TimeFeatureTable::rearrange()
{
     std::stable_sort(
          mData.begin(),
          mData.end(),
          TimeFeatureTable::DataCompare()
          );
     mLastGivenOut = mData.end();
}

void TimeFeatureTable::getData(pReal key, std::vector<pReal>&
data)
{
     if(mLastGivenOut != mData.end())
          if ((mLastGivenOut + 1)->first == key)
          {
               mLastGivenOut++;
               data = mLastGivenOut->second;
               return;
          }

     std::vector<Data>::iterator range;

     range = lower_bound(
          mData.begin(),
          mData.end(),
          key,
          TimeFeatureTable::DataCompare()
          );

     if(range != mData.end())
     {
          mLastGivenOut = range;
          data = mLastGivenOut->second;
          return;
     }
     else throw std::exception("time isn't in range");
}

bool TimeFeatureTable::empty()
{
     return mData.empty();
}
```

```
std::pair<pReal, pReal> TimeFeatureTable::getTimeInterval()
{
     if(mData.empty())
          return std::make_pair(pReal(-1.0), pReal(-1.0));

     else
          return std::make_pair(mData.front().first,
mData.back().first);
}

FeatureDescriptor::FeatureDescriptor()
{
     timeOffset = pReal(0);
     xOffset = pReal(0);
     yOffset = pReal(0);
}

void FeatureDescriptor::getValue(std::vector<pReal>& dest, pReal
time)
{
     this->getRawValue(dest, time - timeOffset);

     dest[0] = dest[0] + xOffset;
     dest[1] = dest[1] + yOffset;
}

void FeatureDescriptor::setStartTime(pReal time)
{
     timeOffset = time;
}

void FeatureDescriptor::setOrigin(pReal x, pReal y)
{
     xOffset = x;
     yOffset = y;
}

ExternalFeature::ExternalFeature(int valLength)
{
     mLength = valLength;
}

void ExternalFeature::getRawValue(std::vector<pReal>& dest,
pReal time)
{
     mValues.getData(time, dest);
}

int ExternalFeature::valueLength()
{
     return mLength;
```

```
}

std::pair<pReal, pReal> ExternalFeature::getTimeInterval()
{
	return mValues.getTimeInterval();
}

HopperLoader::HopperLoader(std::string filepath)
{
	mFilepath = filepath;
	mData = new TimeFeatureTable[4];
}

void HopperLoader::load()
{
	std::ifstream F;
	F.open(mFilepath, std::ios::in);

	if(!F) throw std::exception("file not opened");

	int numOfVals = -1;
	int length = -1;
	F >> numOfVals;

	F >> length;

	std::vector<std::vector<pReal>> values;
	values.resize(4);
	for (int i = 0; i < 4; i++)
		values[i].resize(2);

	pReal time;

	for(int i = 0; i < numOfVals; i++)
	{
		std::string s;

		for(int k = 0; k < length; k++)
		{
			if(k == 0)
				F >> time;
			else
				F >> values[(k - 1)/2][(k - 1)%2];
		}
		//F >> s;

		for(int k = 0; k < 4; k++)
			mData[k].addData(std::make_pair(time, values[k]));
	}

	for(int i = 0; i < 4; i++)
```

```
            mData[i].rearrange();

    F.close();
}

void HopperLoader::getData(TimeFeatureTable& table, int index)
{
    table = mData[index];
}

HopperFeature::HopperFeature(HopperLoader* data, int index) :
ExternalFeature(2)
{
    data->getData(mValues, index);
}

FeatureDerivative::FeatureDerivative(FeatureDescriptor* f, pReal
dt)
{
    mIntegralFeature = f;
    this->dt = dt;
}

void FeatureDerivative::getRawValue(std::vector<pReal>& dest,
pReal time)
{
    std::vector<pReal> futValue;
    futValue.resize(mIntegralFeature->valueLength());

    mIntegralFeature->getValue(futValue, time + dt);
    mIntegralFeature->getValue(dest, time);

    for(int i = 0; i < dest.size(); i++)
        dest[i] = (futValue[i] - dest[i])/dt;
}

int FeatureDerivative::valueLength()
{
    return mIntegralFeature->valueLength();
}
```

RobotState.h:

