2 Overview

2.1 Introduction

This chapter describes the various programming techniques for using the core functions of the toolbox. It also gives an tutorial overview of the graphical user interfaces and other utility functions supplied by the toolbox. It assumes a working knowledge of the Khepera robot (see the Khepera User's Manual) as well as a basic knowledge of the Matlab programming language.

2.2 Programming With The Core Functions

This section provides an example of using the Khepera core functions to create a simple Matlab m-file that will run on the Khepera in real time. The core functionality of this toolbox is provided by the functions `con, skp, sms, rms, cpid, spc, rpc, rad, scps, rmc, cls, rps, and rls`. Each is a Matlab executable command (MEX command or function) that can be issued directly from the Matlab prompt or as part of an m-file or function. Each command allows the execution of one of the Khepera serial commands through a PC serial port. For a detailed explanation the serial commands, see the *Khepera User Manual* (ftp://lamiftp.epfl.ch/khepera). For details of the functions above, see the *Reference* section of this manual.

As an example of programming with the core functions, the code for the Braitenburg vehicle is rewritten in Matlab below. The code that implements the Braitenburg vehicle algorithm resides in the ROM of the Khepera. It is this code that allows the Khepera to be tested after unpacking (jumper mode 0).

The Matlab implementation of the Braitenburg vehicle algorithm is shown below:

```
%BRAIT      Braitenburg Vehicle Simulation
%

loopvar=0;
Intercon=[4 -5;4 -15;6 -18; -18 6; -15 4; -5 4;5 3;3 5];
speed=[10 10];
while loopvar == 0
      motors=(rps(2)*Intercon)/400+speed;
      sms(2,motors(1),motors(2));
end
```

In the very simple m-file above, the commands `rps`and `sms` are iterated until interruption by the user. The command `rps(2)` **r**eads the **p**roximity **s**ensors of the Khepera through PC serial port 2 and is equivalent to issuing the **N** serial command to the Khepera. The `rps` command returns a 1! 8 vector of proximity sensor values which is then multiplied by `Intercon`, an 8! 2 vector defined above the loop. The multiplication results in a 1! 2 vector which is then divided by 400 and added to the 1! 2 vector `speed`. The final result resides in the variable `motors`.

The next command, `sms(2,motors(1),motors(2))`, **s**ets the **m**otor **s**peeds of the Khepera through serial port 2 and is equivalent to issuing the **D** serial command to the Khepera.. The individual elements of the `motors` vector are indexed to provide the arguments to the `sms` command.

Since Matlab is a matrix oriented language, there is no need to implement **for...next** loops to multiply out the individual elements in each vector multiplication. This provides for a considerable reduction in the number of lines of code and, in most cases, makes for easier reading.

All control algorithms can be written in m-file format as done with the Braitenburg example just presented. However, several graphical user interfaces have been created to provide added research utility to the core functions

2.3 Programming With The Graphical User Interfaces

All control algorithms can be written in m-file format as done with the Braitenburg example just presented. However, several graphical user interfaces (GUI's) have been created to provide added research utility to the core functions. These GUI's and their use are described in the following sections.

2.3.1 Using the `Roborunner` Utility

Most research requires a number of trials of an algorithm to be executed and data to be stored after each iteration. This section describes the process of automating an algorithm for the control of a real Khepera using the **Roborunner** GUI. Using this interface, the specification of the number of iteration cycles per run and the number of runs are easily selected and changed. Additionally, there are options which allow variables and data to be stored after each run so that data can be collected and compared over a number of runs.

2.3.2 Converting the M-File to a Function

It is very likely that, up to this point, any algorithm written was implemented as a straightforward MATLAB m-file. The first step in making an m-file usable with the **Roborunner** gui is to convert it to a Matlab function which is callable by **Roborunner**. This is facilitated by copying the m-file code into the *algorithm space* of the `template.m` file (found in khepera\m_files\iface). This file contains all of the interface code required to connect the user-written m-file to the Roborunner gui. The algorithm space of `template.m` is indicated by the following message in the code:

```
%---------------------------------------------------------

%%%
% PLACE YOUR CODE HERE
%%%

%---------------------------------------------------------
```

You may remove or leave the message, but ensure that your code does not go beyond the two dashed lines which bracket the message.

As a running example, let us place the code for the Braitenburg vehicle from the previous section in the `template.m` file. We know that there are two instructions, `rps` and `sms`, which are executed in an endless loop. All other lines define variables or are related to the `while` statement. So, we would place the two lines of the `while` loop into the `template.m` file as shown:

```
while run_count<(runs)
    while cycle_count<(cycles_per_run)
      %---------------------------------------------------------

      %%%
      % PLACE YOUR CODE HERE
      %%%
      motors=(rps(2)*Intercon)/400+speed;
      sms(2,motors(1),motors(2));

      %---------------------------------------------------------
      cycle_count=cycle_count+1;
      end
```

The code segment as shown above will work with the **Roborunner** GUI (the code outside of the dashed lines belong to the `template.m` file). However, you will note that in both the `rps` and `sms` commands, the port through which the commands communicate is "hard wired" at 2. If you wish to exercise control of the port from the GUI, replace each reference to the particular port with the variable **port_id**. The result would look like:

```
while run_count<(runs)
    while cycle_count<(cycles_per_run)
            %---------------------------------------------------------

      %%%
      % PLACE YOUR CODE HERE
      %%%
      motors=(rps(port_id)*Intercon)/400+speed;
      sms(port_id,motors(1),motors(2));

            %---------------------------------------------------------
      cycle_count=cycle_count+1;
      end
```

The next task is to define any required variables. In the `template.m` file, all of the variables you wish to have defined would be placed in the *variable space* which is delimited by the following lines:

```
%----------------------------------------------------------

%%
% Initialize all your variables here
%%

%----------------------------------------------------------
```

For the Braitenburg example, we would write:

```
%----------------------------------------------------------

%%
% Initialize all your variables here
%%
Intercon=[4 -5;4 -15;6 -18; -18 6; -15 4; -5 4;5 3;3 5];
speed=[10 10];

%----------------------------------------------------------
```

Note that `loopvar` is not required since the `template.m` code takes care of all looping.

