Laboratory on Agent-based Mobile Manipulation

Patricio Nebot, Enric Cervera

Part I: Running the Agent Controller

<u>Step 1</u>: Open the system

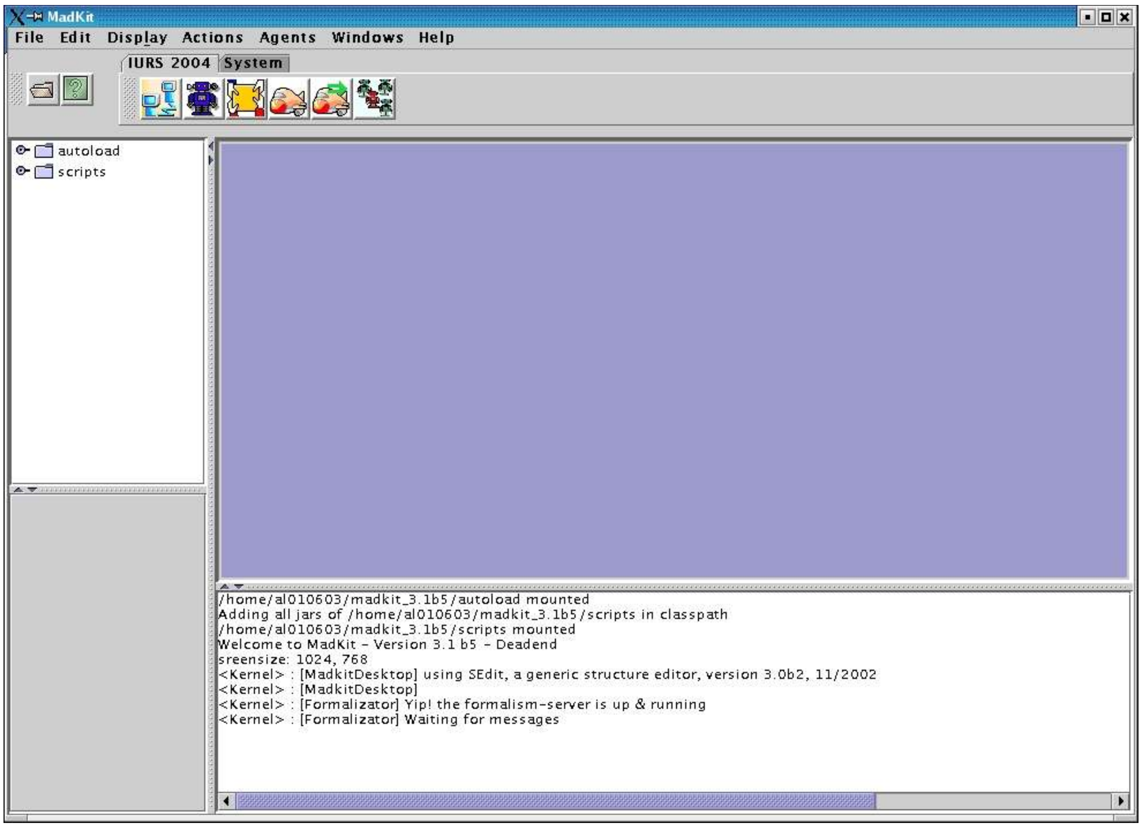
First to all log in to the system, as user **iurs** and password **iurs2004**.
Then open the Madkit Environment, as follows:

1. Open a terminal.
2. Enter the MadKit directory: *cd madkit_3.1b5*
3. Launch MadKit: *desktop.sh &*

The Graphical environment takes a few seconds to load. Please wait.

