dva96tnn@cs.umu.se

KiKS is a Khepera Simulator

www.kiks.f2s.com

Theodor Nilsson

Theodor Nilsson
dva96tnn@cs.umu.se

Abstract

At Umeå University, real physical robots are being used in education and research to study reactive behavior as a means for stable and noise-persistent control systems. Currently, the most commonly used robot is the **Khepera** robot manufactured by **K-team** (www.k-team.com).

As a master thesis, the author of this paper has created a Khepera simulator (**KiKS** – **K**iKS **i**s a **K**hepera **S**imulator). The resulting software simulates Khepera robots well enough to allow programs that have been written for physical robots to be easily transferred to simulated robots with the behaviors intact.

However, for computing and speed reasons several tradeoffs have been made in all components of the simulator, which could cause certain problems when developing complex behaviors in the simulator and transferring them to physical Khepera robots.

dva96tnn@cs.umu.se

Table of contents

1. INTRODUCTION 1

1.1. ABOUT THIS PAPER 1
1.2. THE KHEPERA ROBOT 2
1.3. THE GOAL OF THE PROJECT 2

2. CURRENTLY AVAILABLE KHEPERA SIMULATORS 3

2.1. RICH GOYETTE: KHEPERA SIMULATOR AND TOOLBOX FOR MATLAB 4.2C (FREEWARE) ... 3
2.2. OLIVIER MICHEL: KHEPERA SIMULATOR VERSION 2.0 (FREEWARE) 3
2.3. CYBERBOTICS: *WEBOTS* (COMMERCIAL PRODUCT) 3
2.4. OLIVER MICHEL AND PR. HEINO IWE: *EASYBOT* (FREEWARE) 4
2.5. CONCLUSIONS 4

3. THE KIKS SYSTEM 5

3.1. THE SIMULATION PROBLEM 5
3.2. SIMULATOR OVERVIEW 7
3.3. SIMULATOR KEY COMPONENTS 7
The interface 7
Simulator engine 9
Khepera motors & movement 10
Proximity sensors 11
Proximity sensor noise model 21
Ambient light sensors 22
3.4. OTHER COMPONENTS 23
Pushable objects 23
Linear vision extension module 23
Proximity sensor colour sensitivity 24

4. RESULTS AND DISCUSSION 25

5. ACKNOWLEDGEMENTS 29

6. REFERENCES 29

APPENDIX A: KIKS USER GUIDE 30

INSTALLING KIKS 30
CALIBRATING KIKS 30
STARTING UP KIKS 31
CREATING THE SIMULATED ENVIRONMENT 32
THE KIKS GRAPHICAL USER INTERFACE 34
CONTROLLING SIMULATED KHEPERAS 35

APPENDIX B: MODELLING WITH NEURAL NETWORKS 37

INTRODUCTION 37
USING NEURAL NETWORKS TO APPROXIMATE A FUNCTION 38
NEURAL NETWORKS IN MATLAB 38
EARLY STOPPING 41

1. Introduction

1.1. About this paper

At Umeå University, real physical robots are being used in education and research to study reactive behavior as a means for stable and noise-persistent control systems. Currently, the most commonly used robot is the **Khepera** robot manufactured by **K-team** (www.k-team.com).

Umeå University has a relatively small number of Kheperas, so the students attending the robotics-oriented course 'Artificial Intelligence II' in the spring of 2000 only had four robots at their disposal, which equals around 6 to 7 students per robot. If Umeå University had had a suitable Khepera simulator available, the students could have done the course assignments on simulated Kheperas, using the real robots only for final testing and fine-tuning of behaviors.

Of course, using a simulator in education and research has other advantages. A simulated Khepera does not break if it hits a surface at high speed. A simulated environment can be controlled – you don't have to worry about the serial cable getting tangled up or obstructing the view of the proximity sensors, which means that you can focus on developing the application. This is especially convenient when genetic algorithms are being used to develop behaviors, since you otherwise need to supervise the learning process and keep these kinds of problems from arising.

As a master thesis, the author of this paper has created a Khepera simulator (from here on referred to as **KiKS** – **K**iKS **i**s a **K**hepera **S**imulator).

In this paper, the KiKS system will be discussed on a theoretical level. Implementation details will not be covered. This paper is organized as follows:

Chapter 1 contains an introduction to the Khepera robot and the goal of the project. In chapter 2, some Khepera simulators are briefly discussed. The third chapter provides a walkthrough of the most important components of KiKS. Finally, results are discussed in the fourth chapter.

A user guide for KiKS is available in Appendix A.

Readers who have little or no knowledge of neural networks will probably benefit from reading Appendix B before reading chapter 3.

1.2. The Khepera robot

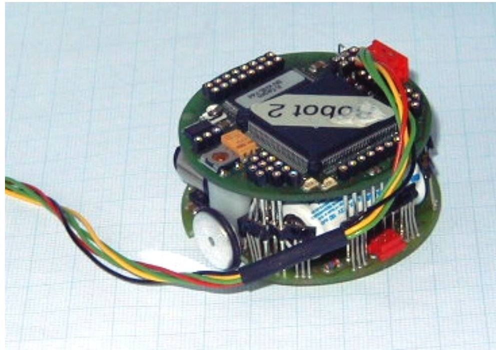


Figure 1

Khepera is a miniature robot with functionality similar to that of larger robots. According to K-team, it is used at more than 350 institutes around the world for research and education.

A Khepera robot is very small, about 55mm in diameter, and has two motors and 8 infrared proximity/light sensors. There are also a large number of extension modules available for the Khepera, for example gripper, camera, and radio communication extension modules.

The Khepera robot can be controlled from a computer by connecting it to the serial port, and for this purpose the Khepera comes with a number of on-board applications that you use to control it. For example, sending the command "D,10,15" followed by a carriage return from the computer over the serial link to the Khepera tells the Khepera to set the left motor speed to 10 pulses/10ms and the right motor speed to 15 pulses/10ms. One pulse/10ms corresponds to roughly 8mm/second. The Khepera also returns information, the command "N" asks the Khepera for the current proximity sensor reading and the Khepera then returns the line "n,v1,v2,v3,v4,v5,v6,v7,v8" where v1,...,v8 is the proximity reading of each of the 8 sensors.

If you wish to use Matlab to control Kheperas, K-team provides the "kMatlab modules" which consists of three commands: kopen.dll, ksend.dll, and kclose.dll. Also included are a number of useful m-files.

1.3. The goal of the project

The goal of this project is to create a simulator that in a realistic way simulates a basic Khepera robot (that is, with no extension modules) connected to the computer. The ambient light sensor capability should only be modeled if there is time, since it is seldom used at Umeå University. Since Umeå University mainly uses Matlab 5.3 to control Khepera robots, a simulated Khepera should be controlled from Matlab, preferably with the exact same commands as if it was a real Khepera physically connected to the computer.

One of the main advantages with simulators is that you can speed up processes in order to save time. Because of this, it is desirable that the simulator can be run both in normal speed and in accelerated mode. Also, visualizing the simulation should be possible.

2. Currently available Khepera simulators

2.1. Rich Goyette: Khepera Simulator and Toolbox for Matlab 4.2c (freeware)

This program would not run under Matlab 5.3. An overview of the documentation and example programs suggests that this program is more suitable as a toolbox for sensor data collection and Khepera monitoring than as a simulator, and even comes with its own commands for serial communication. The simulator part of the program has quite a few limitations, of which the most serious limitation is that a simulated Khepera can only move straight forward/backward or rotate on the spot – that is, the motor speeds cannot be of different magnitude. Also, programs have to be written according to a given template in order to work with the simulator. The author seems to have stopped developing this software in 1997.

2.2. Olivier Michel: Khepera Simulator version 2.0 (freeware)

First of all, this simulator is not intended for use with Matlab. Instead, it provides you with an API that allows you to write behaviors in C and run them on simulated robots as well as on real Kheperas. It seems to be a fairly realistic simulator in terms of sensor and motor modeling, and it supports multiple robots running at the same time. As the author points out in the manual, the simulator has one drawback that can make writing real-time behaviors that work well on both simulated and real Kheperas difficult: at a speed of 10 pulses on both motors, a simulated robot moves exactly 55 millimeters each simulation step, while the real robot moves an unknown distance depending on the speed of the computer, the control algorithm, and the speed of the serial link. This software is probably suitable for development of genetic algorithms since the simplistic sensor and motor model, along with the fact that you not only compile the simulator but also the controlling program, makes the simulator very fast.

