Programming Exercises for the Roger-the-Crab Mobile Manipulation Simulator

Spring 2019

Contents

Programming Exercises for the Roger-the-Crab Mobile Manipulation Simulator

1 Computing Environment

The *Roger-the-Crab* simulator is written in C and uses X windows. It should run roughly "out of the box" on OSX and Linux machines. On Windows machines, however, we strongly recommend that you install *Virtual Box* and *Ubuntu* Linux.

1.1 Running the Simulator on Windows

To get the simulator working on a Windows platform, download and install Virtual Box 4.3.12

https://www.virtualbox.org/wiki/Download_*Old_Builds*_4_3.

You will also need an Ubuntu 18.04 iso (http://releases.ubuntu.com/precise/). Other older versions of this OS should work too, but this particular configuration has been tested and will work.

To create a new virtual machine:

- Click "New Virtual Machine"
- Name the OS "Ubuntu," the OS Type is "Linux," the Version is "Ubuntu (x Bit)," where is "x" is 32 or 64 depending on what you downloaded.
- Set memory size as desired based on system hardware, but no less than 512 MB.
- Create a new Virtual hard drive (VDI) with a size at least 8 GB.
- Once that is completed, click "settings" and select the storage tab. In the "attributes" pane, there is a small CD button next to the CD/DVD Drive drop down. Press this button and then select "choose a virtual CD/DVD disk file..." and locate the ubuntu iso that you downloaded.
- All done! hit start and follow the Ubuntu first time installation instructions.

1.2 Compiling the Simulator

Download the simulator into your working directory from the course website.

compiling the simulator:

1. use the command "unzip roger-2019.tar" to unpack your simulator code.

2. follow the directions in the README

Everytime you change the code in your RogerProjects directory, you must run the Makefile in the RogerProjects directory, which re-compiles your code, re-links to the dynamics.a and SocketComm.a libraries, and produce the new "./roger" executable.

Confine your project work to the files distributed in the RogerProjects subdirectories. It is important that you do not change the directory structure, rename, or add code in files other than those that already exist because the makefile will not include them. Early projects provide important prerequisites for subsequent projects so do them in order and spend some time getting things right before moving on.

1.3 Frequently Asked Questions

- If you run into issues using Ubuntu during compilation regarding missing X11 headers, run the following commands in the terminal:
  - sudo apt-get update
  - sudo apt-get install libxt-dev libxaw7-dev

  This will download the appropriate linux headers for X11.
- Some OSX users may encounter the same messages for missing X11 headers, and will need to install XQuartz (http://xquartz.macosforge.org/landing/). The X11 window system is not included on all versions of OSX. If you are getting errors related to missing X11 header files, this should solve the problem.

2 The Robot and Programming Interface

2.1 Kinematic Definitions

Figure 1 is a kinematic description of "Roger-the-Crab." All geometrical parameters used to define Roger's body geometry are defined as compile-time constants in file include/roger.h. Roger has nine degrees of freedom: 2 "pan" degrees of freedom in the eyes (θ_1, θ_2); 2 degrees of freedom for the left arm (θ_3, θ_4) and the right arm (θ_5, θ_6); and 3 degrees of freedom for mobility $(x, y, \theta)_0$. The three degree of freedom base is controlled using two wheels subject to a nonholonomic constraint (see Chapter 4 of the text). Each configuration variable describes the position of a revolute joint and, except for the wheels, rotates about the world frame $\hat{\boldsymbol{z}}$ axis. Therefore, positive rotations are counter clockwise. Roger's wheels and joints in the arm are free to rotate continuously. His eyes, however, are limited to $-\pi/2 \leq \theta_1, \theta_2 \leq \pi/2$.

The robot uses joint angle sensors to measure the configuration of the eyes and arms. Access to wheel angles is not supported in the robot interface. Instead, the cumulative history of wheel

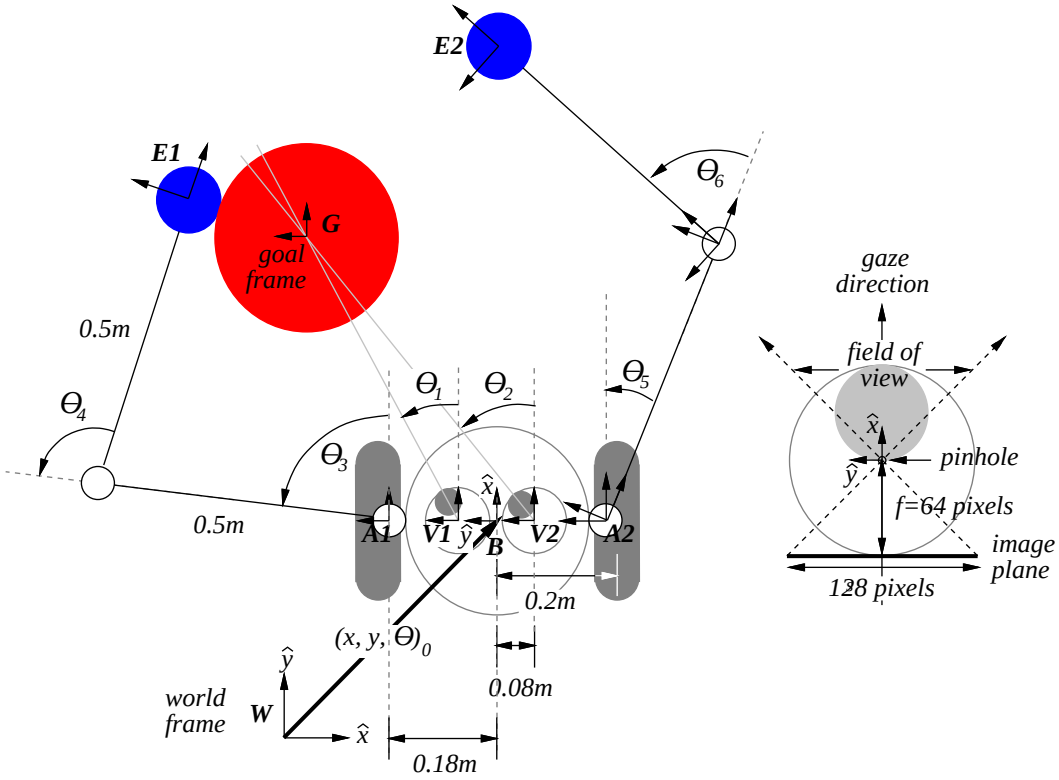


Figure 1: The kinematic definitions and intermediate coordinate frames used to define Roger—a mobile manipulator in the $x - y$ plane.

rotations is integrated to estimate position (x, y, θ) and velocity $(\dot{x}, \dot{y}, \dot{\theta})$ via an embedded odometry process running on the robot. Setpoints are specified for (x, y, θ) as well, however, the mobile base is nonholonomic and, therefore, only provides two instantaneously controllable degrees of freedom for mobility.

Roger's eyes are pinhole RGB cameras with a focal length of 64 pixels that produce a one dimensional image 128 pixels wide. Each eye, therefore, has a field of view of ± 45 degrees directed into the world by motor actions that pan the eyes and/or the body. In addition, Roger has a tactile sensor at the endpoint of each arm that returns a single force vector in the $x - y$ plane.

In the course of several projects, you will program layers of control code that implement integrated robot behavior.

2.2 Simulator I/O

All user-defined control procedures are invoked with arguments "roger" and "time." The "time" variable is a double precision floating point number that contains the simulated clock value in

seconds. Variable "roger" is type "Robot" defined in terms of "Robot_interface" and "Map" data structures (Figure 2). The "Robot_interface" data structure passes sensor data from the simulator to your control code and allows you to write torques back to the simulator. Sensors are READ-only and motor torques are WRITE-only. Joint angle sensors measure robot state consisting of positions and velocities for every degree of freedom in the robot. Every simulated millisecond, your control code will READ state feedback information, transform it into motor torques for every degree of freedom, and WRITE these torques back into the interface.

In addition to this basic I/O structure, the *Robot_interface* data structure includes three different maps of Roger's workspace that can accumulate information over many observations. The *Map* data type is defined in the bottom of Figure 2. It reserves memory for a "configuration space" map for each arm and another *world_map* to represent the Cartesian floorplan in which Roger moves. The maps are called *occupancy grids* that resolve the continuous x, y floorplan into a two-dimensional array of discrete "bins." Bins in the occupancy grid are labeled FREESPACE for the robot by default, but these labels can be changed to OBSTACLE or GOAL for use in path planning and control applications.