We must now globally define any variables we initialized above and wish to keep (this is required because `template.m` is a function and, as such, any variables defined within it will be discarded each time the function returns to the calling program). Globally defining the variables is as simple as listing them after the `global` statement in the following lines:

```
%%%
% LIST THE VARIABLES THAT NEED INITIALIZATION AFTER
% THE GLOBAL STATEMENT BELOW.
%%%
%---------------------------------------------------------
% global
%---------------------------------------------------------
```

Ensure that if you've listed any variable names after the global statement, that you remove the comment symbol (%) from the front of the line. The Braitenburg example shows how this line would appear:

```
%---------------------------------------------------------
global Intercon speed
%---------------------------------------------------------
```

There are two further sections of the `template.m` file which can be customized by the user. The first section, delimited by dashed lines, contains the following message:

```
%%%
% END OF RUN.  EVERYTHING TO BE DONE BETWEEN RUNS SHOULD BE DONE HERE
%%%
```

Any function or command that you would like to see implemented after each *run* can be called at this point. The second section contains the following message:

```
%%%
% PLACE ANY HOUSEKEEPING COMMANDS HERE (IE SMS(2,0,0) TO  STOP REAL
KHEP)
%%%
```

Place any code here that your would like to see implemented after the completion of the very *last cycle of the last run* of the algorithm. For example, you may wish to issue a command to ensure that the Khepera is stopped (eg. `sms(2,0,0)`) or to turn off its lights (eg. `cls(2,1,0)`).

The final modification of the function is to give it a name which is different from `template.m`. For the Braitenburg example, you might change the first line of the `template.m` file from:

```
function [%%Return Variables%%]=
%%name%%(port_id,cycles_per_run,runs,pause_stat)
```

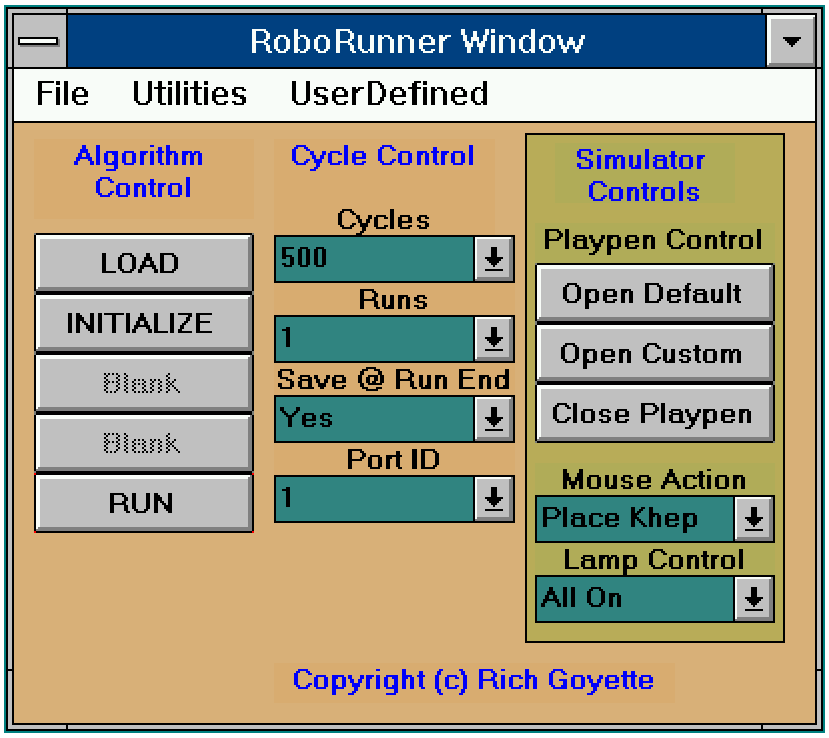
to:

```
function brait(port_id,cycles_per_run,runs,pause_stat)
```

Lastly, save the modified file. The name of the file **must** be the same as the name you gave to the function. This example above would be saved as `brait.m` (this sample file is available in the `khepera\examples`). The m-file is now ready to be used by the **Roborunner** GUI.

2.3.3 Using the *Roborunner* GUI to control an algorithm

To start the gui, type `roborun2` at the Matlab command prompt. This will open a window as shown below.



Cycles: 500
Runs: 1
Save @ Run End: Yes
Port ID: 1
Mouse Action: Place Khep
Lamp Control: All On

Figure 2
RoboRunner Window

The functions of this window that we are concerned with in this section are those listed under **Algorithm Control** and **Cycle Control**. The **Simulator Controls** will be described later.

The execution of the `brait.m` algorithm (created in the previous section) would proceed as follows. First, click on the **Load** button. A filename selection window appears. Select the name of the algorithm by either highlighting it and clicking on OK, or by double-clicking on the name.

Next, if any variables were placed in the initialization section of the `template.m` function when `brait.m` was created, then you will need to click on the **Initialize** button.

Now you may use the drop-down menus under **Cycle Control** to specify the number of cycles per run (500 - 10000) and the number of runs (1-20). The `brait.m` algorithm will run for *n* cycles, *m* times as specified by these menu selections.

If you wish to save your variables at the end of each run, select **Yes** under **Save@Run End**. Whenever a run is completed, execution will pause and a window will appear asking for a file name under which to store the data. Select or type a filename and directory and click **OK**. Note that variables are stored using the Matlab **save** command with only a filename as an argument. As a result, *all* of the current environment variables are stored in MAT file format under the single filename you provide. For details on the MAT file format, see the *Matlab External Interface Guide*. In order to retrieve the variables from this file for later analysis, simply type the command **load** *filename*, where filename is the name you gave the file. If you then execute the command **who**, you will see all of the variables that existed at the end of the run for which they were saved. You may customize the save command when modifying the `template.m` file to save only certain desired variables.

Finally, use **Port ID** to select the port to which all commands should be sent. If you are simulating a Khepera, you may ignore this setting (more about simulation later).