2.3. Cyberbotics: *Webots* (commercial product)

Webots is a newer version of "Khepera Simulator version 2.0" by Olivier Michel. It has support for several platforms and is open for any robot with two-wheel steering. It also has a much nicer interface than "Khepera Simulator version 2.0" including 3D visualization. Webots also has a slightly different time system than the previous version, which allows for extremely fast simulations, but there is not much information available about what this time system takes into account to make the simulator better suited for real time use. Considering how fast you can set the simulation to run, it does not seem as if Webots differs much from "Khepera Simulator version 2.0" in this aspect.

2.4. Oliver Michel and Pr. Heino Iwe: ***EasyBot*** (freeware)

EasyBot is a combined ray tracing program and robot simulator, which can be used to create some pretty nice pictures but unfortunately the simulator crashed as soon as an attempt to run a behavior was made. It appears to have a lot of similarities with Webots, with similar time system and the fact that you write the behaviors in C++ and compile them.

2.5. Conclusions

There does not seem to be that many Khepera simulators available, and even fewer that can actually be useful. In particular, the only simulator aimed towards Matlab is obsolete and has a rather clumsy interface that requires that you write the behaviors according to a provided template in order for the simulator to work.

Also, most if not all of the simulators are probably not well suited for time critical real-time applications since they do not appear to take computational or communication speed into account when updating the world.

3. The KiKS system

3.1. The simulation problem

As previously stated, the simulator should be as realistic as possible while still being fast enough to allow for accelerated execution of the program that runs the behavior. That is, if you develop a program that causes a specific behavior on a simulated Khepera, the simulator is said to be realistic if you get the same behavior when you run the program on a physical Khepera and vice versa.

A behavior can be something as simple as "moving forward at a specific speed", or more complex like "finding the way out of a maze". The latter is usually referred to as an *emergent* or *compound* behavior, since it is made out of a lot of individual behaviors ("move forward", "turn left", "turn right", etc.) that interact and result in a more complex behavior. Usually when we speak about behaviors, we refer to emergent behaviors. Ronald C. Arkin formulated these definitions [4:p.24], which can be used to formally describe how behaviors are formed:

An individual behavior: a stimulus/response pair for a given environmental setting that is modulated by attention and determined by intention.

Attention: prioritizes tasks and focuses sensory resources and is determined by the current environmental context.

Intention: determines which set of behaviors should be active based on the robotic agent's internal goals and objectives.

Overt or emergent behavior: the global behavior of the robot or organism as a consequence of the interaction of the active individual behaviors.

Using these definitions, individual behaviors, attention and intention are what the programmer is trying to define. The simulator must provide the programmer with an environment that is as close to "the real world" as possible, in order for the individual behaviors and thus the emergent behavior to work as intended. Obviously, this means that the Khepera sensors and motors must be accurately simulated, but there is another environmental element that can have a heavy impact on behaviors that may not seem obvious at first.

When you control a Khepera from a Matlab program, you are basically switching between doing 2 things:

- Communicating with the Khepera and

- Making decisions on what to do next

And, of course, while you are doing these things time is passing. And as time is passing, the Khepera is usually moving.

Each time you communicate with the Khepera, the time that passes during this operation (from now on referred to as communication time) mostly depends on the baud rate of the serial link and the size of the messages transmitted and received. The lower the baud rate and the longer the message, the more time communication takes.

However, the time it takes to make the decisions on what to do next (from now on referred to as decision time) purely depends on the speed of the computer running the program, along with the complexity of the algorithms used.

So how does this affect the behavior? It depends on the program. If the Khepera is always on the move, and the program reads sensors and adjusts the movement speed of the Khepera, you will see differences in the behavior if you increase the decision time (for example by introducing pause commands or using a slower computer) or change the baud rate. But if the program is constructed in such a way that it makes a decision, then tells the Khepera to move X millimeters (or rather pulses), waits until the Khepera has moved this distance and then tells the Khepera to stop, then reads sensors, and finally starts over, you will probably not notice any difference in the behavior regardless of how many pause commands you introduce in the code, aside from the behavior probably going slower. The keyword is update rate – how far the Khepera moves until a new decision has been made and the suitable command has been transferred over the serial link. This is the reason why the same program running on two different computers sometimes results in different behaviors, and the reason why most currently available Khepera simulators may not be well suited for real-time applications, where timing is an important factor. So the simulator must not only simulate the Khepera well, it must also keep track of communication time and decision time in order for a simulated Khepera and a physical Khepera to show the same behavior.

3.2. Simulator overview

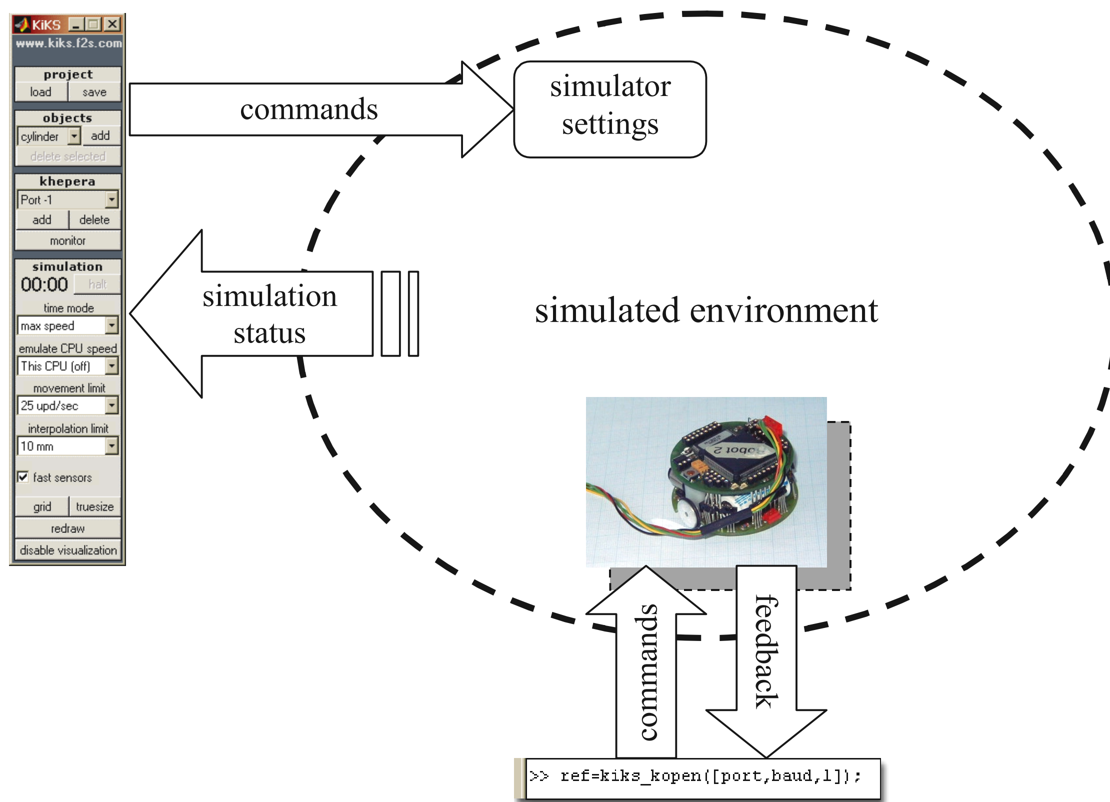


Figure 2

Figure 2 shows an overview of the KiKS system. A GUI is used to control all simulator-specific settings, and Matlab commands are used to interact with simulated Kheperas.

The GUI cannot be used to control simulated Kheperas, aside from placing them in the simulated environment. However, it is possible to monitor the status of simulated Kheperas from the GUI.

3.3. Simulator key components

Now that the most important factors to consider while developing the simulator have been established, the way the simulator is built will be described from the bottom up – that is, by starting with the basics about the user interface and how the simulator engine works and work the way up to how the different sensors have been simulated.