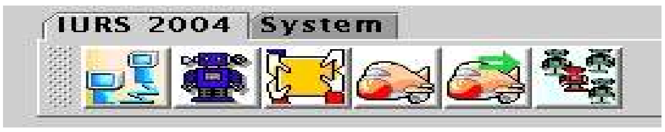
<u>Step 2</u>: Launching the Communicator Agent

When the graphical environment is loaded you can rearrange the frames. You can see 4 different frames or windows, as indicated in the following picture:



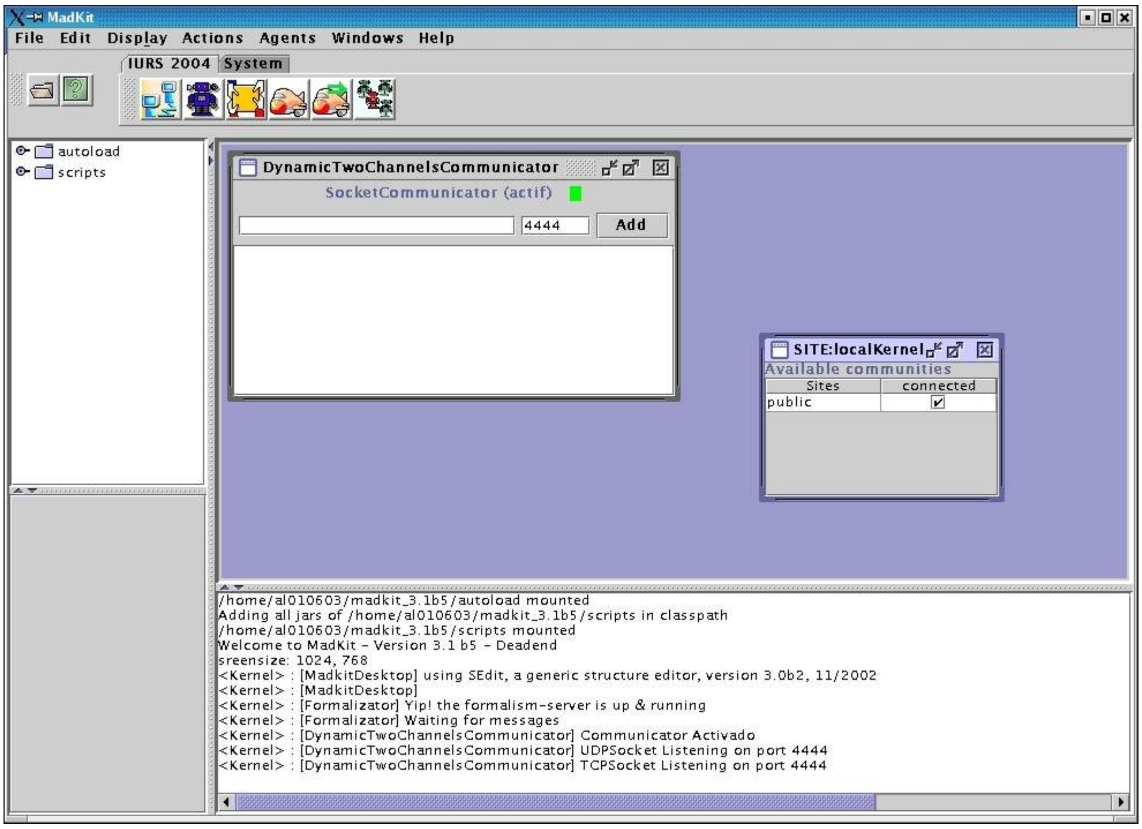
```
/home/al010603/madkit_3.1b5/autoload mounted
Adding all jars of /home/al010603/madkit_3.1b5/scripts in classpath
/home/al010603/madkit_3.1b5/scripts mounted
Welcome to MadKit - Version 3.1 b5 - Deadend
sreensize: 1024, 768
<Kernel> : [MadkitDesktop] using SEdit, a generic structure editor, version 3.0b2, 11/2002
<Kernel> : [MadkitDesktop]
<Kernel> : [Formalizator] Yip! the formalism-server is up & running
<Kernel> : [Formalizator] Waiting for messages
```

You can see 6 buttons in the top side, at the left.