```
#ifndef _ROBOT_STATE_H_
#define _ROBOT_STATE_H_

#include "RobotDescriptor.h"

#include "ProjectTypes.h"
#include "State2D.h"
```

```
#include <map>

namespace MechanismDescription
{
	struct JointNode;
	struct BodyNode;

	class MechanismTreeVisitor
	{
	public:
		void visitJointNode(JointNode* node);
		void visitBodyNode(BodyNode* node);

		virtual void processJointNode(JointNode* node) = 0;
		virtual void processBodyNode(BodyNode* node) = 0;
		virtual void incrementLevel();
		virtual void decrementLevel();
	};

	class MechNode
	{
	public:
		virtual std::vector<std::pair<StlMatrix::Matrix<pReal,
6, 1>, int>> getFreedomAxes() = 0;
		virtual std::string type() = 0;
		virtual std::string name() = 0;

		//state info
		Pos2D<pReal> pos;
		Pos2D<pReal> vel;

		pReal torque;
	};

	struct MechanismGeneralizedCoords
	{
		typedef std::pair<MechNode*, int> CoordHandler;

		std::vector<CoordHandler> coordArray;

		/*gets generalized coords indexes in coords vector
given mechNode,
		* associated with these gen coords, in OxOyOzXYZ order
		*/
		void getCoordIndexes(MechNode* node,
std::vector<std::pair<int, int>>& gcIndexes);
	};

	class BodyNode : public MechNode
	{
	public:
```

```
            BodyNode(Body* b, Joint* pred_joint);
            Joint* getJoint(int i);

            //mech node mathod impls
            std::vector<std::pair<StlMatrix::Matrix<pReal, 6, 1>,
int>> getFreedomAxes();
            std::string type();
            std::string name();

            //tree info
            Body* body;
            Joint* pred_joint;
            std::vector<JointNode*> childs;

            //freedom info
            std::vector<std::pair<StlMatrix::Matrix<pReal, 6, 1>,
int>> freedomAxes;

            void accept(MechanismTreeVisitor* visitor);
        };

        class JointNode : public MechNode
        {
        public:
            JointNode(Joint* joint, BodyNode* pred, BodyNode*
succ);

            //mech node method impls
            std::vector<std::pair<StlMatrix::Matrix<pReal, 6, 1>,
int>> getFreedomAxes();
            std::string type();
            std::string name();

            //tree info
            BodyNode* pred;
            BodyNode* succ;
            Joint* joint;

            //freedom info
            std::vector<std::pair<StlMatrix::Matrix<pReal, 6, 1>,
int>> freedomAxes;

            void accept(MechanismTreeVisitor* visitor);
        };

        class GenCoordsVisitor : public MechanismTreeVisitor
        {
        public:
            GenCoordsVisitor();

            //access
```

```
          MechanismGeneralizedCoords getCoords();

          //tree visitor method impls
          void processJointNode(JointNode* node);
          void processBodyNode(BodyNode* node);

          //helpers
          void print();

     private:
          void processMechNode(MechNode* node);

     private:
          MechanismGeneralizedCoords mCoords;
     };

     class MechanismTree
     {
     public:
          MechanismTree(int floatingBaseDegrees) : bodies(),
joints(), bodies_map(),
               joints_map(), joints_pred_map(),
flBaseDegrees(floatingBaseDegrees)
          {

          }

          ~MechanismTree(){}

          std::vector<std::pair<StlMatrix::Matrix<pReal, 6, 1>,
int>> getFlBaseMatrixCols()
          {
               std::vector<StlMatrix::Matrix<pReal, 6, 1>>
eyeCols =

StlMatrix::MatrixHelper::spartialIdentity<pReal>().splitColumns(
);

               std::vector<std::pair<StlMatrix::Matrix<pReal, 6,
1>, int>> result;

               if(flBaseDegrees == 3)
               {
                    result.push_back(std::make_pair(eyeCols[2],
2)); //for Oz rot
                    result.push_back(std::make_pair(eyeCols[3],
3)); //for Ox transl
                    result.push_back(std::make_pair(eyeCols[4],
4)); //for Oy transl
               }
               else if(flBaseDegrees == 6)
```