The interface

As previously mentioned, you control a Khepera from Matlab using the three kMatlab commands K-team provides – kopen, ksend, and kclose. In order for KiKS to be easy to use, simulated Kheperas should be controlled in the same way. For this reason, simulated Kheperas are controlled

using three commands that work the same way as the kMatlab commands – kiks_kopen, kiks_ksend, and kiks_kclose.

Of course, the best solution would be to create a new set of commands named exactly like the kMatlab commands, but there does not seem to be a way to accomplish this in Matlab without making it virtually impossible to have the kMatlab commands and KiKS installed at the same time. Instead, a slightly different approach is used.

The syntax for kopen is

```
Port_reference=kopen([ com_port, baud_rate, timeout ])
```

where com_port is the number of the serial port the Khepera is connected to. If the Khepera is connected to serial port #1, com_port is 0, and if it is connected to port #2 com_port is 1.

Kiks_kopen has the same syntax, but doesn't only accept com_port numbers larger than or equal to 0. The idea is simple: simulated Kheperas are regarded as Kheperas connected to negative port numbers. Hence, if you call

```
Port_reference=kiks_kopen([ -1, baud_rate, timeout])
```

KiKS will open up communication with a simulated Khepera. If, on the other hand, you call

```
Port_reference=kiks_kopen([ 0, baud_rate, timeout])
```

KiKS will simply redirect the call to kopen.dll. You then use the Port_reference vector returned by kiks_kopen when you call kiks_ksend and kiks_kclose, just like you would if you were using ksend.dll and kclose.dll, and the simulator will automatically redirect the calls to ksend/kclose if necessary.

So, simply put, the kiks_k* commands work as wrappers for the k*.dll commands, and can be used to control simulated Kheperas as well as physical Kheperas.

A simple GUI is used for controlling all settings directly related to the simulator engine. **Appendix A** contains a detailed description of the KiKS GUI.

Simulator engine

Matlab 5.3 does not support any kind of background processes, which in effect means that only one function can execute at a time. Since kiks_ksend is called each time the user wants to interact with a simulated Khepera (for example change motor speeds or read sensors), kiks_ksend also acts as the core of the simulator.

The simulated world is represented by two 2-dimensional matrixes, of which one is called the **obstacle matrix** and the other is the **color matrix**. The two matrixes are of equal size, and one matrix element corresponds to 1x1 mm. Each element in the obstacle matrix is either 0 or 1, where 1 indicates that there is an obstacle present and 0 indicates that there is no obstacle present. Walls, objects, and Kheperas all contribute to the obstacle matrix, which is updated as the objects move around. For each element in the obstacle matrix, the color matrix tells the simulator what grayscale color that element is. The color matrix is also continuously updated. This may sound like an awkward way to represent data, but allows for very fast collision detection and sensor simulation. Since Matlab is optimized for matrix operations, keeping two matrixes continuously updated instead of just one has had no obvious impact on performance.

Note that for each obstacle matrix element that is 0, the corresponding color matrix element is also 0. This makes the number 1 represent the darkest possible grayscale color (black) and 255 is the brightest color (white). In practice, the obstacle matrix is used for collision detection and the color matrix is used for sensor simulation purposes.

The simulator uses a global clock-variable to keep track of simulated time. Each time kiks_ksend is called, the simulator uses a timer to check how much wall clock time has passed since the last call to kiks_ksend – that is, the decision time. Communication time is calculated using the simulated baud rate, the length of the message sent to the simulated Khepera and the length of the message returned from the simulated Khepera:

```
Communication_time=(message_length*11)/(baud_rate)
```

(Since baud rate is the number of bits transferred per second, and each character in the complete message is transferred using 11 bits, of which 3 are start and stop bits).

The decision time and communication time are then added to the simulated time, and the simulated world is updated. Since the simulator does all its work during the time that would normally be spent waiting for the serial communication, the simulator must work quickly in order for the simulated time to go faster than wall clock time. This is why sensors are only updated when the user

specifically requests it, for example by sending a "read sensor" command to the Khepera. By default, the simulator only moves objects and Kheperas when updating the simulated world.

The KiKS GUI offers a few options for those who want to get a better speedup:

"Emulate CPU speed" multiplies the decision time by a scalar. The scalar is calculated by dividing the time it takes for a slower computer to run the kiks_speedtest.m script by the time it takes for the current computer to run the kiks_speedtest.m script.

"Movement limit" limits the amount of world updates per second the simulator does, so some calls to kiks_ksend does not move the Kheperas and objects in the world which in turn causes the simulation process to go faster.

"Interpolation limit" is similar to "movement limit". It tells the simulator how far a Khepera or object is allowed to move in one step before interpolation of the move is required. The further an object is allowed to move without interpolation, the fewer calculations have to be made and the faster the simulation runs.

"Fast sensors" makes the proximity sensor simulation go significantly faster. More on how this is achieved will be covered later on.

Also, turning off the visualization using the "disable visualization" button makes the simulation go faster since graphics in Matlab is pretty slow.

Khepera motors & movement

If no movement limit is set, Khepera positions are calculated each time kiks_ksend is called, making it the most common task for the simulator. The most obvious way to calculate the new position is to find the center of rotation and rotating the Khepera around this point. This requires quite a lot of calculations, especially if motor speeds are changed often, so KiKS uses a less accurate but significantly faster method.

The distance each wheel travels in T seconds is given by

```
Wheel_distance=T*motor_speed*0.08
```

and the average of the left wheel distance and the right wheel distance (that is, **(left_wheel_distance+right_wheel_distance)/2**) is how far the center of the Khepera is

supposed to move (**mid_distance**). The rotation **theta** of the Khepera (in radians) is also easily calculated by subtracting right_distance from left_distance and dividing the result by the axle length of the Khepera.

When theta is small, the length of the arch from the old position to the new position (mid_distance) is almost the same as the length of a straight line (**L**) from the old position to the new position. Therefore, the movement can be estimated by first rotating the Khepera theta/2 radians and then moving it mid_distance millimeters in the new direction, and finally rotating the Khepera theta/2 radians again to get the final position.

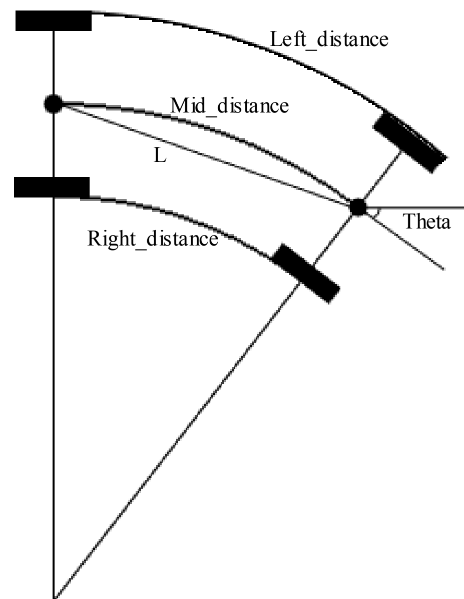


Figure 3

If theta is large, the error in the estimation becomes quite large. KiKS handles this by interpolating the movement in the intervals specified by "interpolation limit" in the KiKS GUI. Interpolation is required in any case to keep the Khepera from jumping over obstacles, and would have to be performed regardless of which method used to calculate movement.

When a simulated Khepera collides with an obstacle, it stops moving but keeps rotating if the motor speeds differ from each other. The wheels keep spinning at around 10-15% of the motor speeds. The wheels will not, however, spin from accelerating the motor speeds too fast since this phenomenon seems to occur very rarely.