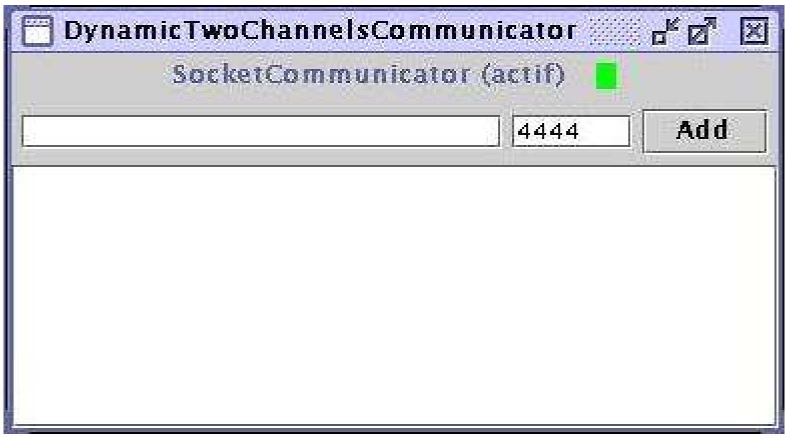
Press the first button, **Communicator**

Now, two windows must be open in the workspace:


MadKit
File Edit Display Actions Agents Windows Help
IURS 2004
System
autoload
scripts
DynamicTwoChannelsCommunicator
SocketCommunicator (actif)
4444
Add
SITE:localKernel
Available communities
Sites
connected
public

```
/home/al010603/madkit_3.1b5/autoload mounted
Adding all jars of /home/al010603/madkit_3.1b5/scripts in classpath
/home/al010603/madkit_3.1b5/scripts mounted
Welcome to MadKit - Version 3.1 b5 - Deadend
sreensize: 1024, 768
<Kernel> : [MadkitDesktop] using SEdit, a generic structure editor, version 3.0b2, 11/2002
<Kernel> : [MadkitDesktop]
<Kernel> : [Formalizator] Yip! the formalism-server is up & running
<Kernel> : [Formalizator] Waiting for messages
<Kernel> : [DynamicTwoChannelsCommunicator] Communicator Activado
<Kernel> : [DynamicTwoChannelsCommunicator] UDPSocket Listening on port 4444
<Kernel> : [DynamicTwoChannelsCommunicator] TCPSocket Listening on port 4444
```

You can see a window labelled as **Dynamic Two Channels Communicator.**



In this window add the IP of your server, following the next table:

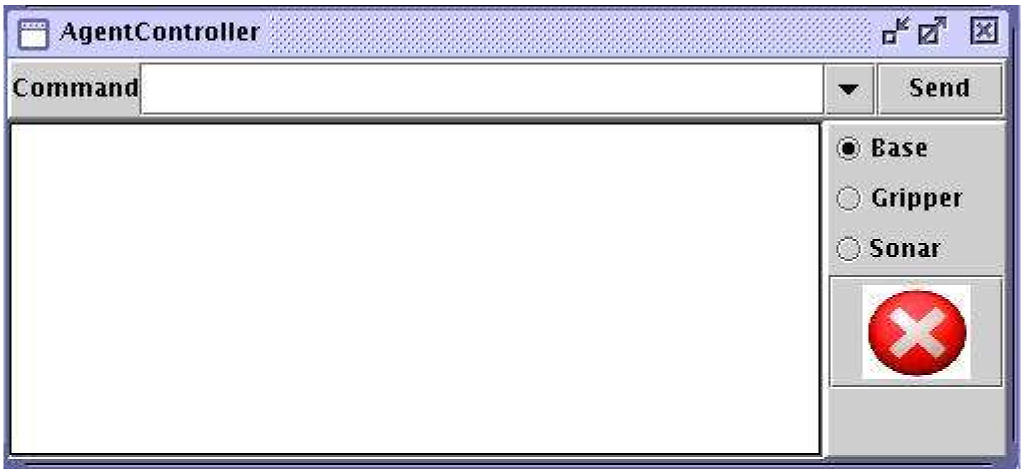
Name	IP
Happy	192.168.69.10
Doc	192.168.69.11
Sneezy	192.168.69.12
Dopey	150.128.82.58

If you have any problem in this Step, please communicate it to the Lab teacher.