2.3.4 Variable Passage and User Defined Functions (Optional)

As you will see later, the RoboRunner GUI has a UserDefined pulldown menu from which you may define and select user written functions to do data manipulation or other activities. However, when the iteration of the algorithm ends and control returns back to the Roborunner gui, any variables that belonged to the algorithm will not be passed back with the **return** unless you specifically direct this to occur. Passing back variables can be done in the following manner. Modify the first line of your `template.m` file from:

```
function [%%Return Variables%%]=
%%name%%(port_id,cycles_per_run,runs,pause_stat)
```

to:

```
function [var1, var2,...]= brait(port_id,cycles_per_run,runs,pause_stat)
```

where *var1, var2*, etc are the variables you wish to pass back to the Roborunner gui. Then, modify the call to the function in the `roborun2.m` file. This file has a small message outlining what the change should look like and is pretty much self explanatory:

```
%%%
% Modify the following line to add variables to be
% passed back from the looping function.  List your variables
% in square brackets, separated by commas in followed
% by an equals sign.  For example:
% [var1,var2]=feval(ctrl_name,port_id,numcycles,numruns, pstat);
%%
feval(ctrl_name,port_id,numcycles,numruns,pstat);
```

You may then pass these variables as arguments to any UserDefined function. Writing and interfacing UserDefined functions is described later. Note: since it is possible to save variables to a file after each run, variable passage may not be required if your userdefined function can be written to load these variables

2.4 Khepera Simulator

2.4.1 Simulation Environment Overview

A number of the serial commands that can be issued from the Matlab prompt for use with a real Khepera are also available for use with a simulated Khepera. Each of these commands is preceded by the identifier **sim_** (see chapter 1). This capability is provided by the function `kheprom()` which emulates the ROM on a real Khepera. The interaction between the function `kheprom()` and the simulator serial functions is shown in the diagram below. In essence, the simulated serial communications functions act to modify the contents of various registers and then `kheprom()` operates on these registers (in the same manner as the ROM on board the Khepera) by interpreting their contents and carrying out the required actions. The `kheprom()` function is a virtual Khepera ROM.

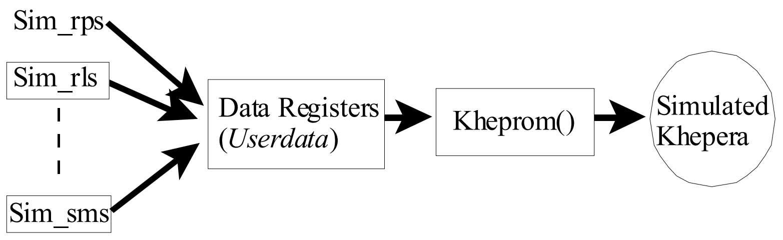


Figure 3
Simulated Serial Command Interaction With Kheprom()

The following sections describe the procedures for using the Khepera simulation environment.

2.4.2 Creating a Virtual Environment (Playpen

This toolbox version offers limited functionality to create and customize a simulation environment in which a simulated Khepera can move about. The environment is called a Virtual Playpen, or just Playpen. Playpens can be designed using the **Khepera Virtual Playpen Designer** window which can be accessed by typing the command `play_pen` at the Matlab prompt or, if the **RoboRunner** window is open, by selecting **Utilities**! **Khepera Simulation**! **Create New Playpen**. When either of these two options is taken, the window shown in Figure 3 appears.

The area inside the blue wall is the Playpen region, where the Khepera will move about. There are three types of object that can be placed down under **Object Types** - **Wall 0**, **Wall 45**, and

Wall 90. Each is a small rectangular block rotated 0, 45, -45, and 90 degrees from the horizontal. Under **Actions**, there are three options - **Add Object**, **Delete Object**, and **Move Object**. The fourth listed item, **Rotate Object**, is not currently available.

To create a wall, select the object type to be placed. Then select the action **Add Object**. Move the mouse pointer to the desired location and click the left mouse key. You may delete it by selecting **Delete Object** from the **Actions** pulldown and clicking on that object with the left mouse key. You may also move it by selecting **Move Object**, placing the mouse on the object, and dragging the mouse to the desired location.