Proximity sensors

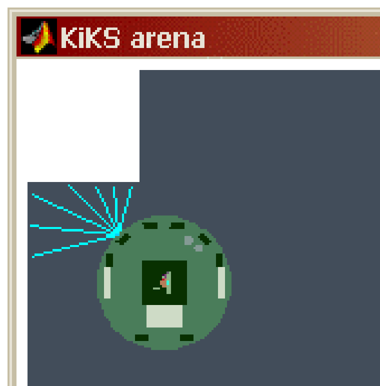


Figure 4

The Khepera has 8 light sensors. These can be used to measure ambient light levels but more importantly are able to send out pulses of light and measure the amount of light reflected by obstacles – thereby also functioning as proximity sensors. Each sensor has a field-of-view of about 120°, and a number of factors affect the sensor readings. These factors include the color and material of the obstacle, ambient light level, the shape and position of the obstacle, individual differences in each sensor [1], and even a certain amount of random noise. Each proximity sensor has been

simulated using a neural network, and to simplify things a bit it was decided that all obstacles should be made out of white paper. Under these circumstances each sensor can detect obstacles about 5cm away, according to the Khepera user manual.

It has been considered important to include the large field-of-view in the sensor model, while keeping the amount of calculations needed to collect input data as low as possible. In the simulator, each sensor therefore sends out seven rays in evenly spaced intervals. Each ray is sent out a maximum of 50mm (or less, if it hits an obstacle). The final length of each ray is then inverted, so if x1=x2=…=x7=0 then all seven rays were sent out 50 millimeters without hitting an obstacle.
If, on the other hand, x1=x2=…=x7=50 then all seven rays immediately hit an obstacle.

These seven values are then to be used as input to a neural network that tries to calculate what the proper sensor reading should be.

Gathering sensor data

To be able to simulate something using neural networks you need training data and test data. The data should consist of input values and their corresponding output value(s). In this case, the input values are the measured distances x1,…,x7 and the output values should be the sensor reading in each situation – a number between 0 and 1023.

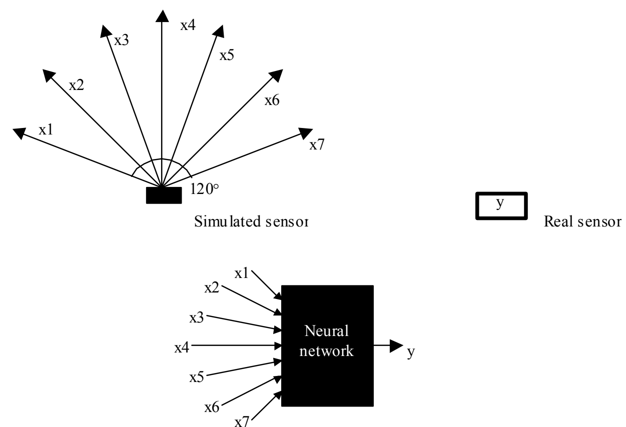


Figure 5

A prototype of the simulator was used to collect the input data and a real Khepera was used to collect the output data by placing a simulated Khepera in the simulated environment and the real Khepera in the real environment *in identical situations.*

Using a simple Matlab script, data was collected for a total of 10188 situations. After removing duplicate entries and zero entries (x1-x7=0 and y=0) around 3000 samples per sensor were used to train and test the neural networks.

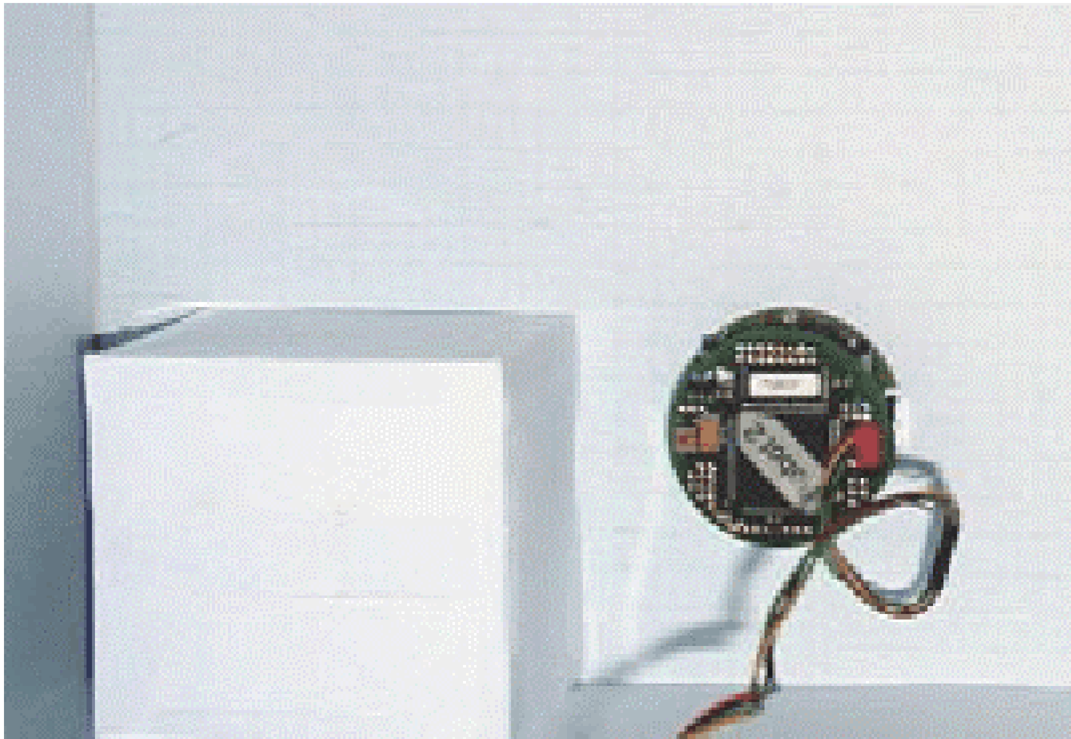


Figure 6

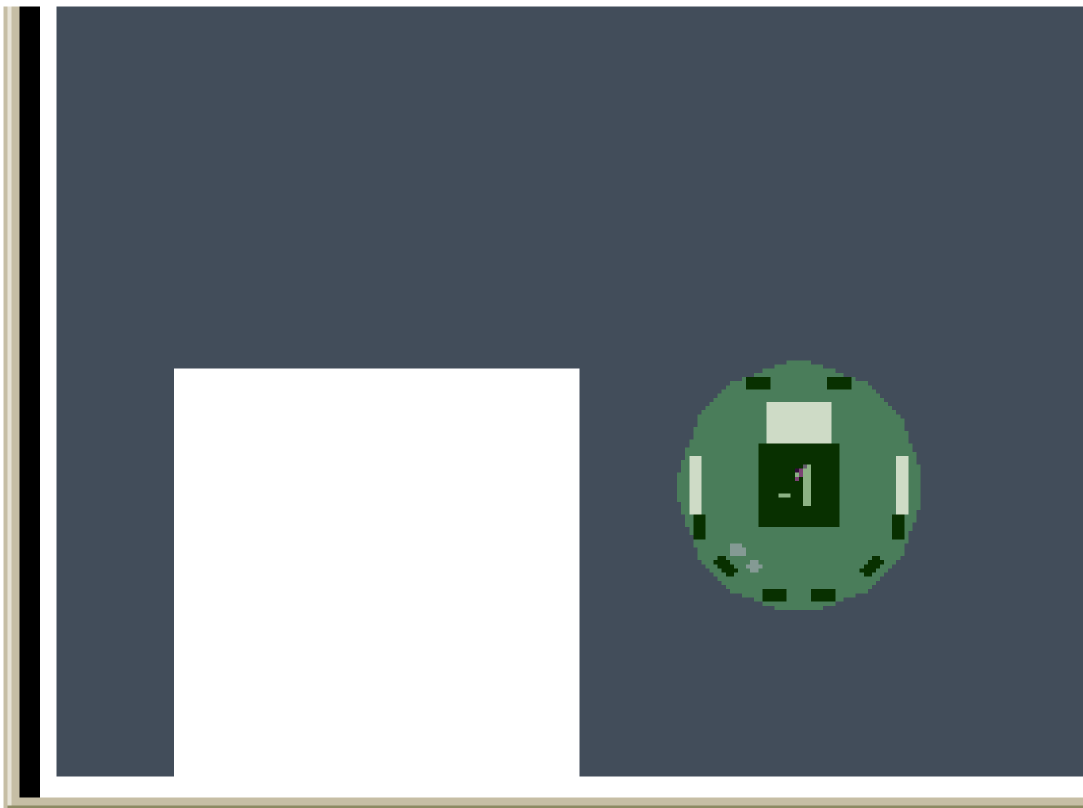


Figure 7

An example setup for collecting sensor data.