User Inputs: There are two primary mechanisms for users to introduce data into the simulator without re-compiling the code.

Procedure *enter_params()* (in project1/Motorunits.c) is called when the "Enter Params" button in the GUI is pushed. You can write code in this procedure to print a prompt and scan input values from the command line. This may be useful for tuning gains, for introducing configuration space goals, for conducting experiments, or for anything else you may need.

Each *User Project* control mode (below) directs the "Enter Params" button event to a specialized *projectX_enter_params()* (in file projectX/projectX.c). This allows for customized input methods for every user project.

GUI interface - mouse events in different regions of the simulator canvas designate spatial data that can be used in your control programs. The simulator is configured with several I/O **modes**:

- **Input: Joint angles** - left-clicking in a two dimensional configuration space (q_1, q_2) region of the canvas defines a joint angle setpoint for the shoulder and the elbow of the corresponding arm.

  Right-clicking in the left/right configuration spaces assigns the designated q_1 coordinate to setpoints for the left/right eyes. Note that joint angles for the eyes are limited to $\pm\pi/2$.
- **Input: Base goal** - clicking in the GUI's Cartesian map defines an (x, y) setpoint for Roger's mobile base.
- **Input: Arm goals** - left-clicking in the GUI's Cartesian map defines (x, y) setpoints for the left arm and right-clicks define setpoints for the right arm. These goals must be transformed into appropriate (q_1, q_2) configuration space setpoints for the arms.

```
typedef struct Robot_interface {
  // SENSORS
  double eye_theta[NEYES];
  double eye_theta_dot[NEYES];
  int image[NEYES][NPIXELS][NPRIMARY_COLORS];
  double arm_theta[NARMS][NJOINTS];
  double arm_theta_dot[NARMS][NJOINTS];
  double ext_force[NARMS][2];      /* (fx,fy) force on arm endpoint */
  double base_position[3];         /* x,y,theta */
  double base_velocity[3];

  // MOTORS
  double eye_torque[NEYES];
  double arm_torque[NARMS][NJOINTS];
  double wheel_torque[NWHEELS];

  // TELEOPERATOR
  int button_event;
  double button_reference[2];

  // CONTROL MODE
  int mode;
  Map world_map, arm_map[NARMS];

  // REFERENCE INPUTS
  double base_setpoint[3];             /* ref (x,y,theta)_w of base */
  double arm_setpoint[NARMS][NJOINTS]; /* ref arm joint angles */
  double eyes_setpoint[NEYES];         /* ref eye angles */
} Robot;


typedef struct _map {
   int occupancy_map[NBINS][NBINS];
   double potential_map[NBINS][NBINS];
   int color_map[NBINS][NBINS];
} Map;
```

Figure 2: The *Robot* and *Map* data types define I/O to the simulator and discrete models of the workspaces for use with path planning algorithms.

- **Input: Ball position** - in this mode, a left mouse click drops a stationary red ball at the corresponding position in the Cartesian map. A right click places a polyball [1]
- **Input: Map editor** - a means of editing obstacles and goals in the Cartesian occupancy grid.
- **User Project** - placeholder for a user/project defined input mode. Mouse click events are passed to appropriate function *projectX_configuration_input()* or *projectX_cartesian_input()* (in projectX/projectX.c), respectively.

3 Behavioral Build Files

A project build file contains four procedures for: implementing a control process, introducing user input data, visualizing the geometry of observations and goals, and resetting the system (Figure 3).

3.1 MDP State/Action Structure

Beginning with Project #4 - SearchTrack() (Section 4.4), your projects will use state feedback from the underlying primitives to decide how to compose multiple control actions in a sequence to build behavioral programs. To standardize the representation and use of state over all the projects, every action returns a discrete value that classifies the dynamic response of the action in the current run-time environment:

0 : (NO_REFERENCE) the reference stimulus is not found;

1 : (TRANSIENT) the reference stimulus is detected and the controller in enroute to its equilibrium condition; or

2 : (CONVERGED) the controller has converged to the setpoint.

Assuming feedback of this sort from multiple primitive control actions, a unique integer state can be computed and used to switch actions in a finite state supervisor as illustrated in Figure fig:SAstructure.

4 Programming Projects

Roger's cumulative behavior is constructed layer by layer in a hierarchy of projects that generate movements autonomously in response to sensory observations. Applications are written that perceive goals and that issue setpoints to motor units every millisecond to create patterns of changing

[1] A "polyball" is a rigid body comprised of N elastic spheres positioned at the vertices of a regular polygon of degree N (specifiable by the user).

```
/*************************************************************************/
/* File:        projectX.c                                               */
/* Description: User project #X - empty project directory for project    */
/*              development                                              */
/* Date:        6-01-2018                                                */
/*************************************************************************/
#include <math.h>
#include <stdio.h>
#include "Xkw/Xkw.h"
#include "roger.h"
#include "simulate.h"
#include "control.h"
#include "modes.h"

void projectX_control(roger, time)
Robot* roger;
double time;
{ }

/*************************************************************************/
void projectX_reset(roger)
Robot* roger;
{ }

// prompt for and read user customized input values
void projectX_enter_params()
{
  printf("Project X enter_params called. \n");
}

// function called when the 'visualize' button on the gui is pressed
void projectX_visualize(roger)
Robot* roger;
{ }
```

Figure 3: The common structure of the behavioral build file.

```
double recommended_setpoints[NACTIONS][NDOF]

int state = 0;
int action;

for (action=0;action<NACTIONS; ++action) {
   state += execute_action(action, recommended_setpoints[i]) * POW(3,action);
}

switch(state) {
   case(0):
     \\ choose action[i], 0 <= i < NACTIONS
     submit_setpoints(action[i]); break;
   case(1):
     \\ choose action[i], 0 <= i < NACTIONS
     submit_setpoints(action[i]); break;
     .
     .
     .
   case(3^NACTIONS):
     \\ choose action[i], 0 <= i < NACTIONS
     submit_setpoints(action[i]); break;
}
```

Figure 4: The common State-Action structure for programming, planning, and learning

postures, contact forces, and sensor geometries. These actions act, in turn, as abstract actions in other hierarchical control programs in order to accumulate a library of interdependant, switch-able, and state dependent behavior.

4.1 Project #1 - Basic Motor Units

The first layer of control code consists of closed-loop control processes that move the robot's body to eliminate errors with respect to reference postures. In this project, we will tackle controllers that move the eyes, the base, and the arms. The main parts of Project #1 are summarized in Figure 5 and a partial implementation of these motor controllers can be found in *1-MotorUnits/project1.c*.

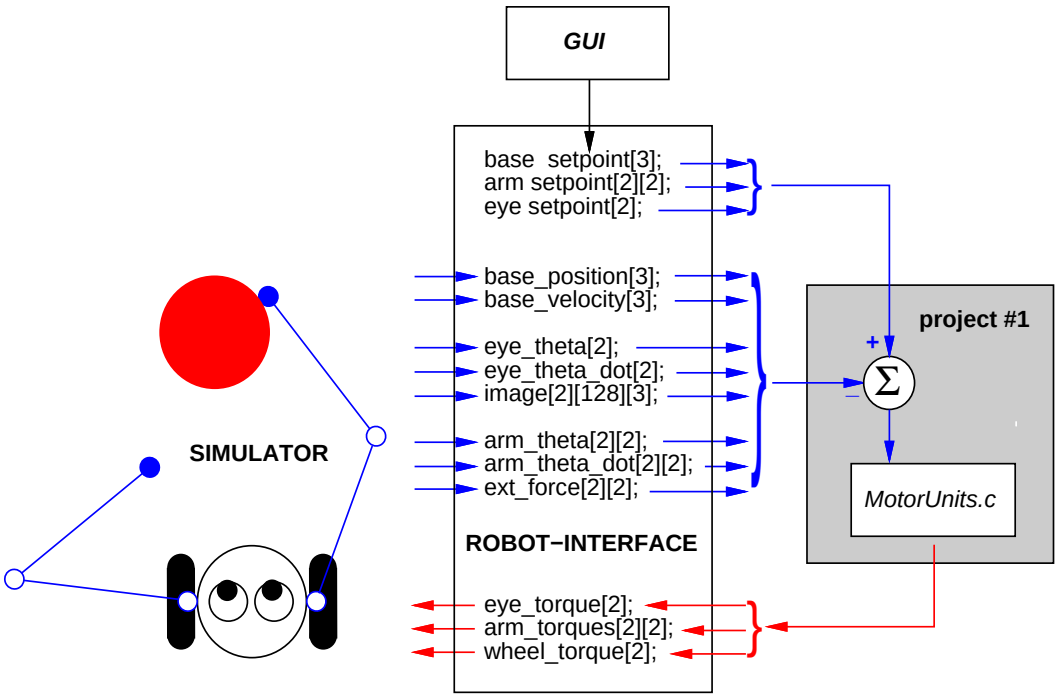


Figure 5: The structure and organization of the code that implements Roger's basic motor functions.