Step 3: Executing the Robot's Agent Controller

Now, launch the Agent Controller pressing its button

A new window is opened in the workspace.



Now you can select different commands using the menu, and the robot must execute the different orders. If anything goes wrong, press the stop button. Enjoy!

Part II: Creating a Simple Agent

Step 1: writing the agent's source code

VERY IMPORTANT: if Madkit is running, please quit from the application.

Go into the *templates* directory and edit the file *square.java*. This file contains the basic skeleton of a Madkit agent:

```
public class square extends Agent{
  String robot = "Sleepy";
  public void activate(){
    requestRole("Seven Dwarfs", robot, "square", null);
  }
  public void live(){
    /* Here goes your code */
  }
  public void end(){
    leaveRole("Seven Dwarfs", "Sleepy", "template");
  }
}
```

Important: Change the variable *robot* with the appropriate name of the robot which you connect.

Prior to adding any code, you should learn how the agent sends commands to the elements of the robot. This is done by means of the function **sendOrder**, which sends a string message (the order) to the specified element of the robot:

```
String[] sendOrder(String element, String command)
```

The result of this function is an array with the data returned (if any) by the robot element, the base, the sonar or the gripper.

An example of use of this function is:

```
String[] tmp = sendOrder("Base", "(getPose)");
```

Such example returns an array of three elements (x, y, heading) of the actual position and orientation of the robot.

Now, using the functions listed in the reference at the end of this document, you have to write an agent that drives the robot to follow a square trajectory.

Once your code is ready, you should save the file and quit from the editor.

Step 2: Compiling and running the agent

In order to compile your agent's code, you should simply run *make –f Makefile_square* in the same directory.

Next, you should start Madkit, and run the Communicator agent, as previously, making the connection with your robot.

Now, press the third button ; your program will execute, and hopefully move the robot in the planned trajectory (in case of failure, go back to editing the source).

Part III: creating a *wandering* agent

Follow the same steps of the previous part to create this agent. In this case, the file to open is w*ander.java*. You only need to edit the **live** method in the agent's template. The goal of this agent is to drive the robot in a random walk through its environment without colliding with the surrounding objects.

To program this behaviour, you can follow two points of view. One is to move the robot in the direction towards free space, or the most distant obstacle. The other is drive the robot straight until it detects that an obstacle is near, then the robot moves to the free space.

This is only a suggestion. You can make your own program in the way that you want.

Compile the agent with *make –f Makefile_wander*. To run it, press the button .

Important: some useful functions are implemented in the file wander*.java*:

- **sendOrder(element, command)**: sends the specified command to the robot element.
- **stringToDouble(string_array)**: converts the array of strings into an array of doubles.
- **stringToBoolean(string_array)**: converts the array of strings into an array of booleans.
- **mayor(double_array)**: returns the maximum element of the array.
- **calculate_dir(num_sonar)**: returns the direction corresponding to the specified sonar.

Part IV: creating a *wandering & grasping* agent

Follow the same steps of the previous part to create this agent. In this case, the file to open is *graspWander.java*. You only need to edit the **live** method in the agent's template. The finality of this agent is similar to the previous agent, it must drive the robot in a random way through its environment and when it detects an object between the gripper, it must catch it.
Compile the agent with *make –f Makefile_graspWander*. To run it, press the button .

Important: some useful functions are implemented in the file graspWander*.java*:

- The functions from *wander.java*.
- **stringToInt(string_array)**: converts the array of strings into an array of integers.