```
                {
                        result.push_back(std::make_pair(eyeCols[0],
0));
                        result.push_back(std::make_pair(eyeCols[1],
1)); //for Oy rot
                        result.push_back(std::make_pair(eyeCols[2],
2));
                        result.push_back(std::make_pair(eyeCols[3],
3)); //for Ox transl
                        result.push_back(std::make_pair(eyeCols[4],
4));
                        result.push_back(std::make_pair(eyeCols[5],
5)); //for Oz transl
                }

                return result;
        }

        StlMatrix::Matrix<pReal, 6, 6> getFlbaseMatrix()
        {
                //assuming that zeroing row equals zeroing a row
in an eye matrix

                StlMatrix::Matrix<pReal, 6, 6> result =

StlMatrix::MatrixHelper::spartialIdentity<pReal>();

                if(flBaseDegrees == 3)
                {
                        StlMatrix::Matrix<pReal, 6, 1> zeroRow =

StlMatrix::MatrixHelper::zeroMatrix<pReal, 6, 1>();

                        for(int i = 0; i < 6; i++)
                        {
                                result[i] = zeroRow[0];
                        }
                }

                return result;
        }

        std::vector<Body>& getAllBodies()
        {
                return this->bodies;
        }

        std::vector<Joint>& getAllJoints()
        {
                return this->joints;
        }
```

```
        BodyNode* getBodyNode(std::string bodyName)
        {
             return this->bodyNodesMap[bodyName];
        }

        JointNode* getJointNode(std::string jointName)
        {
             if(this->rootJoint->joint->getName() == jointName)
                  return rootJoint;
             return this->jointNodesMap[jointName];
        }

        void accept(MechanismTreeVisitor* visitor)
        {
             this->rootJoint->accept(visitor);
        }

        void setNodeState(std::string name, Pos2D<pReal> pos,
Pos2D<pReal> vel);
        void getNodeState(std::string name, Pos2D<pReal>& pos,
Pos2D<pReal>& vel);

        void addBody(Body* b);
        void addJoint(Joint* j);

        void buildTree();

        BodyNode* rootNode;
        JointNode* rootJoint;

    private:

        const int flBaseDegrees;

        void findRoot();
        void loadTree();
        void addFlBaseJoint();
        void loadSubtree(BodyNode* root);

        //raw content
        std::vector<Body> bodies;
        std::vector<Joint> joints;

        //tree helpers
        std::map<std::string, Body*> bodies_map;
        std::map<std::string, Joint*> joints_map;
        std::multimap<std::string, Joint*> joints_pred_map;

        //tree access helpers
        std::map<std::string, BodyNode*> bodyNodesMap;
```

```
        std::map<std::string, JointNode*> jointNodesMap;
    };

    class MechanismDescriptor
    {
    public:
        MechanismDescriptor(int fbdegrees);
        ~MechanismDescriptor();

        //access
        MechanismTree* getTree() { return &tree; }
        MechanismGeneralizedCoords getCoords() { return coords;
}

        void update()
        {
            tree.buildTree();

            GenCoordsVisitor coordBuilder;
            this->accept(&coordBuilder);
            coords = coordBuilder.getCoords();
        }

        void addBody(Body* body)
        {
            tree.addBody(body);

            update();
        }

        void setObjectState(std::string name, Pos2D<pReal> pos,
Pos2D<pReal> vel)
        {
            tree.setNodeState(name, pos, vel);
        }

        void addJoint(Joint* joint)
        {
            tree.addJoint(joint);

            update();
        }

        void updateCoords()
        {
            GenCoordsVisitor builder;
            this->accept(&builder);
            this->coords = builder.getCoords();
        }

        void printTree()
```

```
            {
                printTree(this->tree.rootNode, "root", "");
            }

            void printTree(BodyNode* node, std::string jointName,
std::string ident)
            {
                std::cout << ident << "(" << jointName << ")" << "
body " << node->body->getName() << std::endl;
                for(int i = 0; i < node->childs.size(); i++)
                {
                    printTree(node->childs[i]->succ, node-
>childs[i]->joint->name, "\t" + ident);
                }
            }