How do neural networks work?

In short, neural networks consist of inputs, outputs, and hidden layers. Each hidden layer contains one or more neurons, and each neuron receives its' input from all neurons in the preceding layer (and thus sends its' output to all neurons in the following layer). Neurons in the same layer are not connected. Each neuron uses some function g(x) on the weighted sum $\sum(w_j i_j)$ (for j=1,...,number of neurons in the preceding layer) of the received inputs and sends the resulting value $g(\sum(w_j i_j))$ to the next layer. Appendix B provides a more thorough description of neural networks.

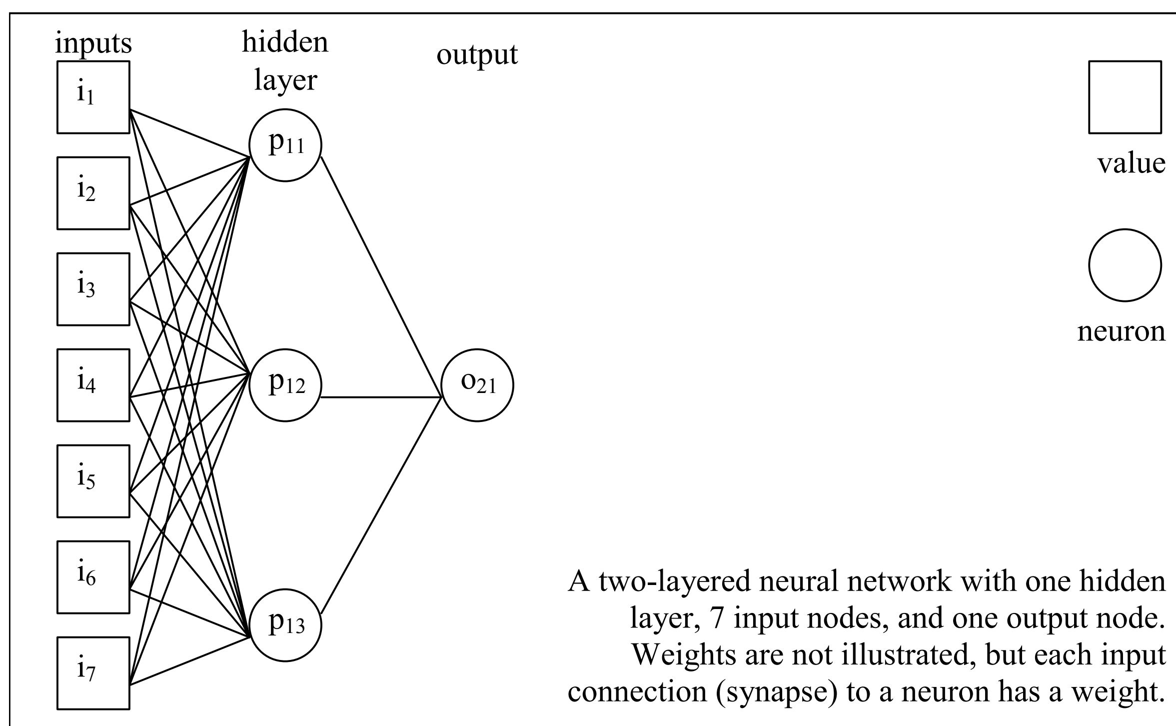


Figure 8

Training the neural networks

The general idea behind neural networks is that if we know a set of input and output values (called the training set), the neural network can be adapted to associate each input with its output only by adjusting the synaptic weights. This process is called *training the network*. The most common training method is back-propagation [5:p.578].

As mentioned earlier, the neural network for each sensor should have 7 input neurons and 1 output neuron. The output should be a positive number between 0 and 1023, which is why the linear transfer function should be appropriate in the output neuron.

7 input neurons, 0 hidden neurons, 1 linear output neuron

A Matlab script that divided the data for sensor #1 into one training set and one test set, constructed five networks each consisting of 7 input neurons, no hidden layer and one linear output neuron, trained the networks with early stopping (where the test set was also used as validation set) for a maximum of 100 epochs each and finally picked the net that showed the smallest maximum and mean errors in the test set was used to train a neural network. This was the result:

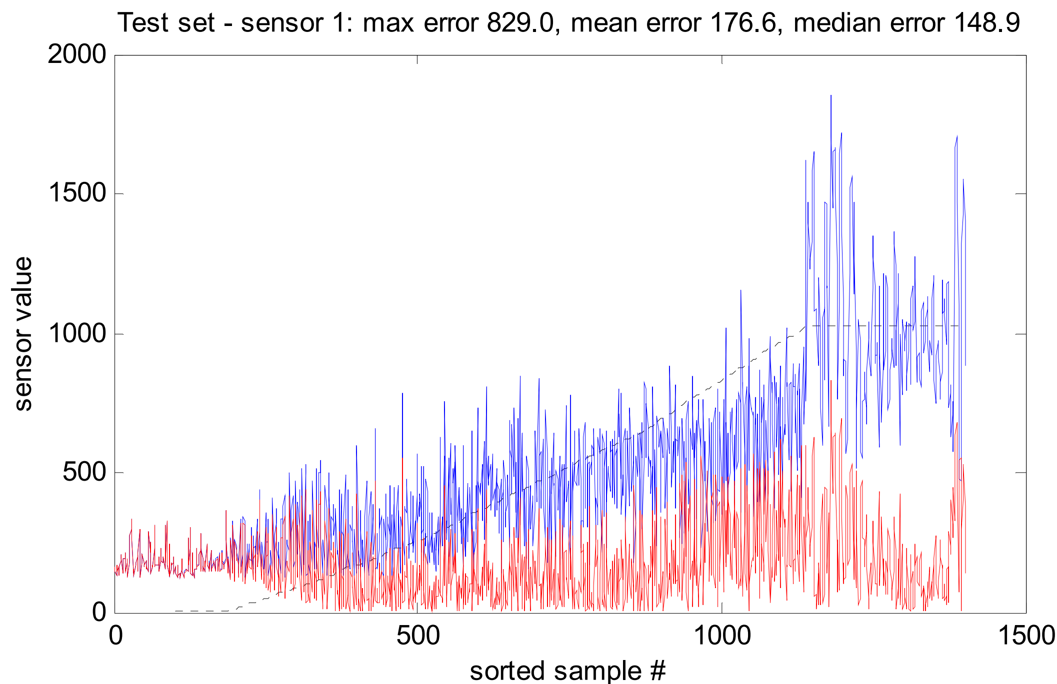


Figure 9

The dotted black line is the expected output for each sample (i.e. the value collected from the real Khepera) and the blue line is the actual neural network output. The red line (at the bottom) is the absolute error. The results are sorted so the expected output is increasing from 0 to 1023.

Obviously, a hidden layer had to be introduced or the output neuron had to be modified. Changing the transfer function in the output neuron did not have any positive effects.

Figure 10 is a plot of the first 250 samples of the test set, which might give a better idea of how well the neural network performed. In figure 11 error distribution is shown. In this case, large errors were not uncommon.