Our approach relies on your implementation of proportional-derivative (PD) feedback controllers (Figure 6) for the robot's independently controllable degrees of freedom. These controllers compute motor torques

$$\tau = -K(\theta_{act} - \theta_{ref}) - B\dot{\theta}_{act}.$$

After you have implemented the controller correctly, their performance still depends on the values you choose for gains K and B. Use your implementations to select good values for these parameters empirically. Adjust K and B using the "Enter Params" button to call the *enter_params()* procedure in Motorunits.c. If you start too small, nothing much will happen. If you guess values that are too

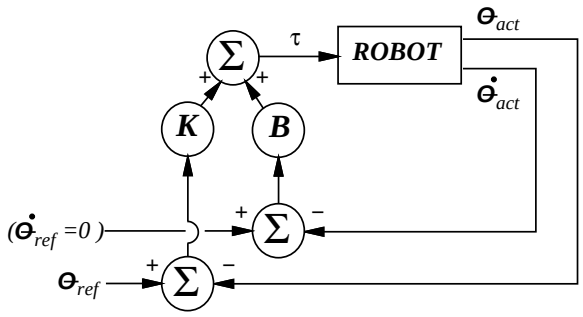


Figure 6: The proportional-derivative feedback control loop for a single rotational degree of freedom.

large, Roger may blow up. Your objective is to find gains that satisfy your sense of aesthetics and performance.

4.1.1 Oculomotor Control

reading: Chapter 3
Sections 2, 3

lecture: Introduction to Control Theory

The current (right/left) eye angle is illustrated by the small green square in the corresponding (right/left) configuration space panels in the GUI. The value of the eye angle is reported in the interval $(-\pi \leq q_1 < +\pi)$.

In the "1-Motor Units" project with input mode set to "Input: Joint angles," right mouse clicks in the (right/left) configuration space panels define references for the corresponding eyes. In the project development GUI, these references are written directly into setpoints in the *Robot-interface* data structure.

Complete the implementation of the PD controller for the eyes in procedure *PDController_eyes()*. Write up a $\leq$ **2 page report** describing how your controller performs. Reports should be sufficiently detailed so that your results can be replicated. The report should include the following parts:

1. describe a procedure for searching for critically damped gains, report gains you end up with and decribe any issues you encounter;
2. hold K fixed and change B to generate plots for under-, over-, and critically-damped responses for the eye; and
3. construct an experiment for comparing the critically damped response of your implementation to the theoretical response. Determine the initial and final posture for the experimental movement. On the same plot, compare the error $(\theta_{ref} - \theta_{act})$ vs. time from your implementation to the analytical result.

4.1.2 Arm Control

reading: Chapter 3, Sections 2, 3 **lecture:** Introduction to Control Theory

The current (right/left) arm configuration is illustrated using a small blue square in the (right/left) configuration space panels of the GUI. The shoulder angle q_1 and elbow angle q_2 maps into these panels in the interval $-\pi \leq q_i < +\pi$.

In the "1-Motor Units" control project with input mode set to "Input: Joint angles," left mouse clicks in the (right/left) configuration space panels define references for both the shoulder and the elbow of the corresponding arm. In the project development GUI, these references are written directly into setpoints in the *Robot-interface* data structure.

Complete the implementation of the PD controller for the arms in procedure *PDController_arms()* that controls both degrees of freedom as if they are independent spring-mass-damper systems. Write up a $\leq$ **2 page report** describing your implementation. Plot the error versus time during the simultaneous movement of both joints from $q_1 = q_2 = 0.0$ to $q_1 = q_2 = \pi/2$. Report the gains you decided to use for these motor units and discuss your experimental result—does your plot depict the theoretical second-order response?

4.1.3 Base Controller

reading: Chapter 4, Sections 2, 3 **lecture:** Homogeneous Transform

In control mode *1-Motor Units* and input mode *Input: Base Goal*, mouse clicks in the Cartesian panel corresponds to (x, y) coordinates in the world frame. The GUI stores this coordinate in the x and y elements of the three dimensional base_setpoint[·] array in the *Robot_interface* data structure. The GUI also computes a default reference heading $\theta = atan2(r_y, r_x)$, where $r_x = (x_{ref} - x_{act})$ and $r_y = (y_{ref} - y_{act})$. It stores θ in the third element of the base_setpoint[·] array (i.e. base_setpoint[2]). If you choose, you may use compile-time constants X, Y, and THETA defined in control.h to make your code more readable, e.g. roger→base_setpoint[X], roger→base_setpoint[Y], and roger→base_setpoint[THETA]).

Write PD controllers for both base rotation and base translation. Complete the implementation of procedure *PDController_base()* (in project1/MotorUnits.c) by writing two controllers:

- *PDBase_translate()* - define a f_x command for a translational acceleration; and
- *PDBase_rotate()* - define a m_z command for creating a desired rotational acceleration.

These controllers implement two spring-damper configurations with different gains that apply a longitudinal force f_x and rotational moment m_z to the robot.

The proposed control design is illustrated in Figure 7. Note that the GUI provides references in

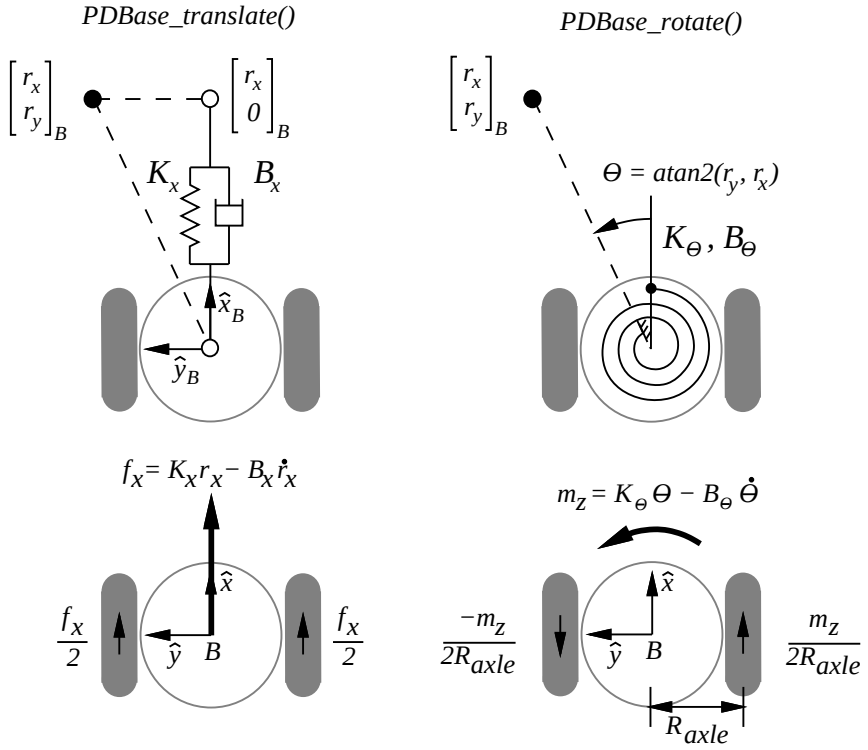


Figure 7: A mobility control framework in terms of independent translational and rotational controllers.

the world coordinate frame, which is fine for the rotation controller, but the translation controller measures its error in base coordinates. Starting with error

$$\boldsymbol{r}_W = \begin{bmatrix} r_x \\ r_y \end{bmatrix}_W = \begin{bmatrix} x_{ref} - x_{act} \\ y_{ref} - y_{act} \end{bmatrix}_W ,$$

the translation error required in Figure 7 is the $\hat{\boldsymbol{x}}$ component of this error vector written in the base coordinate frame. To compute this quantity, we pad $\boldsymbol{r}_W$ to form the equivalent homogeneous position vector (designated with the "bar" $\bar{\boldsymbol{r}}$ notation) and transform it into base coordinates

$$\bar{\boldsymbol{r}}_B = \begin{bmatrix} r_x \\ r_y \\ 0 \\ 1 \end{bmatrix}_B = {}_B T_W \begin{bmatrix} r_x \\ r_y \\ 0 \\ 1 \end{bmatrix}_W$$

and the translation error is just the $\hat{\boldsymbol{x}}$ component of $\bar{\boldsymbol{r}}_B$. The procedure *construct_wTb()* is provided for your use.