Part V: creating a *visual grasping* agent

The finality of this agent is to grasp an object using the vision. To make it possible it is necessary to use or communicate with two new agents, the agent that manage the camera and the agent that communicates with the ACTS server.

The ACTS (Activmedia Color Tracking System) is a software which, in combination with a color camera and frame grabber, lets the applications track up to colored objects. The ACTS agent makes feasible the communication between the whole system and the ACTS server. In the programming reference at the end of this document, you can find the whole set of functions to manage the camera and to communicate with the ACTS system.

To program this agent, you can follow the next steps: First, you have to find the object in the space, to do it, you can make an horizontal sweeping with the camera. Next, you can turn the robot to this direction and refine the seeking to a better calibration of the direction of the robot. The next steep is to move the robot towards the object, but it's necessary to make a visual realignment in each step to not lose the object. When the gripper detects the object between its paddles, you have to stop the robot and to catch the object.

Remember, this is only a suggestion. You can make your own program in the way that you want.

Follow the same steps of the previous part to create this agent. In this case, the file to open is g*raspVision.java*. You only need to edit the **live** method in the agent's template.

Compile the agent with *make –f Makefile.graspVision*. To run it, press the button .

Important: in the file graspVision*.java* there are implemented the same functions as in *graspWander.java.*

Programming Reference

Base class

This class is in charge of the physical operation of the robot. It is responsible of the movement of the robot, as well as to monitor the battery voltage or to check if one of the motors is blocked.

areMotorsEnabled	indicates if the motors are enabled.
disableMotors	disables the motors on the robot.
enableMotors	enables the motors on the robot.
getBatteryVoltage	gets the battery voltage of the robot.
getLeftVel	gets the velocity (metres/second) of the left wheel of the robot.
getPose	gets the global position of the robot. Returns an array containing the global coordinates X and Y, in metres, and the orientation, in radians [X,Y,Th].
getRightVel	gets the velocity (metres/second) of the right wheel of the robot.
getRotVel	gets the velocity (metres/second) of the right wheel of the robot.
getVel	gets the velocity (metres/second) of the robot.
isLeftMotorStalled	indicates if the left motor is stalled.
isRightMotorStalled	indicates if the right motor is stalled.
move <metres>	tells the robot to move the given distance forward/backwards. If the distance is positive, the robot move forward, in other case, the robot moves backwards.
setDeltaHeading <radians>	tells the robot to turn on his axis the given angle. If the angle is positive, the robot turns to left, and if the angle is negative, the robot turns to right.
setRotVel <rad/sec>	sets the rotational velocity of the robot. If the velocity is positive, the robot turns to left, and if the velocity is negative, the robot turns to right.
setVel <m/sec>	sets the velocity of the robot. If the velocity is positive, the robot moves forward, and if the velocity is negative, the robot moves backwards.
setVel2 <m/sec> <m/sec>	sets the velocity of each of the wheels on the robot independently. If the velocities are positive, the wheels move forward, and if the velocities are negative, the wheels move backwards.
stop	stops the robot, by telling to have a translational velocity and rotational velocity of 0.

Gripper class

This class takes charge of the operation of the robot gripper, as well as to monitor its state.