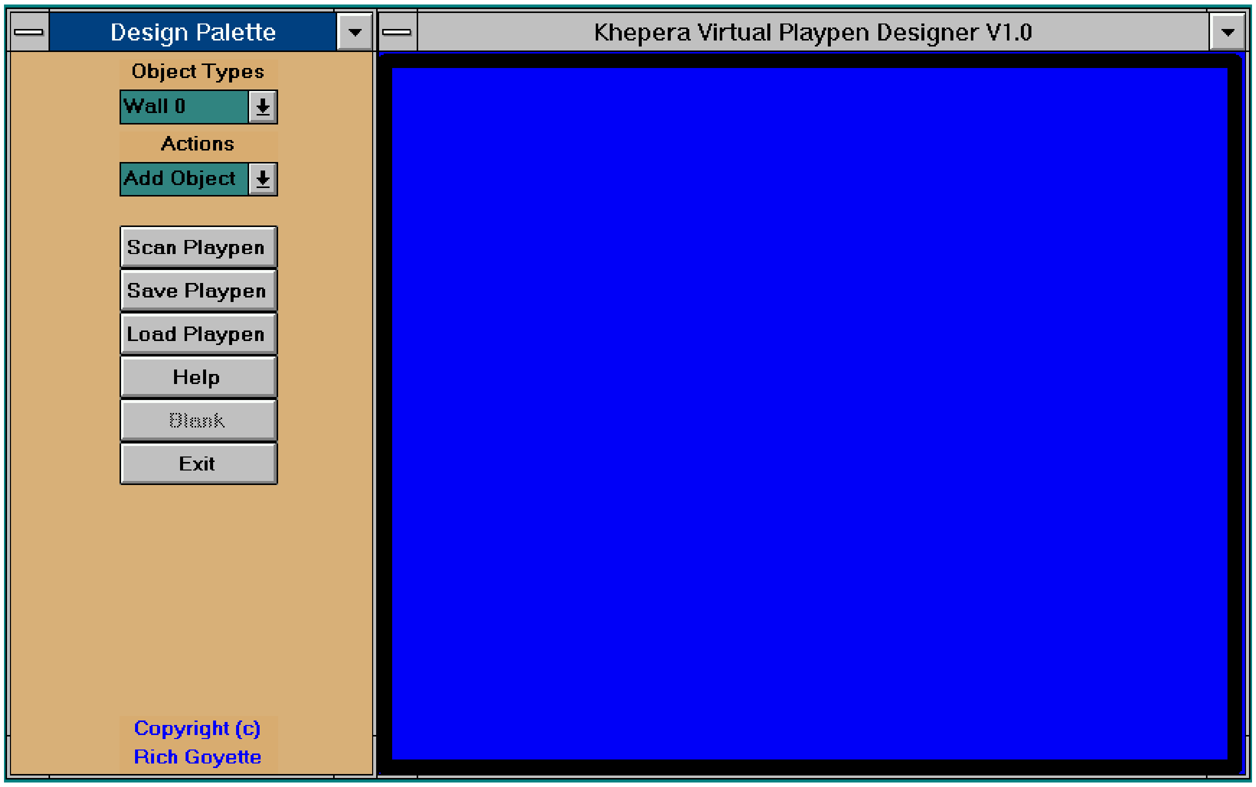


Figure 4
Khepera Virtual Playpen Designer

If the above description seems tedious for the creation of even a simple environment, that is because it is. Thus, if you have a three button mouse, select **Move Object** from the **Actions** menu and leave it at that selection. Clicking on the middle mouse button places the currently selected object type at the current mouse location. You may drag-and-drop using the left mouse button. If you do not want the object at all, place the mouse pointer over it and click the right mouse button and the object will be deleted.

To save the Virtual Playpen for later modification, click on **Save Playpen**. A **Save-As** window will come up with the default file name *.mat. Ensure your file name has the extension .mat and click OK when you have supplied the required name and destination directory. *This file will not be usable in the simulation environment until it has been scanned. Since scanned files cannot be read back into the Playpen Designer (the environment is converted to an image map and all object properties are lost), the purpose of saving at this point is to allow the work you have done to be loaded back in and modified later.*

To make the Virtual Playpen usable as a simulation environment, click on **Scan Playpen**. Wait a few moments while Matlab creates an image map, and then supply the required file name (with the .mat extension) and destination directory.

If you wish to continue work on or modify an existing environment, click on Load Playpen and select the desired file from the dialog box which appears.

Exercise care in selecting names when saving, loading, or scanning. All of the relevant files have the .mat file extension but scanned files and saved files contain different information. Do not try to load a scanned file using the **Load** button of the **Playpen Designer**. In the same manner, in the **RoboRunner** window, do not select a file saved using **Save Playpen** above after clicking on the **Open Custom** button.

When a file is scanned, it is flipped along its X axis. Therefore, when you open a scanned file from the **RoboRunner** window to use as the environment for a simulated Khepera, it will appear rendered upside-down.

Periodically, the top (and sometimes the right side) of the Playpen will disappear while editing an environment. It is not gone, rather just not visible. To bring it back to the top, add an object, move it, then delete it. This problem will be addressed when this author's thesis is done.

2.4.3 Simulation Options in the RoboRunner Window

The controls for the simulation of a Khepera are located in the off-coloured box region of the **RoboRunner** window (labelled **Simulator Controls** - see Figure 2).

Before a robot can be simulated, an environment must be present for it to move about in. Clicking on the **Open Default** button will open the window shown in the figure below.

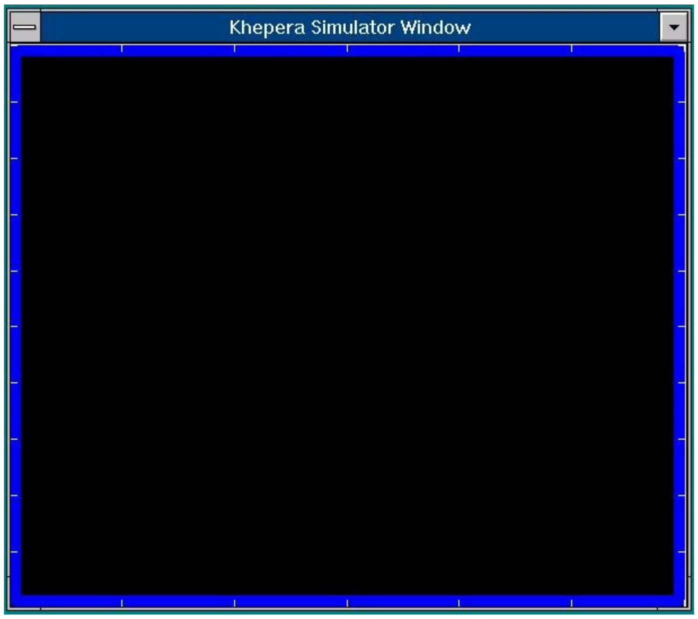


Figure 5
Default Playpen

This window displays the default Playpen which consists of basically four retaining walls and no other obstacles. By using the **Create New Playpen** option under **Utilities** as described previously, it is possible to add obstacles/walls to this default playpen. If this has been done, then you may open the modified playpen by clicking the **Open Custom** button.

You may close the playpen at any time by clicking the **Close Playpen** button.

The **Mouse Action** pulldown menu describes the action that will take place when a mouse click is detected in the **Playpen** window There are separate entries in the pulldown for placing and removing the Khepera from the **Playpen** window. However, if the **Mouse Action** pulldown is left on the **Place Khep** action, then clicking the left mouse button places the simulated Khepera while clicking the right mouse button removes it.

You may rotate the Khepera left or right by selecting the **Rot Left/Rot Right** action and clicking in the **Playpen** window with the left mouse button. Since there is only one Khepera allowed in the **Playpen** window, it does not matter where the mouse pointer is when Khepera related actions are performed.