            BodyNode* getBodyNode(std::string bodyName)
            {
                return tree.getBodyNode(bodyName);
            }

            JointNode* getJointNode(std::string jointName)
            {
                return tree.getJointNode(jointName);
            }

            void getNodeState(std::string name,  Pos2D<pReal>& pos,
Pos2D<pReal>& vel)
            {
                tree.getNodeState(name, pos, vel);
            }

            void accept(MechanismTreeVisitor* visitor)
            {
                this->tree.accept(visitor);
            }

        private:
            MechanismTree tree;
            MechanismGeneralizedCoords coords;
        };

        class MechanismTreePrinter : public MechanismTreeVisitor
        {
        public:
            MechanismTreePrinter();

            void processJointNode(JointNode* node);
            void processBodyNode(BodyNode* node);
            void incrementLevel();
            void decrementLevel();
```

```
    private:
        std::string currJointName;
        int ident;
    };

    class MechanismStateHelper
    {
    public:
        typedef Pos2D<pReal> Transform;
        typedef Pos2D<pReal> Position;
        typedef Pos2D<pReal> Velocity;

        static Transform mergeTransforms(Transform bToBaseT,
Transform anchorT, Transform jointT);
        static Transform ortVelFromPos(Transform T, pReal
rate);
        static Transform rotateTransform(Transform T, pReal
angle);

        static pReal atan2(pReal x, pReal y);
        static pReal getTangentAngle(Position var, Position
base);
        static pReal getTangentAngleVel(Position varPos,
Position basePos, Velocity varVel, Velocity baseVel);
    };

    class MechanismStateVisitor : public MechanismTreeVisitor
    {
    public:
        MechanismStateVisitor(MechanismDescriptor* descriptor);

        void setState(std::vector<pReal> q, std::vector<pReal>
dq);

        //visitor methods impl
        void processJointNode(JointNode* node);
        void processBodyNode(BodyNode* node);
        void incrementLevel();
        void decrementLevel();

    private:
        std::vector<MechanismStateHelper::Transform>
mBtoBtransformStack;
        std::vector<MechanismStateHelper::Velocity>
mPrevBVelStack;

        std::vector<pReal> q;
        std::vector<pReal> dq;
        MechanismGeneralizedCoords mCoords;
```

```
    };

    //given gen state deduces per-object state
    class MechanismStateResolver
    {
    public:
        MechanismStateResolver(MechanismDescriptor*
descriptor);

        //access
        void setGeneralizedState(std::vector<pReal> q,
std::vector<pReal> dq);

        void getObjectState(std::string name, Pos2D<pReal>&
pos, Pos2D<pReal>& vel);

    private:
        MechanismDescriptor* mDescriptor;
    };

    class MechanismControlApplier : public MechanismTreeVisitor
    {
    public:
        MechanismControlApplier(MechanismDescriptor*
descriptor, pReal scale);

        void applyControl(std::vector<pReal>& u);

        //visitor methods impl
        void processJointNode(JointNode* node);
        void processBodyNode(BodyNode* node);
        void incrementLevel();
        void decrementLevel();

    private:
        MechanismDescriptor* mDescriptor;
        pReal mScale;
        std::vector<pReal> u;
    };

    class MechanismGenCoordsVisitor : public
MechanismTreeVisitor
    {
    public:
        MechanismGenCoordsVisitor(MechanismDescriptor*
descriptor, std::vector<pReal>& q, std::vector<pReal>& dq);

        //visitor methods impl
        void processJointNode(JointNode* node);
        void processBodyNode(BodyNode* node);
```

```
            void incrementLevel();
            void decrementLevel();

        private:

            std::vector<pReal>& q;
            std::vector<pReal>& dq;
            MechanismGeneralizedCoords mCoords;