The command $[\ f_x\ m_z\ ]^T$ on the base is transformed into wheel torques using the Jacobian describing

the differential steering geometry.

$$\boldsymbol{\tau} = \mathbf{J}^T \mathbf{w}$$

$$\begin{bmatrix} \tau_L \\ \tau_R \end{bmatrix} = \begin{bmatrix} 1/2 & -1/(2R_{axle}) \\ 1/2 & 1/(2R_{axle}) \end{bmatrix} \begin{bmatrix} f_x \\ m_z \end{bmatrix}$$

Complete the code that implements *PDController_base()* and write up a $\leq$ **2 page report** describing your implementation and its performance.

1. Describe your criteria for selecting control gains and your final choices.
2. Plot x, y, and θ errors in the base position as a function of time for a reference:
   (a) 0.75 meters directly in front of the robot (in the $\hat{\boldsymbol{x}}_B$ direction),
   (b) 0.75 meters to the left of the robot (in the $\hat{\boldsymbol{y}}_B$ direction), and
   (c) a pure $\pi/2$ rotation around the $\hat{\boldsymbol{z}}_B$ axis.

4.1.4 Feedforward Inertial Compensation

Simulator Upgrade:

1. Get the `new_RogerSimulator` tar file from the web site and place it in the directory containing `RogerSimulator` and `RogerProjects`.
2. Uncompress the tar file—this will upgrade your simulator.
3. `make clean; make` both the simulator and the projects—after this, you will work exclusively in the `RogerProjects` directory again, as usual.
4. Make a backup copy of your existing `project1-Motorunits` directory so that you can reuse it in subsequent projects if you wish.
5. Modify the working version of `project1-Motorunits/project1.c` as directed below to create a single, eight-dimensional, feedforward compensated version of `control_roger()`.

---

reading: Chapter 5, Section 3 **lecture:** Dynamics

The whole-body computed torque equation can be used to produce a feedforward compensator that linearizes and decouples the behavior of individual degrees of freedom,

$$\widetilde{\boldsymbol{\tau}} = \mathbf{M}\ddot{\boldsymbol{q}} + \mathbf{V} + \mathbf{G} + \mathbf{F}, \tag{1}$$

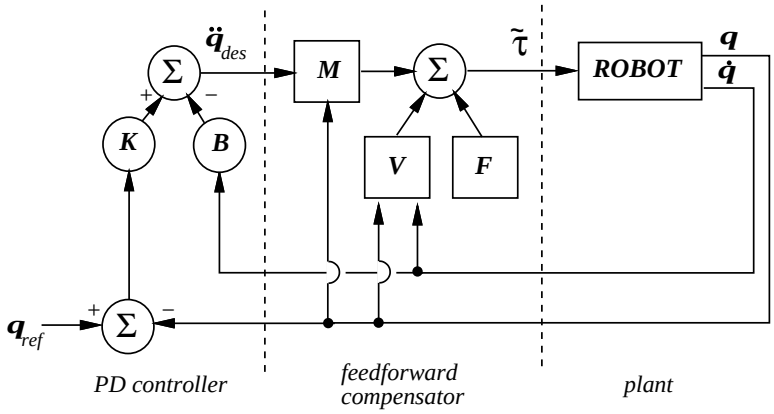


Figure 8: A Linear PD Control Model for a Feedforward Compensated Robot.

where: $\widetilde{\boldsymbol{\tau}} \in \mathbb{R}^8$ is a column vector of force/torque commands for each degree of freedom (DOF) that creates a second-order response to the errors in that DOF and compensates for all inertial perturbations from other DOFs; $\mathbf{M} \in \mathbb{R}^{8\times 8}$ is the generalized inertia matrix; $\ddot{\boldsymbol{q}} \in \mathbb{R}^8$ is a column vector of desired accelerations computed by idealized "unit inertia" PD controllers for each joint[2]; $\mathbf{V} \in \mathbb{R}^8$ is the column vector of Coriolis/centrifugal loads acting on each DOF; $\mathbf{G} \in \mathbb{R}^8$ is a column vector of gravitational loads (there is no gravity for Roger); and $\mathbf{F} \in \mathbb{R}^8$ is a column vector of external forces applied to the robot.

In this programming exercise, you will build a whole body compensator (Figure 8) to account for inertial forces and, thus, use motor effort to suppress the inertial disturbances propagating through the body. Our goal is to *linearize* and *decouple* Roger's eight DOF controlled dynamics and then test the result empirically to verify the theoretical second-order, closed-loop response.

The upgraded simulator code provides procedure

$$void\ get\_dynamics(double\ M[8][8],\ double\ V[8],\ double\ G[8],\ double\ F[8]).$$

to compute the coefficients **M**, **V**, **G**, and **F** of the computed torque equation for the current state of the robot. Use the PD controllers you implemented in previous projects to define the 8×1 vector of desired accelerations. While not required, it may be convenient to refactor your PD control code DOF-by-DOF as shown here.

$$\ddot{\boldsymbol{q}}_{DES} = \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{q}_2 \\ \ddot{q}_3 \\ \ddot{q}_4 \\ \ddot{q}_5 \\ \ddot{q}_6 \\ \ddot{q}_7 \end{bmatrix}_{DES} = \begin{bmatrix} PDBase\_translate(\ ); \\ PDBase\_rotate(\ ); \\ PDLeftEye(\ ); \\ PDRightEye(\ ); \\ PDLeftShoulder(\ ); \\ PDLeftElbow(\ ); \\ PDRightShoulder(\ ); \\ PDRightElbow(\ ); \end{bmatrix}.$$

Determine control gains using the theoretical critically damped result for a second-order spring-"unit m/I"-damper reference model.

[2]They can all be exactly the same PD controller!

$$\begin{bmatrix} \widetilde{\tau}_0 \\ \widetilde{\tau}_1 \\ \widetilde{\tau}_2 \\ \widetilde{\tau}_3 \\ \widetilde{\tau}_4 \\ \widetilde{\tau}_5 \\ \widetilde{\tau}_6 \\ \widetilde{\tau}_7 \end{bmatrix} = \begin{bmatrix} f_x \\ m_z \\ \tau_{\ left\ eye} \\ \tau_{\ right\ eye} \\ \tau_{\ left\ shoulder} \\ \tau_{\ left\ elbow} \\ \tau_{\ right\ shoulder} \\ \tau_{\ right\ elbow} \end{bmatrix}$$

Compute the value of all of the elements of the $\ddot{\boldsymbol{q}}_{DES}$ vector and finish evaluating Equation 1 to arrive at the motor commands $\widetilde{\boldsymbol{\tau}}$.

Most of the generalized torques $\widetilde{\boldsymbol{\tau}}$ can be written directly into the roger data structure as command torques. The exception is the force $\widetilde{\tau}_0$ and moment $\widetilde{\tau}_1$ that are applied to the mobile base—they must be decomposed into left and right wheel torques as before (Figure 7).

Notes: You may use the same gains for every degree of freedom in the compensated system, however, it may be required to tune gains for each type of DOF (base translate, base rotate, eyes, shoulders, and elbows) to arrive at satisfactory relative performance. Be careful about saturating motor torques—this will cause differences between theoretical and observed responses.

Turn in a $\leq$ **2 page report** that includes:

1. a short problem statement/motivation;
2. your choice for the final gains for in light of the feedforward compensator for each type of DOF (eyes, shoulder, elbow);
3. plots that compare the experimental and theoretical responses for simultaneous unit step inputs to base translate, base rotate, a shoulder, an elbow, and an eye; and
4. a short discussion of your results.

4.2 Project #2 - Arm Kinematics

reading: Chapter 4
Section 4.4

lecture: Manipulator Kinematics

In control mode *2-ArmKinematics* and input mode *Input: Arm Goals*, mouse clicks in the GUI correspond to Cartesian $\boldsymbol{r} = [\ x\ y\ ]^T_W$) references that are passed to *inv_arm_kinematics()* in *project2-ArmKinematics/project2.c*.