Note that a **Playpen** window must be open to enable this pulldown menu. Also, there are no error checks to ensure that the Khepera is not placed on top of an object. The **Move Khep** option is not yet implemented since the Khepera can be moved by successive **Place** and **Delete** commands.

The last two options in the **Mouse Action** menu are **Place Lamp** and **Remove Lamp**. There is no limit on the number of lamps that can be placed. They can be placed anywhere in the Playpen window (including on top of walls). This software version does not implement the shadowing effects of walls. When deleting a lamp, the lamp which is closest to the mouse when the left mouse button is pressed is deleted, followed by the next closest, etc.

The **Lamp Control** pulldown menu allows three options for controlling the sequencing of lamps in the environment. The first option will keep all the lamps on during any experiment. The next option, **Random**, will turn on a lamp at random at the beginning of an experiment. Whenever the robot moves sufficiently close to the lamp or wall, it will switch off and another will be selected from those in the playpen to be switched on. The last option, **Sequenced**, will perform the same way as random, except that the lamps will be sequenced in the order in which they were placed down. This pulldown can be changed at any time (when an algorithm is not running) and the selected operation will begin at that point.

2.4.4 Programming Considerations for Simulating a Khepera

When programming to use a simulated Khepera, use the simulated serial command set. These commands include `sim_skp, sim_sms, sim_rms, sim_spc, sim_rpc, sim_rmc, sim_rps`, and `sim_rls`. The usage of each is exactly the same as its real counterpart, the only apparent difference being the affixed `sim_` identifier. Each command even accepts a serial port identification argument even though it is not used. There are some functionality differences. See the notes on the `kheprom` command in the *Functions and Commands* section to get an appreciation for these differences.

In a real Khepera, when a serial command is received, the ROM acts independently to carry out the command. The host machine can "fire-and-forget" each serial command. In simulation, this is not the case. The host machine is responsible for updating the condition of the simulated Khepera. This is done by issuing the `kheprom()` command *once during every cycle of the algorithm.* A segment from the file `simsamp1.m` (in `khepera\examples`) is shown below (this file used `template.m` as described in *Automating a Khepera Control Algorithm* to make it accessible from the **RoboRunner** window).

```
while run_count<(runs)
    while cycle_count<(cycles_per_run)

      %-----------------------------------------------------------
      %%%
      % PLACE YOUR CODE HERE
      %%%
      sim_sms(2,1,1);
      a=sim_rps(2);
      if any(a>900)
            sim_sms(2,-1,1);

      end
      kheprom(btmap1);
      %-----------------------------------------------------------
      %% LOOP CODE - DO NOT REMOVE
      cycle_count=cycle_count+1;
    end
```

This algorithm will cause the Khepera to move forward whenever no objects are detected by the proximity sensors. Whenever any sensor "sees" an object (any sensor value greater than 900), the Khepera is instructed to move in a circle (`sim_sms(2,-1,1)`).

Notice the use of the simulated serial command set. Also note that after all commands have been issued, and just prior to the end of the loop, the `kheprom()` command is issued. `btmap1` is the only argument taken by `kheprom()` and must always be globally defined in the function as shown below.

```
%-----------------------------------------------------------
global btmap1
%-----------------------------------------------------------
```

The kheprom command carries out all the functions that would normally be done by the ROM on a real Khepera robot. See `kheprom()` in the *Commands and Functions* for a detailed review of the simulation environment.

2.4.5 Converting a Simulation Algorithm for use with a Real Khepera

The implementation similarity between the simulated serial command set and the real serial command set was imposed purposely to allow algorithms to be written for use with the simulator *first* as a proof of concept. When details are ironed out, the algorithm can be searched using a text editor and each occurrence of `sim_` can be removed. The command `kheprom()` must also be removed (as well as the global definition of `btmap1`) since the real Khepera ROM will take care of the robot. Once this is done, the algorithm will operate through the serial port with a real Khepera. However, the notes on kheprom in the *Commands and Functions* section should be reviewed thoroughly to appreciate the differences between the real and simulated commands.

If the code segment of the previous section was converted to operate on a real Khepera, it would appear as follows:

```
      while run_count<(runs)
          while cycle_count<(cycles_per_run)

            %----------------------------------------------------------
            %%%
            % PLACE YOUR CODE HERE
            %%%
            sms(2,5,5);
            a=rps(2);
            if any(a>900)
                  sms(2,-5,5);
            end
      %----------------------------------------------------------
      %% LOOP CODE - DO NOT REMOVE
      cycle_count=cycle_count+1;
    end
```

Notice the absence of the kheprom command as well as the use of `sim_` on any of the Khepera commands. The global definition of `btmap1` has also been removed.

Also note the fact that the magnitudes of the motor values in `sms` have been increased to five. The simulator operates with one speed, so that the magnitudes of the numbers assigned to the motor speeds in the command `sim_sms` are irrelevant. So, for the original code, they were assigned a value of 1 or -1. However, in the real Khepera, the magnitude of the speed values is very important. A magnitude of 1 is very, very slow. So, the speed values were changed to 5. This is an example of some of the subtle differences between the simulator command set and the real command set.

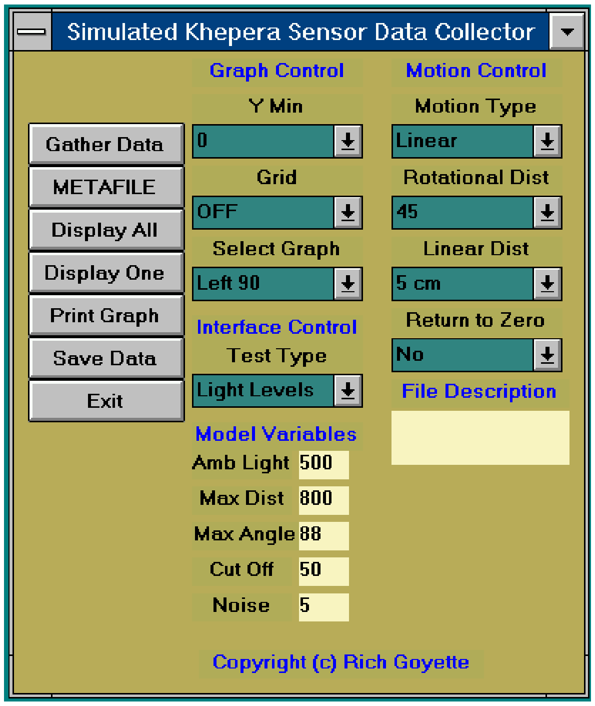
2.5 Gathering Sensor Data from a Real Khepera