            JointNode* mLastJointNode;
        };

        class MechanismGenCoordsResolver
        {
        public:
            MechanismGenCoordsResolver(MechanismDescriptor*
descriptor);

            void setState(
                std::string name,
                pReal x, pReal y, pReal theta,
                pReal dx, pReal dy, pReal dTheta
                );

            void resolve(std::vector<pReal>& result_q,
std::vector<pReal>& result_dq);

        private:
            MechanismDescriptor* mDescriptor;
        };

        class MechanismCloner : public MechanismTreeVisitor
        {
        public:
            MechanismCloner(MechanismDescriptor* descriptor);

            MechanismDescriptor* getClone();

            //visitor methods impl
            void processJointNode(JointNode* node);
            void processBodyNode(BodyNode* node);

            void incrementLevel();
            void decrementLevel();

        private:
            MechanismDescriptor* mDescriptor;
            MechanismDescriptor* mClone;
        };

        class MechanismCoppier : public MechanismTreeVisitor
```

```
{
public:
     MechanismCoppier(MechanismDescriptor* descriptor);

     void copy(MechanismDescriptor* to);

     //visitor methods impl
     void processJointNode(JointNode* node);
     void processBodyNode(BodyNode* node);

     void incrementLevel();
     void decrementLevel();

private:
     MechanismDescriptor* mDescriptor;
     MechanismDescriptor* mClone;
};

class MechanismFeature
{
public:
     virtual void fetchFeature(MechanismDescriptor* descr) =
0;
     virtual void getValue(std::vector<pReal>& dest) = 0;
     virtual int valueLength() = 0;
};

class MechanismBVFeature : public MechanismFeature
{
     /*
      * Body-Velocity feature. Returns velocity of the body.
      */
public:
     MechanismBVFeature(std::string name);

     void fetchFeature(MechanismDescriptor* descr);
     void getValue(std::vector<pReal>& dest);
     int valueLength();

private:
     std::string mNodeName;
     BodyNode* mNode;
};

class MechanismBPFeature : public MechanismFeature
{
     /*
      * Body-Position feature. Returns position of the body.
      */
public:
     MechanismBPFeature(std::string name);
```

```
            void fetchFeature(MechanismDescriptor* descr);
            void getValue(std::vector<pReal>& dest);
            int valueLength();

      private:
            std::string mNodeName;
            BodyNode* mNode;
      };

      class MechanismInternalPointPos : public MechanismFeature
      {
      public:
            MechanismInternalPointPos(std::string name, pReal intX,
pReal intY);
            void fetchFeature(MechanismDescriptor* descr);
            void getValue(std::vector<pReal>& dest);
            int valueLength();

      private:
            std::string mNodeName;
            Pos2D<pReal> mInternalPos;
            BodyNode* mNode;
      };

      class MechanismInternalPointVel : public MechanismFeature
      {
      public:
            MechanismInternalPointVel(std::string name, pReal intX,
pReal intY);
            void fetchFeature(MechanismDescriptor* descr);
            void getValue(std::vector<pReal>& dest);
            int valueLength();

      private:
            std::string mNodeName;
            Pos2D<pReal> mInternalPos;
            BodyNode* mNode;
      };

      class MechanismCOMPos : public MechanismFeature, public
MechanismTreeVisitor
      {
      public:
            MechanismCOMPos();

            void fetchFeature(MechanismDescriptor* descr);
            void getValue(std::vector<pReal>& dest);
            int valueLength();

            //visitor methods impl
```

```
            void processJointNode(JointNode* node);
            void processBodyNode(BodyNode* node);

            void incrementLevel();
            void decrementLevel();

        private:
            MechanismDescriptor* mDescriptor;
            Pos2D<pReal> mCenter;
            pReal mMass;
        };

        class MechanismCOMVel : public MechanismFeature, public
MechanismTreeVisitor
        {
        public:
            MechanismCOMVel();

            void fetchFeature(MechanismDescriptor* descr);
            void getValue(std::vector<pReal>& dest);
            int valueLength();

            //visitor methods impl
            void processJointNode(JointNode* node);
            void processBodyNode(BodyNode* node);

            void incrementLevel();
            void decrementLevel();

        private:
            MechanismDescriptor* mDescriptor;
            Pos2D<pReal> mCenter;
            pReal mMass;
        };
}

#endif
```