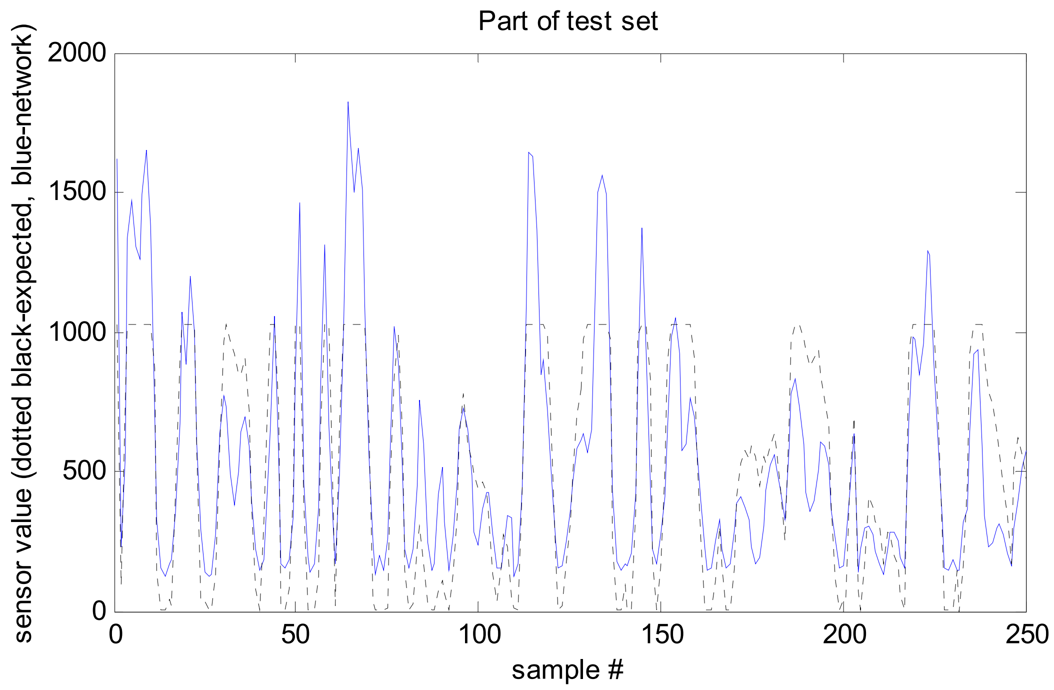


Figure 10

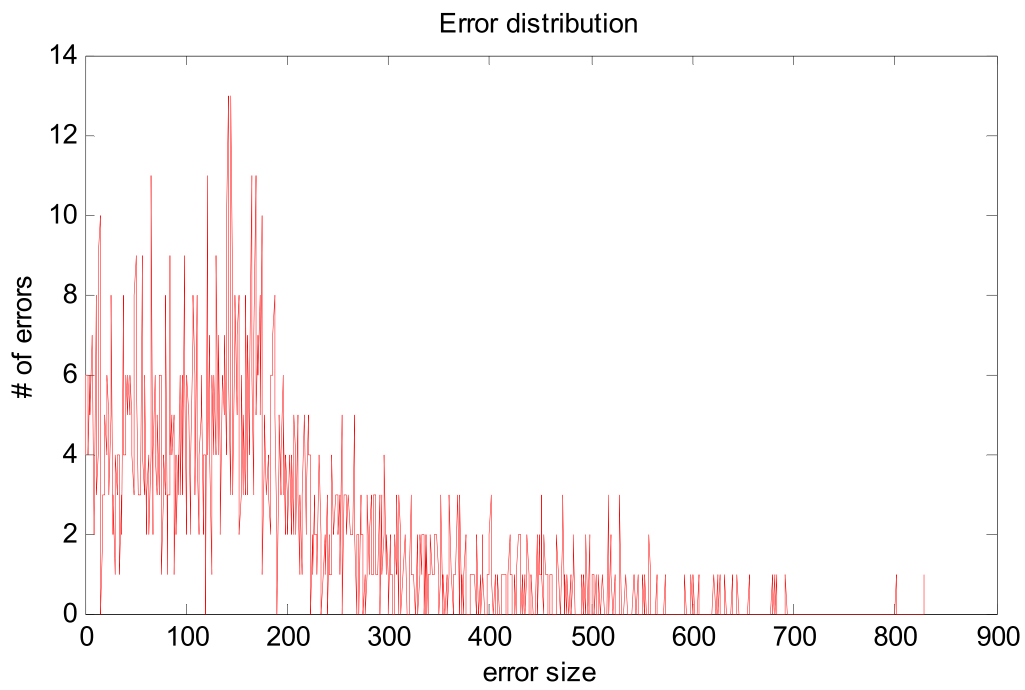


Figure 11

7 input neurons, 7 linear neurons in 1 hidden layer, 1 linear output neuron

Introducing a hidden layer with 7 linear neurons improved the performance a lot –the median error on the test set dropped from 154.4 to 76.5. At this point it started to become clear that getting the error on the test set down to close to 0 was not a realistic goal. In theory, it should be possible to get a very small error on the training set by making the neural network large enough and/or using different transfer functions in the neurons. But the larger and more complex a neural network is the slower it gets. Having a huge, complex neural network would thus make the simulator too slow to be of much use.

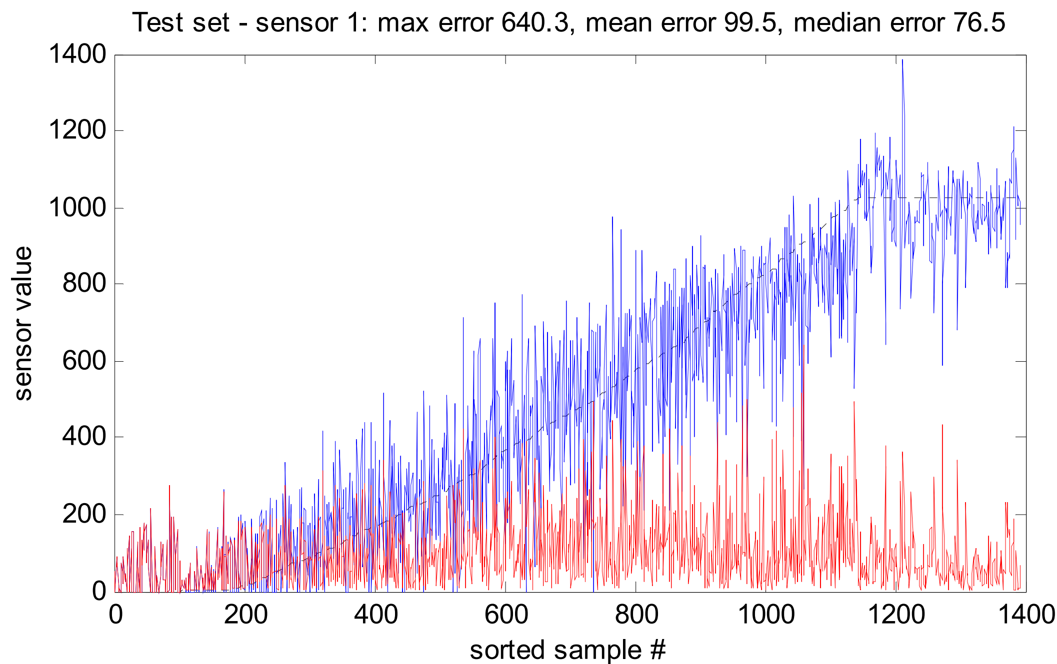


Figure 12

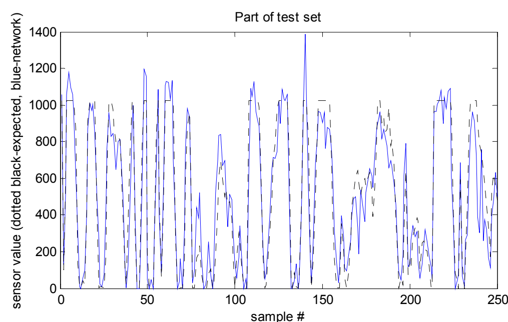


Figure 13

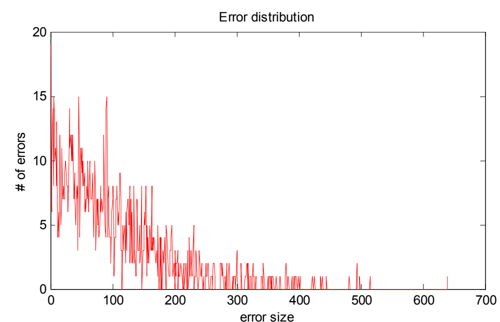


Figure 14

7 input neurons, 13+7 neurons in 2 hidden layers, 1 linear output neuron

Larger neural networks with varying transfer functions were evaluated. After countless hours of testing different neural networks, it was decided that a network with 2 hidden layers, whereof the first consists of 13 neurons with the hyperbolic tangent sigmoid transfer function and the second one consists of 7 linear neurons, has the best balance between execution speed and error size. In particular, larger networks did not achieve better performance on the test set than the 13+7 network but showed signs of overfitting on the training set, which suggests that more than 7 variables for describing the environment are needed to gain better results.