It is sometimes desirable to get a feel for the values of the light and proximity sensors at various distances and angles from both walls and light sources prior to working with the Khepera. It is desirable because not all surfaces exhibit the same IR reflection and/or absorption characteristics and hence will provide different values at different distances and angles for the proximity sensors than would some other material. As well, a range of different light sources may be used, each of a different wattage or type. Having an understanding of the effect of these variables on the Khepera's sensors is important because, undoubtedly, decisions will have to be made within most algorithms based on threshold values. A quick example is the algorithm segment provided in the previous section. It decided that the Khepera would spin if any sensor registered a value greater than a threshold of 900.

Sensor data gathering from a real Khepera is described in this section. Sensor data can also be gathered from a simulated Khepera and this is described in the next section. The collection and comparison of sensor data from both real and simulated Khepera's is important for sensor calibration which is the topic of Chapter 3.

2.5.1 Performing a Linear Approach Test

Structured experiments can be performed to gather data using the **Utilities**! **Real Khepera**! **Sensor Testing** menu option in the RoboRunner window (or, from the Matlab command prompt, type `tstsense`). Selecting this option opens the window shown below.


Simulated Khepera Sensor Data Collector

Graph Control
Y Min: 0
Grid: OFF
Select Graph: Left 90

Interface Control
Test Type: Light Levels

Model Variables
Amb Light: 500
Max Dist: 800
Max Angle: 88
Cut Off: 50
Noise: 5

Motion Control
Motion Type: Linear
Rotational Dist: 45
Linear Dist: 5 cm
Return to Zero: No

File Description

Copyright (c) Rich Goyette

Figure 6
Sensor Testing for Real Khepera

Gathering data involves two steps. First, set up the experiment. This is done by selecting options from among the pulldown menus under **Interface Control**, and **Motion Control**. Let us assume we wish to examine the effect that a 1 watt light bulb has on the ambient *light* sensors as the Khepera advances along a straight line path from 55 cm away to 0 cm away. Let us also assume that the Khepera is connected to port 2 of the PC (and that the power cube is plugged in - really, I'm not joking, make sure you check it before you spend 3 hours looking for the problem).

Under **Interface Control**, set **Port ID** to 2 and **Test Type** to **Light Levels**. Under **Motion Control**, set the **Motion Type** to **Linear**, and set **Linear Distance** to 55. Now, if you wish to have the robot move back to the location it started from automatically (ready for the next trial), set **Return to Zero** to **Yes**. Finally, you may set the **Max Speed** and **Max Acceleration** control values so that the robot moves in a reasonable way. The values shown in the lower right are the default values. If you set the speeds to 4, the robot will move much more slowly and many more data points will be taken for each test. Ensure that the left and right speeds and accellerations are equal or the Khepera will not move in a linear fashion.

The next step is to conduct the experiment, examine the results, and save them if required. To conduct the experiment, simply click on **Gather Data**. The robot moves forward 55 cm, tabulating light data as it goes. If **Return to Zero** is set to **Yes**, it waits for 5 seconds at the end and then automatically goes back to the starting point.

Now the data may be displayed. If you want a global picture of what each sensor reads, click on **Display All**. This action will produce a graph similar to Figure 6. Each graph corresponds to a sensor. The axes are not marked since there is very little room. However, for light experiments, the Y axis is the ambient light sensor reading. For proximity experiments the Y axis corresponds to the proximity value measured by each sensor. For linear tests, the X axis has units of centimetres, and the axis is reversed. This is because it is assumed that the end position is the 0 centimetre position. For rotational tests, the X axis has units of degrees.

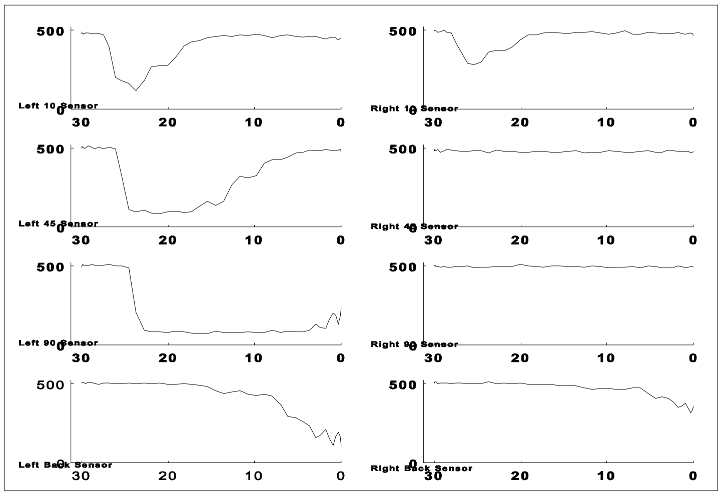


Figure 7
Graph of all Sensors

Let's assume that you like this graph and would like to incorporate it into a word processing package. Click on **MetaFile** to save the most current Matlab figure to the *Windows* clipboard. Now lets assume you wish to examine the left 10 degree sensor in detail. Close or minimize the current graph (to unclutter the desktop environment). In the **Graph Control** column under **Select Graph**, choose **Left 10**, then click on **Display One**. A larger graph appears showing data from only the chosen sensor. An example of such a graph is shown in the figure below.