Complete the implementation of this procedure.

1. First, transform $\boldsymbol{r}_W$ supplied by the GUI into the base coordinate frame. Then, compute inverse kinematic solutions for $\boldsymbol{r}_B$. Existing code in procedure *inv_arm_kinematics()* converts $\boldsymbol{r}_B$ into the appropriate arm frame ($A1$ or $A2$ in Figure 1) based on the input *int limb* variable. Your code determines if the target $\boldsymbol{r}_B$ is reachable, and if so, computes the complete inverse kinematic solution for that limb.

2. In general, the complete solution for Roger's arms will provide two distinct candidate postures of the arm. You must select one of them and write it into the corresponding *arm_setpoint* in *Robot_interface*. Once the setpoints are updated, the appropriate motor unit (*PDController_arms()* in Project 4.1.2) will generate motor torques in the arm that achieve the new equilibrium posture.

3. Finally, if the target point is reachable, procedure *inv_arm_kinematics()* returns 1, else it returns 0.

Report - Write a short report (no more than 2 pages and **no code**) that includes:

1. a description of your criteria for selecting inverse kinematic solutions;

2. a plot of the Cartesian error vs. time for the endpoint frame of the arm in an experimental demonstration of your choice; and

3. a short discussion of your results.

4.3 Project #3 - Visual Tracker

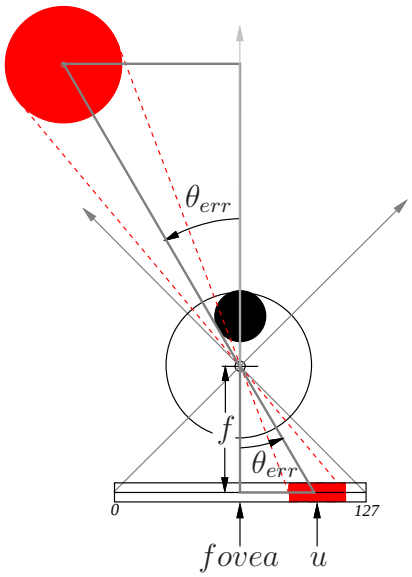
In this project, you are to create a simple feature detector that can be used to recognize and track the red ball when it is present in Roger's environment. To test your result, you will also use this feature detector to measure the *foveation error*—the error between the apparent center of the red ball and the image center. The error is used to update setpoints for the underlying motor units that actuate the eyes to form a closed-loop foveation controller—a controller that suppresses foveation errors.

Implement a simple image processing procedure (`average_red_pixel()`) that checks for the color *red* on both image planes and, if present, estimates the image coordinates, *ul* and *ur*, of the center of the red segments on the left and right images.

```
int average_red_pixel(roger, ul, ur)
{
   if (red is detected in both images) {
      estimate image coordinates ul and ur;
      return(TRUE);
   }
   else return(FALSE);
}
```

For now, we'll get by with a very simple feature in rgb color space where the color of the red ball is represented as $r = 255$, $g = 0$, $b = 0$ and we'll reserve this special color so that it never occurs

elsewhere in Roger's environment. Under these assumptions, the ball is detected whenever at least one pixel for which $r = 255$ is found on the image plane.



To check your implementation, use the distance between *ul* and *ur* and the center of their respective image planes to compute angular foveation errors and use them to update the setpoints for the eyes every millisecond. The underlying motor units for the eyes will take over from there to orient each eye to stationary red balls and to track moving red balls.

Turn in a $\leq$ **2 page report** that includes:

1. a description of your *average_red_pixel()* algorithm;
2. a plot showing the oculomotor error as a function of time for both eyes in response to three consecutive discrete placements of the red ball in the field of view; and
3. a plot showing oculomotor error as a function of time for both eyes in response to a moving red ball that bounces back and forth in front of Roger; and)
4. a short discussion of your results.

4.4 Project #4 - SearchTrack()

This project is the first example of a skill composed of an orchestrated sequence of other actions. Actions in this framework are either *control primitives* or composite *skills*. At any level of abstraction, actions compute setpoints for motor units *and* report their own status to superior (parent) programs.

Every action returns a discrete value that classifies the dynamic response of the action in the current run-time environment:

0: (NO_REFERENCE)—the reference stimulus is not found;

1: (TRANSIENT)—the reference stimulus is detected and the controller has not yet converged; and

2: (CONVERGED)—the controller has converged to the setpoint.

The SearchTrack($\cdot$) skill requires the integration of two primitive control actions.

1. Search($\cdot$) samples reference cycloptic gaze directions in world coordinates from a probability distribution describing where the red ball might be found and submits corresponding setpoints

to the underlying motor units. For this project, reference gaze directions are uniformly distributed over the interval from $-\pi$ to $+\pi$ in world coordinates.

Use the procedure *sample_gaze_direction()* that is provided to you in *sampling.c* and complete procedure SEARCH($\cdot$) to:

(a) define setpoints for the base;

(b) choose complementary setpoints for the eyes; and

(c) generate the return value for SEARCH($\cdot$).

2. TRACK($\cdot$) combines the oculomotor controller developed in Project #3 with a complementary base controller to track the red ball continuously, keeping it within the field of view and the limited range of motion of the eyes.

   (a) use the oculomotor control developed in Project #3 to compute eye setpoints and complementary setpoints for the base that re-centers the eyes within their respective range of motion constraints; and

   (b) generate the return value for TRACK($\cdot$).

3. Construct an integrated SEARCHTRACK($\cdot$) skill. The internal state necessary to make control decisions is a multi-digit *word* where each digit is the return value of the constituent SEARCH($\cdot$) and TRACK($\cdot$) actions in the current run-time context.

   Implement a finite state supervisor that finds and tracks the red ball, recovers if the ball escapes the field of view, and reports the control status of the SEARCHTRACK($\cdot$) behavior.

   (a) evaluate the current status of SEARCH($\cdot$) and TRACK($\cdot$);

   (b) use this *internal state* to submit *either* SEARCH($\cdot$) or TRACK($\cdot$) setpoints to the motor units that control the base and the eyes, and;

   (c) return the SEARCHTRACK($\cdot$) control status.

Turn in a $\leq$ **3 page report** that includes:

1. a description of how you decided to distribute control over the eyes and the base;

2. a description of the finite state SEARCHTRACK($\cdot$) supervisor (a state-action table).

3. a plot showing the base control error versus time for a complete SEARCHTRACK($\cdot$) episode with at least two SEARCH($\cdot$) subgoals before TRACK-ing; and

4. a short discussion.

4.5 Project #5 - Stereo Triangulation

reading: Chapter 4
Section 4.5

lecture: Kinematics of Stereo Reconstruction

Sensory data is subject to noise and uncertainty. Certain kinds of uncertainty can modeled using Gaussian probability distributions parameterized by the distribution's mean and variance (μ, σ). Representations such as this support optimal estimators that suppress noise and improve precision by combining multiple observations (see Project 4.6).

In this programming exercise, you implement a triangulation procedure for localizing visual features in the base coordinate frame by estimating the position and covariance of observation.

```
typedef struct _observation {
   double pos[2];
   double cov[2][2];
   double time;
} Observation;
```

The `Observation` data structure is defined to represent the Cartesian mean and covariance of a spatial observation. In this case, it will record the stereo triangulation result and the covariance matrix estimated using a (scaled) quadratic form ($\mathbf{J}\mathbf{J}^T$) based on the localizability Jacobian.

Implement the procedure *stereo_observation()* in the project file *project5-StereoKinematics/project5.c* in which you will solve the triangulation equations, transform into world coordinates and write the mean and covariance of the result into the `Observation`. You can use the *project5_control()* procedure to call your implementation of *stereo_observation()* procedure every simulated millisecond and to conduct experiments (render examples and scale the covariance matrix).