Figures 15, 16 and 17 are the different graphs for the 13+7 neural network trained with data from sensor #1. Neural networks for sensor #2,…,#8 were trained using the same type of network, with similar results.

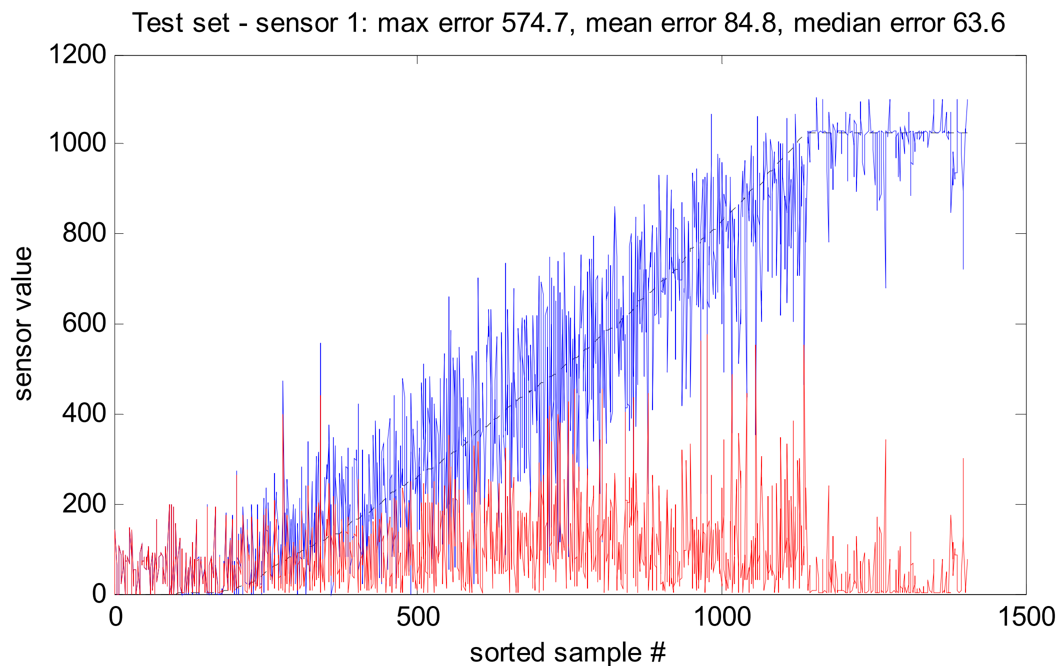


Figure 15

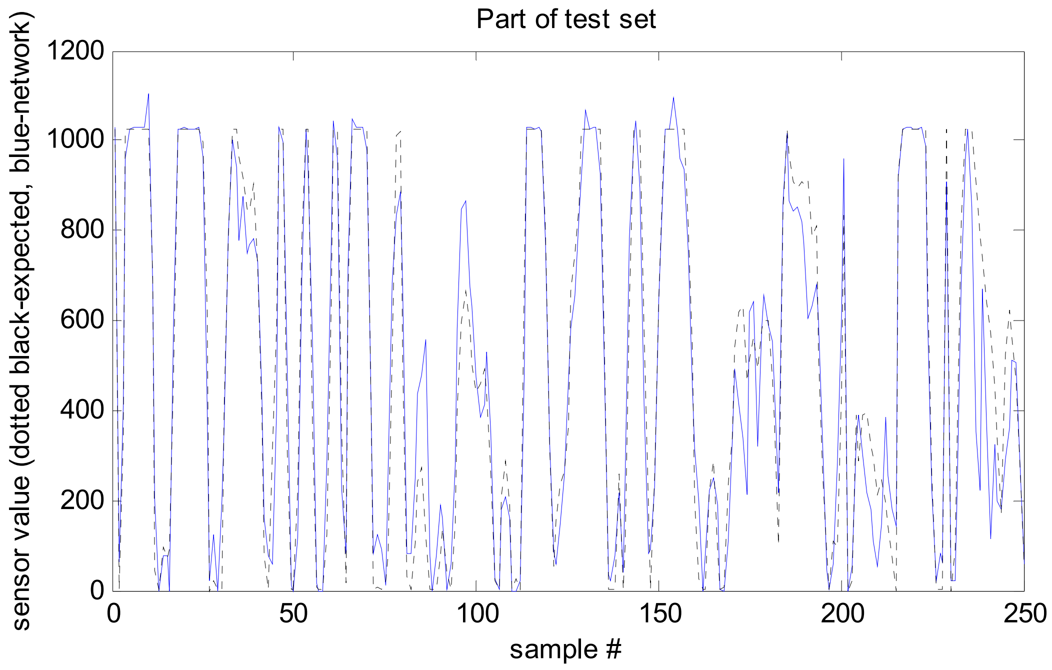


Figure 16

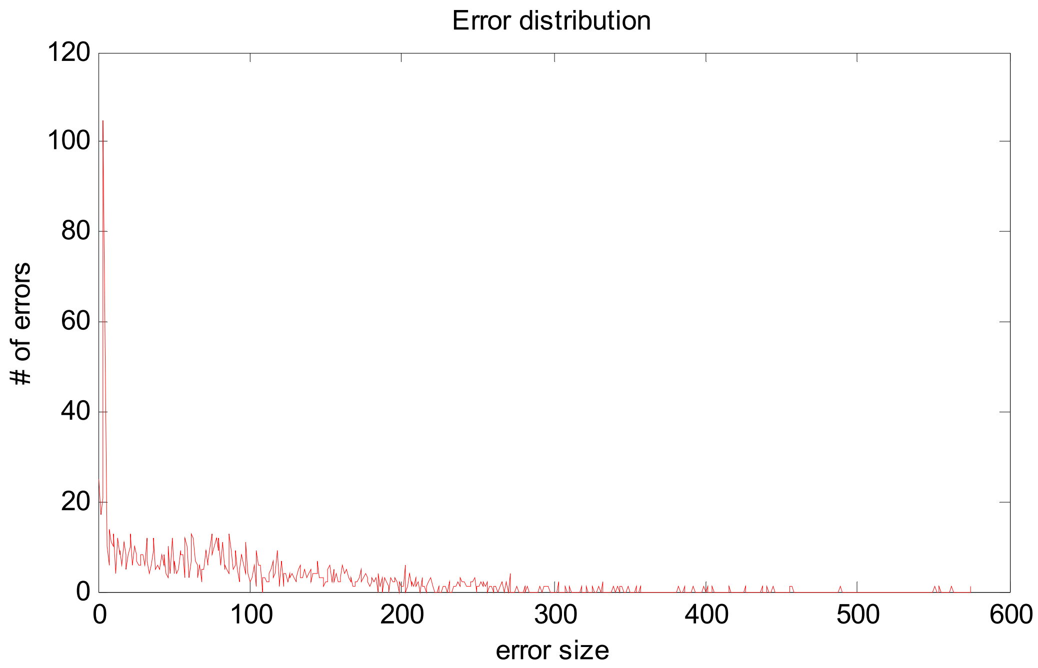


Figure 17

The "Fast sensors" setting

When the "fast sensors" checkbox in the KiKS GUI is checked, KiKS looks up the sensor value in a pre-calculated 7-dimensional array instead of calculating it with the appropriate neural network. Looking up sensor values in the array is very fast, but if all possible integer values for each of the 7 variables would have been stored in the array (51^7)*4=3,5896 terabytes of space would have been required.

Instead, the array only contains pre-calculated sensor values for variable values that are multiples of 10 (that is 0,10,20,30,40,50) which only requires (6^7)*4=1,1197 megabytes of space. This means that during simulation the seven measured distances are rounded off to the nearest multiple of 10, and the corresponding sensor value is then looked up in the array.

Note that all sensors use the same array when KiKS is running in "fast sensors" mode. For memory saving reasons, the values in the array are the average of the values calculated by each neural network during pre-calculation. Additionally, the sensor sample data is used instead of neural network calculations whenever possible – why use neural networks to estimate an already known sensor value?

Proximity sensor noise model

By placing a Khepera close to an obstacle and observing the variation of the proximity sensor response over time, you can calculate the median sensor value `MS` and the relative error

```
max(abs(sensor_values-MS))/MS
```

Figure 18 shows how relative error gets smaller as sensor readings get higher. Note that in reality, there is not a perfect straight line from 0.25 to 0.1. The shape of the curve depends on the sensor.