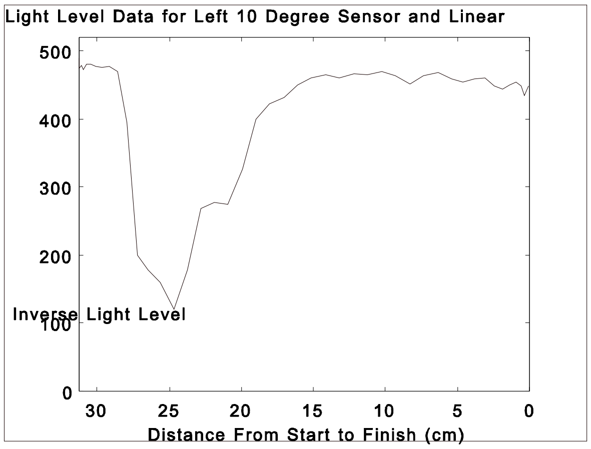


Figure 8
Graph of Individual Sensor

If all the data points are above 200 and you would like a closer look, close or minimize this graph, select 200 from the **Y Min** pulldown under **Graph Control**, and click on **Display One** again. A new graph is displayed showing all data above 200 on the Y axis. If you would like a grid, close or minimize the current graph, select **On** from the **Grid** pulldown under **Graph**

Control and click on **Display One** again. Print the graph by clicking on **Print** (note that the print command is set up for the Canon bj10e - see the `dataprnt` command in the *Functions and Commands* section for changing the default printer).

If you want to save the data, click on **Save Data**. See the command `tstsense` in the *Functions and Commands* section for details on the format in which the data is stored and how to retrieve the data when required. If you wish to add some informative text to the data you are about to save (like the test type, number, date, wife's birthday, etc), enter this information in the **File Description** box. The number of characters per line and number of lines in this box are essentially unlimited.

2.5.2 Positioning the Khepera for a Test

What if the distance was misread and the robot fell short of or ran into the light bulb during the test? If you want the robot to end up exactly at the bulb, place it there with its back facing the line of approach. Select -55 from the **Linear Dist** pulldown (which makes the robot go backwards), ensure that **Return to Zero** is **Off**, and click on **Gather Data**. This moves the robot back exactly 55 cm. The robot will collect light data while it moves back, but you will not use it since you are using this function to position the robot. You may now reset **Linear Dist** to 55, **Return to Zero** to **On**, and click **Gather Data** again to take the actual measurements. The robot will proceed right up to the bulb because it measured itself back 55 cm.

2.5.3 Performing a Rotational Test

You may now wish to see what effect rotating beside a wall would have on the *proximity* sensors. Select **Proximity** from the **Test Type** pulldown, **Rotational** from the **Motion Type** pulldown, and, say 360 degrees from the **Rotational Dist** pulldown. Click on **Gather Data** and watch the Khepera rotate. All other graphing and saving functions are the same as for the linear test described above. If you click on **Display One**, with **Select Graph** set to **Back Right**, you might see a graph like the one in Figure 9 below. Rotational tests are useful for examining the effect that the angle between the sensor and the light or object has on the light or proximity measurement. It is also useful in determining the effect (if any) of ambient light reflected off of surfaces and of the shadowing effect of obstacles.

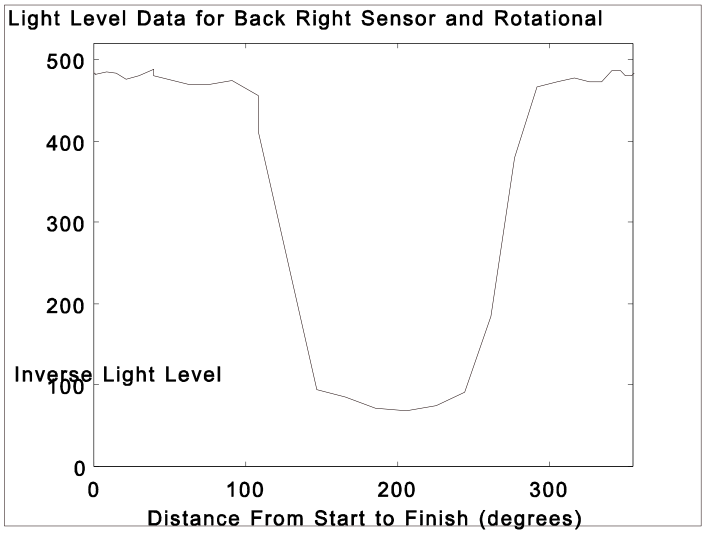
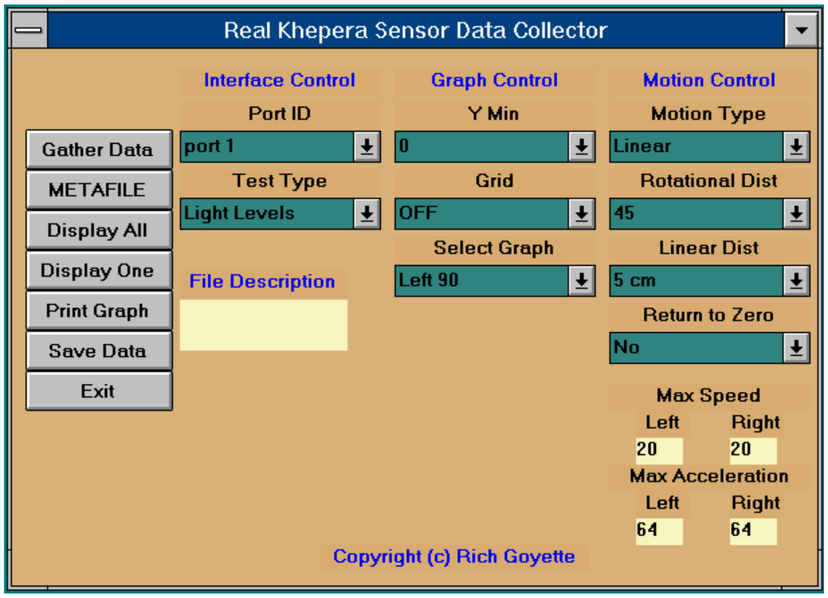


Figure 9
Individual Sensor With Rotational Data

2.6 Gathering Sensor Data from a Simulated Khepera

Sensor data can be gathered from a simulated Khepera in almost exactly the same way as that of a real Khepera. This can be done by selecting the **Utilities**! **Khepera Simulation**! **Sensor Testing** option in the RoboRunner window. This action will result in a window as shown below.