Procedure *stereo_observation()* uses the coordinates *ul* and *ur* of a feature on the left and right image planes to estimate the position of the signal source in the (x, y) plane. Implement the stereo triangulation transformation to map $(ul,\ ur)$ into base frame $(x, y)_B$ and finally into world frame $(x,\ y)_W$ coordinates. The covariance matrix for the observation is assumed to be proportional to $\mathbf{J}_W\mathbf{J}_W^T$—the quadratic form constructed from the stereo localization Jacobian written in the world coordinate frame. The derivation of the localization Jacobian used the base frame, so you must rotate the Jacobian $\mathbf{J}_W = {}_W\mathbf{R}_B\mathbf{J}_B$ before you compute the quadratic form in order to orient the result correctly in the world frame. Check out the matrix operators provided in *matrix_math.c* or write your own. You will have to scale the localizability ellipsoid appropriately to account for error due to pixelization on the image plane. Select the smallest scale that draws an ellipse such that the ellipse still contains the true center of the red ball even as the estimate is disturbed by noise. It is often useful to do this by observing the performance of triangulation while TRACK-ing a moving ball.

Illustrate your results using procedure *draw_observation()* in the visualization procedure in the *project5.c* file to verify your result and show a few examples.

Report - Write a short report ($\leq$ **2 pages** and **no code**) that includes:

1. a description of the theory and your implementation;
2. a screen shot illustrating Roger's visual acuity, see if you can figure out how to illustrate several `observations` on a single screen shot; and
3. a short discussion of your results.

4.6 Project #6 - Kalman Filter

An implementation of the Kalman filter introduced in the book and the lecture notes is provided. Vary the relative magnitude of the process noise and the sensor noise to see the effect these parameters have on filter outputs.

The ball trajectory is subject to nonlinear events when the ball bounces of the walls (or off Roger himself). The linear filter design will find these events very challenging. Implement several extensions of the filter that can improve the performance of the filter in these conditions.

1. Compare the response of the filter to bouncing events for two different models of stereo reconstruction uncertainty: (a) the stereo localizability ellipsoid, and; a conservative isotropic model.
2. To reflect the range dependent uncertainty of a stereo observer, devise an isotropic $\sigma_{OBS}(x)$, where x is the distance between the robot and the estimated position of the ball.
3. Incorporate a model of the walls in the room to augment the process model in the Kalman filter. When the current estimate predicts that a contact will be formed, reflect the velocity estimate appropriately.

Report - Write a short report (no more than 3 pages and no code) that includes:

1. Explain the criteria you use to settle on a final filter configuration. Report the uncertainty parameters $\sigma_{OBS}(x)$ and σ_{PROC} that you selected to optimize behavior;
2. prepare a plot comparing the ground truth for the ball position and the output of the optimal linear estimator for a moving ball with a bounce event for the following cases.
   (a) using the observation uncertainty based on the stereo localizability ellipse vs. the isotropic model; and
   (b) the extended process model that reflects velocity at a bounce versus the strictly linear process model.

4.7 Project #7 - ChaseTouch()

This is the first comprehensive, whole-body application. The qualitative behavior of your implementation will depend on all the control decisions you've incorporated in your robot so far and how these closed-loop actions interact collectively with the whole-body dynamical system. If you approach the apparent center of the ball and reach to the same point with both arms, the result might appear to be *chase-touch*, *chase-grab* or a *chase-punch* depending on the controlled dynamics of your robot. This distinction may be relevant in subsequent projects where this basic eye-base-hand controller can be extended into a family of related skills in subsequent projects.

All constituent skills—SearchTrack(), Chase(), and Touch()—and the new composite ChaseTouch() skill should subscribe to the established convention for return state, i.e. return state $\in$ {no_reference, transient, converged} in a hierarchical control design.

Chase() : uses a finite state supervisor that executes SearchTrack() and *stereo_observation()* to write Cartesian setpoints $(x, y, \theta)_0$ to the motor units for the mobile base and returns its control status to the parent program.

Touch() : uses procedure *inv_arm_kinematics()* to define setpoints for one or both arm(s) to reach to and touch the red ball and returns its control status to the parent procedure. A converged status from Touch() should denote that a contact **with the ball** is detected on one or both hands. Tactile forces can result from self-collision as well as contacts with external objects. You might need to write a filter that distinguishes between these cases.

ChaseTouch() : implements a finite-state control supervisor orchestrating Chase() and Touch() actions on the basis of their feedback state and returns its control status. During execution, the field of view of the robot must be managed to eliminate self-inflicted occlusions. Include logic in the control supervisor based on the internal state of ChaseTouch() that returns both arms to a user specified "home" position if: (1) the ball is out of sight, or; (2) the ball is out of reach.

The development environment for ChaseTouch() (the *project7_control()* procedure) should run ChaseTouch() continuously to evaluate the control design.

Report - Write a short report (no more than 3 pages and no code) that includes:

1. a complete description of the finite-state supervisor (a state-action table) that sequences behavior in Chase(), Touch(), and ChaseTouch();
2. starting with the ball out of sight, plot: (a) Roger's base heading; (b)the distance between the red ball and Roger's base, (c) the distance between Roger's hands and the red ball on the same plot as a function of time for a complete ChaseTouch() episode. Explain the experimental data and indicate on the error plots where your program switches between SearchTrack(), Chase(), and Touch() actions; and

3. Discussion - How might changing gains, establishing pre-postures for the arms, offsets from the ball center for base and arm setpoints, and the relative timing of base and arm control actions influence how powerfully or precisely Roger can "touch" the red ball. How might these parameters influence the ability to catch and "grasp" the red ball?

4. (OPTION) Design a automated learning procedure that replaces the uniform SEARCH($\cdot$) distribution with a better, empirical model $Pr(gaze\ angle \mid \gamma_{track} = \text{CONVERGED \&\& } (x,y)_{roger})$. Gather data to evaluate the hypothesis that the empirical distribution will result in fewer SEARCH($\cdot$) actions to find the red ball. A good way to do this is to wait until an integrated CHASETOUCH() policy has been completed.

4.8 Project #8 - Harmonic Function Path Control: HARMONICPATHPLAN()

Complete the implementation of a harmonic function-based trajectory controller to execute collision-free trajectories to goals in the Cartesian occupancy grid. User defined obstacle and goal locations are defined using the GUI Input:Map Editor.

Tasks:

1. **obstacle dilation** - transform Roger into a "point robot" by dilating the obstacles labeled in the Cartesian world map (roger->world_map.occupancy_map[ ][ ]).

   (a) Define a "travel posture" for Roger that makes him compact and define the minimum radius, ROBOT_DILATE_RADIUS, that surrounds Roger's body and hands.

   (b) Complete procedure *dilate_obstacles()* in the project5.c file to dilate OBSTACLE boundaries in the occupancy map using this radius. Re-label FREESPACE bins that are within ROBOT_DILATE_RADIUS of an OBSTACLE from FREESPACE to DILATED_OBSTACLE.

2. **path relaxation** - procedure *sor()* repeatedly runs procedure *sor_once()* until the convergence criteria is reached. Complete procedure *sor_once()* by adding the SOR numerical iteration equation using potential values stored in roger->world_map.potential_map[ ][ ].

3. **define search goals** - add a set of discrete (x, y) goal locations in the Cartesian map (for use in procedure *init_search_locations()*) where Roger should go if he can't find the red ball. Adjust these locations as necessary in the second part of this project.

4. **return:** NO_REF - no more goals exist;
   UNCONV - enroute to a goal; and
   CONV - robot is in a goal state.

When your harmonic function is complete and the simulator is running, the potential map is displayed in grayscale on the map. If you push the "Visualize" button, streamlines are displayed to help visualize the potential function.

Report - Submit a brief report (no more than 2 pages). The report should include:

1. a concise problem statement;
2. a screenshot showing an example of your harmonic function illustrating obstacle dilation and streamlines visible in the grey level potential map, identify any type 1 (saddle) critical points; and
3. a short discussion of your results.

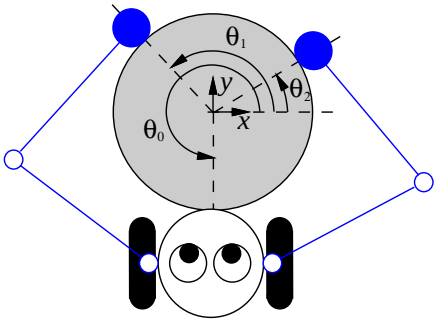
4.9 Project #9 - Learning an Integrated GRASP()

In this project, we will implement the FCLOSURE() reflex (a slight modification of homework exercise **??.??**) that optimizes a grasp configuration. Together with several other previously implemented primitives and macros that you built for Roger (SEARCH(), TRACK(), SEARCHTRACK(), CHASE(), a bimanual TOUCH(), and CHASETOUCH()), we will construct a Markov state-action space in which a reinforcement learning process will construct an integrated grasping behavior.