deployPos	this command puts the gripper in a deployed position, ready for use. Puts the gripper at bottom with the paddles open.
getGraspTime	gets the seconds that the gripper will continue grasping for after both paddles are triggered, and stopped.
gripClose	this command closes the gripper paddles.
gripOpen	this command opens the gripper paddles.
gripPressure <seconds>	sets the amount of pressure that the gripper applies. This command sets the seconds that the gripper will continue grasping for after the paddles stop.
gripState	returns the state of the gripper paddles. 0 if paddles are between open and closed; 1 if paddles are open; and 2 if paddles are closed.
gripStop	stops the gripper paddles.
halt	halts the lift and the gripper paddles.
isGripMoving	returns “true” if the gripper paddles are moving.
isLiftMaxed	returns “true” if the lift is either all up or down.
isLiftMoving	returns “true” if the lift is moving.
liftCarry <seconds>	raises the lift by the given seconds.
liftDown	lowers the lift to the bottom.
liftStop	this command stops the lift.
liftUp	raises the lift to the top.
paddleState	returns the state of each gripper paddle. 0 if no paddles are triggered; 1 if the left paddle is triggered; 2 if the right paddle is triggered; and 3 if both paddles are triggered.
sensorsState	returns the state of the gripper’s breakbeams. 0 if no breakbeams are broken; 1 if the inner breakbeam is broken; 2 if the outer breakbeam is broken; and 3 if both breakbeams are broken.
storePos	puts the gripper in a storage position. Puts the gripper at top with the paddles closed.

Sonar class

This class is the responsible to return the appropriate sonar values when these are required.

areSonarsEnabled	returns "true" if the sonars are enabled.
getClosestSonarNum <start_angle> <end_angle>	returns the number of the sonar that has the closest current reading in the given range. The angles must be done in radians.
getClosestSonarRange <start_angle> <end_angle>	returns the closest of the current sonar readings in the given range. The angles must be done in radians. The value returned is the distance from the physical sonar disc to where the sonar bounced back, in metres.
getNumSonar	returns the number of sonars that there are.
getSonarNewVector	gets an array indicating for each sonar disc if his reading is new or not.
getSonarRange <num_sonar>	gets the range of the last sonar reading for the given sonar. "num_sonar" must be between 0 and 7. The sonar range is the distance from the physical sonar disc to where the sonar bounced back, in metres.
getSonarRangeVector	gets an array with the sonar ranges for all of the sonar discs.
isSonarNew <num_sonar>	this command find out if the given sonar has a new reading. "num_sonar" must be between 0 and 7.

Camera class

This class is the responsible to move the camera in an appropriate way and to return the correct values when these are required.

canZoom	returns “true” if the camera can zoom.
getPan	returns the angle the camera is pan, in radians.
getTilt	returns the angle the camera is tilt, in radians..
getZoom	returns the value of the zoom.
getMaxPan	gets the maximum value the camera can pan to, in radians.
getMinPan	gets the minimum value the camera can pan to, in radians.
getMaxTilt	gets the maximum value the camera can tilt to, in radians.
getMinTilt	gets the minimum value the camera can tilt to, in radians.
getMaxZoom	gets the maximum value for the zoom.
getMinZoom	gets the minimum value for the zoom.
haltPanTilt	halts all pan-tilt movement.
haltZoom	Halts zoom movement.
reset	resets the camera to the home position.
pan <radians>	pans the camera to the given direction, in radians.
panRel <radians>	pans relative to the current position the camera by the given radians.
tilt <radians>	tilts the camera to the given direction, in radians.
tiltRel <radians>	tilts relative to the current position the camera by the given radians.
panTilt <radians> <radians>	pans and tilts the camera to the given directions, in radians.
panTiltRel <radians> <radians>	pans and tilts relatives to the current position the camera by the given radians.
zoom <value>	zooms the camera to the given value.
power <0/1>	indicates the camera to switch on or off.

Acts class

This class is in charge to manage the communication with the ACTS for vision tracking.

isConnected	returns “true” if there is a connection with the ACTS server.
receiveBlobInfo	gets the blob information from the connection to the ACTS server.
getNumChannels	returns the number of channels that can be used by the Acts server.
getMaxBlobs	returns the maximum number of blobs that can be tracked in each channel.
getNumBlobs <channel>	gets the number of blobs that the specified channel is tracking in this moment.
getBlob <channel> <blob>	returns the information of the specified blob. The information is returned in an array. In the first position, there is the area of the blob; in the second and third, the coordinates x and y of the center of gravity; the fourth coordinate, the left border of the blob; the fifth, the right border; the sixth, the top border; and the seventh, the bottom border. .