Port ID: port 1
Test Type: Light Levels
Y Min: 0
Grid: OFF
Select Graph: Left 90
Motion Type: Linear
Rotational Dist: 45
Linear Dist: 5 cm
Return to Zero: No
Max Speed Left: 20 Right: 20
Max Acceleration Left: 64 Right: 64

Figure 10
Sensor Testing for Simulated Khepera

While it is possible to open this window by typing `simsense` at the command prompt, the interface will not work properly unless there is a **Playpen** window open with a simulated Khepera in it. To open a **Playpen** window, click on **Open Default** (or if you've designed your own playpen, **Open Custom**). When the **Playpen** window appears, select **Place Khep** from the **Mouse Actions** pulldown in the **Roborunner** window and click at some desired location in the **Playpen** window to place the simulated robot. If you wish to have a light source, select **Place Lamp** from the **Mouse Actions** pulldown and click at the desired location in the **Playpen** window. As described earlier, you may rotate, move, and delete the simulated Khepera and lamps as required to set up the test environment.

Notice that while the **Simulated Khepera Sensor Data Collector** GUI is slightly different from the **Real Khepera Sensor Data Collector** GUI, the functionality of both is basically the same. The **Simulated Khepera** gui's size and position was modified to allow it to fit on a 1024! 768 screen along with the **RoboRunner** window and the **Playpen**. The only other differences are 1.) the removal of the **Port ID** pulldown menu from the **Interface Control** column of pulldowns and the speed and acceleration option settings (this was done since a simulated Khepera has no need for a port identification or for speed control); and 2) this window incorporates a number of settable variables for the control of the modelled light response. These settings are discussed in chapter 3.

For a description of using the interface to perform various tests, see the previous section.

2.7 Writing UserDefined Functions

In the RoboRunner window, under the menu option **UserDefined**, there are several items labelled **User1**, **User2**, etc. These were placed here to allow the user to access functions written by him or her. The functions may be simple or complex. They may open separate GUI's, or they may just do a simple calculation. An example of a UserDefined function is given in `khepera\examples` as `User1.m`.

To interface a UserDefined function to the UserDefined pulldown, you must first compose the function. When this is done, name it `user`*`x`*`.m` where *x* is the number of the UserDefined menu that you wish to occupy. For the simple case where the function is completely independent (passes back no variables and requires no variables), nothing else is required. Remember that, for functions, the name of the file must be the same as the name of the function. In this case, if we chose the User1 position, the file would be named `user1.m` and would have the first line `function user1().`

If the function requires variables passed to it from the **RoboRunner** environment, then modification of the `roborun2.m` file is required. Open roborun2.m in a text editor and find the line:

```
%% Userdefined Actions
```

Each elseif statement corresponds to one entry in the UserDefined menu. Find the `elseif` for the number of the pulldown you have chosen (eg user1, user2, etc,) and append the list of required variables to the end of the user statement. For example, if we chose the User2 position, and our function needed the variables `a` and `b` (passed back from some other Userdefined function?), then we would change the following code segment:

```
elseif strcmp(command_str,'ud2')
      if exist('user2')
            user2;
      end
```

to read:

```
elseif strcmp(command_str,'ud2')
      if exist('user2')
            user2(a,b);
```

```
        end
```

The most complex case would be if the UserDefined function both required variables to be passed to it and passes back variables for use in the RoboRunner environment. If the function takes `a` and `b` as above, and passes back `c` and `d`, then the code segment above would look like:

```
    elseif strcmp(command_str,'ud2')
        if exist('user2')
            [c,d]=user2(a,b);
        end
```

It is recommended that, where possible, the functions be written in an independent fashion. As mentioned in a previous section, data can be passed back and forth between functions by saving them to a file using the `save` and `load` commands. Since these commands are not thoroughly described in the *Matlab Reference Guide*, it is recommended that all entries in the Matlab technical support database (www.mathworks.com) for `load` and `save` be reviewed.

2.8 Khepbase: A Useful GUI for Sensor Viewing

This toolbox contains a useful function named `khepbase.m`. It allows the viewing of the light, proximity, and motor speed values in a graphical manner. When the function is invoked by the call khepbase, a window opens in the bottom left of the screen which displays an overhead projection of the Khepera. At each sensor, a pair of bars are drawn which will show graphically the values of the light and proximity sensors. As well, two bars are drawn beside the motors to indicate the current speed. During each iteration of an algorithm, this window can be updated by calling the khepbase function with light, proximity, and speed values as arguments. See the *Functions and Commands* section for more details. `simsamp2.m` in the `examples` directory is a sample function (which was visited before) which implements `khepbase.m` on a simulated Khepera. A typical khepbase window is shown in Figure 11.

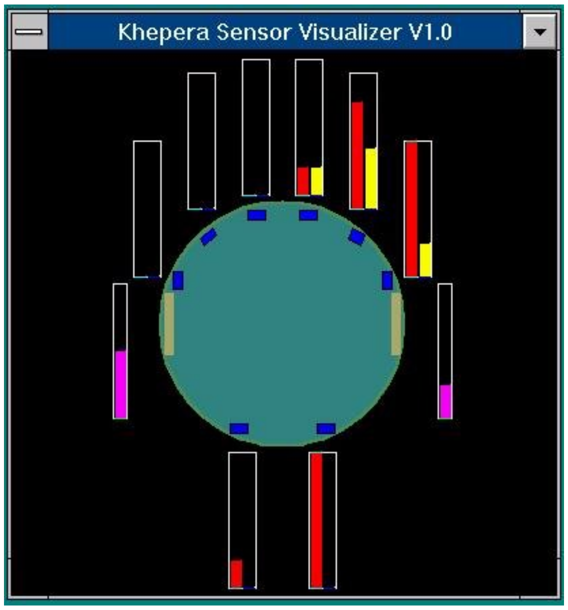


Figure 11
Khepbase Visual Sensor Display