Be aware that the performance of your learning system will depend on the natural dynamics of your integrated CHASETOUCH() skill. The learning robot will encounter reward much more often if the underlying dynamics of the system is conducive to the FCLOSURE() reward.

1. **Part I** - FCLOSURE()

   The FCLOSURE() reflex turns estimates of contact wrenches derived from sensory feedback into differential contact movements that optimize grasp configurations. A necessary condition for equilibrium force closure grasps requires that the sum over all contact wrenches is zero. Moreover, a robust grasp distributes contact forces evenly over the three contacts, so this project considers the optimal placement of three frictionless point contacts that transmit unit magnitude forces normal to the object's surface at each contact.

   Before contact, values for θ_0, θ_1, and θ_2 can be estimated from knowledge of the positions of Roger's base, hands, and visual estimates for the position of the red ball. Given prior knowledge of the circular object geometry, we know that the wrench applied to the object by contact i is

   $$\mathbf{w}_i^T = [f_x \ f_y \ m_z] = [-cos(\theta_i) \ -sin(\theta_i) \ 0].$$

   This is a kind of visually-guided "*preshaping*" of the manipulator enroute to a grasp target.

   After contact, forces can be measured directly using tactile sensors and normalized to generate the contact wrenches.

   We could make two versions: FClosureVisual() (or "PreShape()") and FClosureTactile(), and let the reinforcement learning algorithm sort them out.

Define the "wrench residual" produced by these three contacts, to be

$$\boldsymbol{\rho} = \sum_{i=0}^{2} \mathbf{w}_i.$$

so that a navigation function can be defined representing the squared wrench residual $\phi_{\mathbf{w}} = \rho^T \rho$. The FCLOSURE() reflex recommends angular displacements for contacts θ_1 and θ_2 to descend $\phi_{\mathbf{w}}$ (we will consider θ_0 fixed wherever it initially falls on the object because it is not as easily controllable as θ_1 and θ_2).

Part I Report - Write a short report (no more than 2 pages and no code) that includes the details of your FCLOSURE() implementation:

(a) Write a closed-form expression for potential function $\phi_{\mathbf{w}} = \rho^T \rho$ that represents the (scalar) squared wrench residual.

$$\begin{aligned} \rho^T &= \begin{bmatrix} \sum_{i=0}^{2} -cos(\theta_1) & \sum_{i=0}^{2} -sin(\theta_i) & 0 \end{bmatrix} \\ &= \begin{bmatrix} -c_0 - c_1 - c_2 & -s_0 - s_1 - s_2 & 0 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \rho^T \rho &= 3 + 2(c_0 c_1 + s_0 s_1) + 2(c_0 c_2 + s_0 s_2) + 2(c_1 c_2 + s_1 s_2) \\ &= 3 + 2cos(\theta_0 - \theta_1) + 2cos(\theta_0 - \theta_2) + 2cos(\theta_1 - \theta_2) \end{aligned}$$

(b) Write a closed-form expression for the control Jacobian for the wrench residual,

$$\mathbf{J}_c^{\mathbf{w}} = \begin{bmatrix} (\frac{\partial \phi_{\mathbf{w}}}{\partial \theta_1}) & (\frac{\partial \phi_{\mathbf{w}}}{\partial \theta_2}) \end{bmatrix} \text{ so that, } \Delta\phi_{\mathbf{w}} = \mathbf{J}_c^{\mathbf{w}} \Delta\boldsymbol{\theta}.$$

$$\mathbf{J}_c^{\mathbf{w}} = 2[sin(\theta_0 - \theta_1) - sin(\theta_1 - \theta_2) \quad sin(\theta_0 - \theta_2) + sin(\theta_1 - \theta_2)]$$

(c) Implement force-closure controller: given θ_0, θ_1, and θ_2 into your closed-form expression for the control Jacobian to obtain a numerical $\mathbf{J}_c^{\mathbf{w}} \in \mathbb{R}^{1\times 2}$; compute the pseudoinverse numerically; and compute a differential contact movement that descends the navigation function $\phi_{\mathbf{w}}$

$$\Delta\boldsymbol{\theta} = (\mathbf{J}_c^{\mathbf{w}})^{\#} \Delta\phi_{\mathbf{w}} = -\kappa (\mathbf{J}_c^{\mathbf{w}})^{\#} \phi_{\mathbf{w}}.$$

(d) Transform displacement $\Delta\theta$ into new joint angle setpoints for the arms. There are several ways to do this: use $\Delta\boldsymbol{\theta}$ displacements on the surface to solve for new Cartesian goals and then use your *inverse_kinematics()* procedure to get new setpoints for the arms.

(e) Implement two "sensors" for estimating contact state:

i. visual guidance regarding contact state

ii. tactile guidance regarding contact state

two figures defining the governing geometry, both generate θ_0, θ_1, and θ_2.

(f) Implement two variants of FCLOSURE()

i. VFCLOSURE() (*preshape*) that uses the visually-guided estimates for $(\theta_0, \theta_1, \theta_2)$—for use when contact points on the object's surface are not yet reachable; and

ii. TFCLOSURE() that uses tactile estimates for $(\theta_0, \theta_1, \theta_2)$—for use when both contact points are reachable and higher quality tactile feedback is available.

These versions of Force Closure reflexes will be coordinated with other behavior to produce an integrated GRASP() action.

FCLOSURE(), like other actions, returns $state \in \{$NO_REFERENCE, TRANSIENT, CONVERGED$\}$.

2. **Part II** - formulate procedure GRASP() that explores actions

$$\mathcal{A} = \{\text{SEARCH}(), \text{TRACK}(), \text{SEARCHTRACK}(), \text{CHASE}(), \text{a bimanual TOUCH}(), \text{CHASETOUCH}(), \text{VFCLOSURE}(), \text{AND TFCLOSURE}()\}$$

and uses the Q-learning sampled backup to estimate a value function for deciding which of the controllers should submit setpoints to the underlying motor units using a 7-digit base three state identifier (2187 states!) to represent the multi-modal state feedback returned from candidate actions. Each action should introduce a penalty (a negative reward) that represents the cost of executing that action and the learning agent should receive a large positive reward inversely proportional to $\phi_{\mathbf{w}}$—the navigation function representing the force closure objective.

Randomly select actions from the current state, and backup the Q function to estimate expectations for the discounted sum of future rewards in the resulting rewards from the (s, a, s') transition.

Part II Report - Write a short report (no more than 4 pages and no code) that includes:

(a) a description of the penalty/reward;

(b) a learning performance curve showing the average rewards with error bars generated by greedy actions as a function of training;

(c) a comparison/discussion of learning performance curves when only primitives are used, when only macros are used; and when all primtives and macros are used.

4.10 Project #10 - Visuo-Tactile Modeling: Room and Objects

4.11 Project #11 - Belief and Model-Referenced Belief Dynamics

4.12 Project #12 - PONG

A new version of the simulator is used for this project that allows two Roger implementations to play against one another in a variation of the 1970's vintage video game "pong" (if you don't remember pong, think air hockey).

Rules of the Game - To begin a point, the ball is automatically placed in the center of the playing area. The ball is served when the simulator automatically launches the ball at a fixed "serve" velocity in a randomly chosen direction (bidirectionally, in appropriate intervals). Your objective is to keep the ball from reaching your opponent's goal while you try to get the ball to reach your goal. When the ball reaches either goal, play on that point is suspended, the score is updated appropriately, both players are reset to their starting position, and the ball is served once again.

1. If the ball stops and remains stationary on your side of the playing area for five seconds, a point is awarded to you opponent.
2. if any of the round parts your Roger's body touch the walls surrounding the playing area or cross the centerline, a point is awarded to your opponent (see *setpoint_filter()* method described below).
3. If a point continues for 20 simulated seconds, the point is awarded to the player with the least possession time.
4. The first player to reach 11 points wins.

The project reuses many of the previous projects, however several new behaviors are required to make a competitive PONG player, but there are probably more actions you should consider to build an intergrated PONG player that complies with all the rules. Among these:

4.12.1 Respecting the Arena Boundaries

Roger may not touch any of the walls or cross mid-court with any part of his body. This means that you should, at a minimum, never define setpoints for the robots body or hands that violate these constraints.

Make a model of the constraints and write a *setpoint_filter()* method that filters out all setpoint specifications that violate the arena boundaries. Demonstrate: (1) a collision with x and y boundaries without *setpoint_filter()* and the same manuever with the *setpoint_filter()* illustrating the behavior.

4.12.2 Offense()

The Offensive Ball Strike - The ChaseTouch() behavior coordinates the whole body of the robot to generate a touch event, but the quality of the ball strike can be low or can vary quite a bit.

Your goal is to develop an integrated ChasePunch() that approaches and strikes the ball reliably toward the opponents goal. Several different versions may be possible. State a design criteria and design a sequential policy that controls setpoints for a subset of the motor units of the robot (up

to the entire effector space) to establish pre- and post-strike postures that perform a high-quality ball strike.

The RETREAT() **Action** - In competitive games like PONG, it is often necessary to advance in an offensive strategy (as in CHASEPUNCH( )) and then retreat to a fixed defensive position where you can protect your goal more effectively. In both cases, it is wise to keep your eyes on the ball throughout. You've already written a CHASE() action to advance along a forward direction to approach the ball. In this project, we will build the complementary RETREAT() action, where Roger backs-up to a defensive position while keeping his eyes trained on the ball.

Create a new version of CHASE() called RETREAT() that defines appropriate $(x, y, \theta)_0$ setpoints for the base rotate and translate motor units, that backs Roger up to a goal-line defensive posture while TRACK()-ing the ball.

The Integrated CHASEPUNCHRETREAT() **Action** Describe a new CHASEPUNCH() + RETREAT() behavior using a state-action table. Try your strategy from many different positions of the robot and positions/velocities of the ball. Describe how you tested the result. Comment on the strengths and weaknesses of your implementation. Are all components of your integrated CHASEPUNCHRETREAT() good enough for all cases Roger will encounter in PONG? Are more varieties CHASE(), PUNCH, or RETREAT needed?

Demonstrate the performance of the action in a representative situation—use plots, screengrabs in a sequence, whatever you need to show a complete cycle of CHASEPUNCHRETREAT() behavior with a moving ball where Roger starts from and returns to a defensive position in front of his goal line that you define.

4.12.3 DEFENSE()

Ball Tracking and BLOCK() - Implement an integrated BLOCK() behavior. Devise a means of observing the ball, predicting the future path and computing an intercept point where you can move the body and/or the hands to block the motion of the ball toward the goal you are defending. Define how does this behavior returns NO_REFERENCE, TRANSIENT, and CONVERGED status?

BLOCK() may mark the transition from a defensive to an offensive ball strike discussed above.

4.12.4 Your Competitive PONG player

Design and implement a supervisor for an integrated PONG player using all the pre-packaged "macro" behavior (SEARCH(), TRACK(), SEARCHTRACK(), CHASE(), TOUCH(), BLOCK(), PUNCH(), RETREAT()) and any other primitives you decided to build (i.e. maybe AIM()) or hierarchical "macros of macros," Remember to fully define the state underlying the finite state supervisor in

terms of the return values of the primitives and macros that you employ and to exploit hierarchy as much as possible.

You are encouraged to play an experimental Roger against your best integrated design in the PONG arena environment to objectively evaluate the skill of design options you are considering. Document any of these experiments that influenced your design.

Your goal is to crush all the other Rogers in a competetive 2 player scenario.

Report - Submit a brief report (no more than 4 pages and no code) that includes:

1. a concise problem statement;
2. a description of the state-action table defining the PONG supervisor;
3. the convention for return values describing status of PONG; and
4. your thoughts on the PONG tournament and what you would change if you could do it over again.

Create and submit a tar file containing your implementation using "tar -cvzf USERNAME.tar.gz RogerProjects".

5 Matrix Library

Several basic matrix operations support application development. These methods are compiled into the simulator and are used by using the following calling procedures.

void matrix_copy(int m, int n, double M1[m][n], double M2[m][n])
copy matrix $\mathbf{M1} \in \mathbb{R}^{m \times n}$ into output matrix $\mathbf{M2} \in \mathbb{R}^{m \times n}$.

void matrix_mult(int m1, int n1, double M1[m1][n1], int n2, double M2[n1][n2], double M3[m1][n2
given inputs: $\mathbf{M1} \in \mathbb{R}^{m1 \times n1}$ and $\mathbf{M2} \in \mathbb{R}^{n1 \times n2}$, generate the output matrix product $\mathbf{M3} = (\mathbf{M1}\ \mathbf{M2}) \in \mathbb{R}^{m1 \times n2}$.

void matrix_transpose(int m, int n, double M1[m][n], double M2[n][m])
input $\mathbf{M1} \in \mathbb{R}^{m \times n}$ is used to generate the output $\mathbf{M2} = \mathbf{M1}^T \in \mathbb{R}^{n \times m}$.

void matrix_add(int m, int n, double M1[m][n], double M2[m][n], double M3[m][n])
given inputs $\mathbf{M1} \in \mathbb{R}^{m \times n}$ and $\mathbf{M2} \in \mathbb{R}^{m \times n}$, procedure matrix_add() generates the sum $\mathbf{M3} = (\mathbf{M1} + \mathbf{M2}) \in \mathbb{R}^{m \times n}$.

void matrix_subtract(int m, int n, double in1[m][n], double in2[m][n], double out[m][n])
given inputs $\mathbf{M1} \in \mathbb{R}^{m \times n}$ and $\mathbf{M2} \in \mathbb{R}^{m \times n}$, procedure matrix_subtract() generates the difference $\mathbf{M3} = (\mathbf{M1} - \mathbf{M2}) \in \mathbb{R}^{m \times n}$.

int matrix_invert(double* M, int n, double* M_inv)
given a square input matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$, this procedure generates the inverse $\mathbf{M\_inv} = \mathbf{M}^{-1} \in \mathbb{R}^{n \times n}$, the function returns 1 on success and 0 on failure.

void pseudoinverse(int m, int n, double M[m][n], double M_pinv[n][m])
given matrix $\mathbf{M} \in \mathbb{R}^{m \times n}, n > m$, procedure pseudoinverse() generates the Moore-Penrose right pseudoinverse $\mathbf{M\_pinv} = \mathbf{M}^{\#} \in \mathbb{R}^{n \times m}$.

void nullspace(int m, int n, double M[m][n], double M_pinv[n][m], double N[m][m])
given inputs consisting of matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ and its pseudoinverse $\mathbf{M\_pinv} \in \mathbb{R}^{n \times m}$, procedure nullspace() computes the nullspace (annihilator) of a matrix $\mathbf{M}$, $N(\mathbf{M}) = (I_n - M^{\#}M) \in \mathbb{R}^{n \times n}$.

void HT_invert(double aTb[4][4], double bTa[4][4])
a more efficient means of computing the inverse of a homogeneous transform that exploits the structure of homogeneous transforms.

void construct_wTb(double base_pos[3], double wTb[4][4])
a particular homogeneous transform (world-to-base) that comes up often.

6 Drawing Library

Each project development environment includes a procedure named *projectX_visualize()* in which the user can draw a number of different annotations onto the canvas for debugging purposes. The following methods reside in file xrobot.c and can be called from project code.

void draw_observation(Observation obs)
draws visual and tactile observations on the Cartesian canvas in world coordinates.

```
typedef struct _observation {
  double pos[2];    /* (x,y) */
  double cov[2][2]; /* covariance matrix J J^T */
  double time;      /* time stamp for the observation */
} Observation;
```

void draw_estimate(double scale, Estimate est)
draws the result of the Kalman filter estimator—the object position and velocity as well as the covariance $\in \mathbb{R}^{4\times4}$ in world coordinates.

```
typedef struct _estimate {
  double state[4];    /* (x,y, x_dot, y_dot) */
  double cov[4][4]; /* covariance matrix */
  double time;      /* time stamp for the observation */
} Estimate;
```

void draw_history()
Histories of up to 1000 robot postures are recorded and rendered when `Boolean HISTORY = TRUE` (in "include/simulate.h"). Data is recorded in array `History history[MAX_HISTORY]`, where type History is defined:

```
typedef struct _history {
  double arm_pos[NARMS][2];    /* (theta1, theta2) */
  double base_pos[3];          /* (x,y,theta) */
} History;
```

Procedure `draw_history()` draws the arm postures in the corresponding configuration space panels and the base position in the Cartesian panel.

void draw_streamline()
A streamline is the connected path from source to sink in the harmonic potential function. This procedure samples intial states near obstacle boundaries and follows the gradient all the way to a goal.