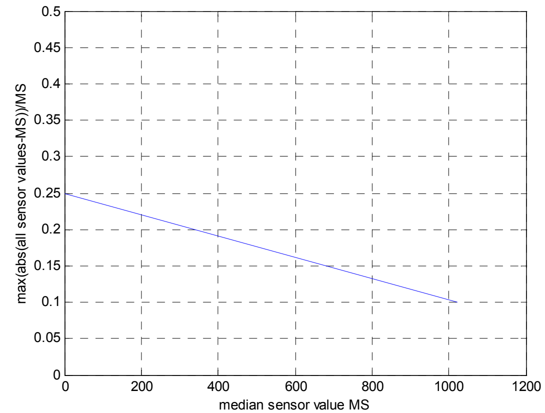


Figure 18

However, the tendency in the relative error is that when readings are small, somewhere around 25% of the sensor reading appears to be noise and as sensor readings become larger, the noise decrease towards 10%.

Also, large errors are significantly less common than small errors.

To model this uncertainty in proximity sensors, noise is added to the calculated sensor value:

$$E = 5$$

$$noise = 0.25 - \left(\frac{sensor_value}{1000}\right) * (0.25 - 0.10)$$

$$final_sensor_value = sensor_value * (1 \pm noise * random(0.0, 1.0)^{E})$$

Where ± is randomly chosen with equal probability.

The E constant makes sure that large errors are much less common than small errors, since the maximum relative error is multiplied by a random number between 0.0 and 1.0 to the power of E.

Ambient light sensors

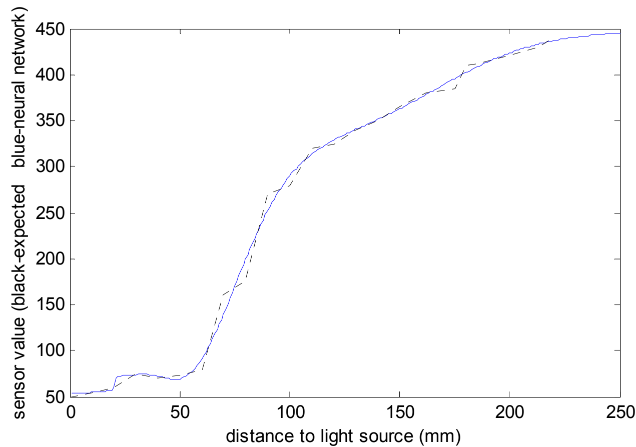


Figure 19

On pages 9 and 10 in the Khepera user manual [1], two graphs show how measurements of ambient light vary depending on the distance and angle to a 1-watt light source. Since simulating light sensors has not been considered important for this project, these two graphs have been the only sources of information when creating the light sensor model.

First, a neural network was created and trained to produce roughly the same curve as the graph on page 9 in the Khepera user manual, as shown in figure 19.

This neural network takes as input the distance **d** from the sensor to the light source and returns the value $\mathbf{R_0}$ the sensor should return if the angle **a** between the sensor and the light source had been 0°.

As the graph on page 10 of the Khepera user manual shows, the sensor reading is heavily dependant of the angle between the sensor and the light source. To incorporate this into the sensor model, a value proportional to a falloff value **F** is added to R_0 to get the final sensor value **R**.

$R=R_0+F*(500-R_0)$

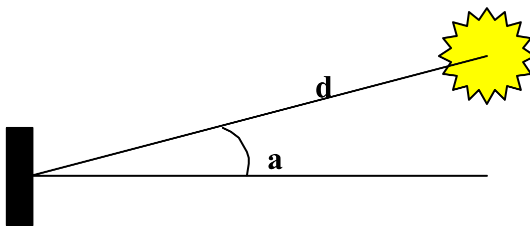


Figure 20

where F is 1-cos(a) if the angle between the sensor and the light source is less than 90°, or 1 if the angle is equal to or larger than 90°.

If more than one light source is available in the simulator, the one that gives the strongest sensor response (i.e. lowest value) is chosen. Five light sources at the same spot will thus have the same effect on sensors as one light source.

3.4. Other components

To make the simulator a bit more useful, especially for developing "Khepera soccer" behaviors, support for pushable objects and the linear vision extension module was added to the simulator. Also, the proximity sensor model has been adjusted to include surface color sensitivity. Since these components were not originally part of the project and have not been thoroughly tested, only a brief explanation of them will be given.

Pushable objects

There are two types of pushable objects in the simulator: cylinders and balls. The names are chosen to somewhat reflect their physics. Cylinders do not have mass and speed and only move when a Khepera or another cylinder is pushing them. A cylinder simply tries to "move out of the way" when another object pushes it.

Balls, on the other hand, have a more realistic physics model. They have mass and speed, and will bounce according to collision formulas found in Vince: "Virtual Reality Systems" [3] if they hit an obstacle. Balls require a lot more calculations than cylinders, which is why you probably will not want to use more than 2 or 3 balls at once. Cylinders, however, have practically no computational cost unless they are being pushed and are therefore well suited for "room cleaning" behaviors and other applications where you want to have 10 or more objects in the environment at once.

Note that balls and cylinders do not work well together, since the bounce/push mechanisms do not work when a ball and a cylinder collide.

Linear vision extension module

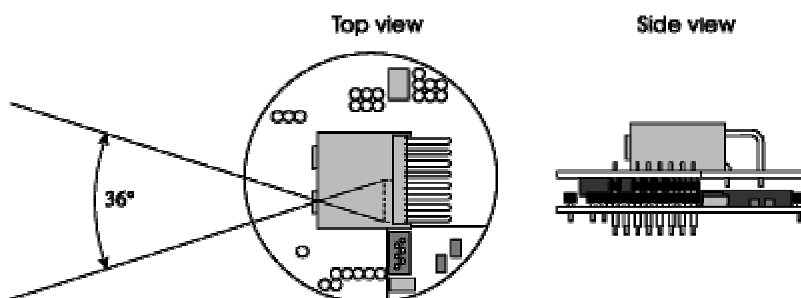


Figure 21

One of the most commonly used extension modules for Khepera robots is the K213 vision turret. It is a linear vision camera, with a resolution of 64x1 pixels and 256 gray levels per pixel. Figure 21 is taken from http://www.k-team.com/robots/khepera/K213.html.

KiKS simulates the K213 turret by sending out 64 rays, one for each camera pixel, in directions from -18° to 18° relative to the angle the Khepera is facing, starting 5 cm in front of the Khepera. When a ray hits an object, the corresponding pixel is assigned the grayscale color the object should

have according to the color matrix. If the ray was sent out more than 25 cm, the corresponding pixel is heavily blurred to incorporate the limited view distance into the model.

When all pixels are calculated, noise and falloff is added to the image. Also, the picture is blurred if the Khepera is moving.

The amounts of noise and falloff have been chosen with the example images on page 16 in the K213 user manual [6] as guidance.

Proximity sensor colour sensitivity

As previously explained, when creating the proximity sensor model the color of all simulated obstacles were assumed to be white and the neural networks were trained using data from measurements of obstacles made out of white paper. However, dark surfaces reflect less light than bright surfaces and in order to achieve a more realistic sensor simulation, a simple modification of the sensor model was made to reflect this.

For each of the seven rays x1,...,x7, after measuring the inverted distance **x** to the obstacle x is adjusted depending on the color of the obstacle:

```
x=x-(x*0.5)*((255-obstacle_color)/255))
```

So the darker the obstacle is, the smaller the inverted values of x1,...,x7 are. Simply put, dark obstacles will appear to be farther away from the sensor than they in fact are, and because of this the value returned by the sensor will be smaller than if the color of the obstacle had not been considered at all.

4. Results and discussion

Recall from the project specification that if you develop a program that causes a specific behavior on a simulated Khepera, the simulator is said to be realistic if you get the same behavior when you run the program on a physical Khepera and vice versa.

To test whether this condition holds when transferring a behavior from a physical Khepera to a simulated Khepera, some real-time behaviors were tested in the simulator.

Not only did simple behaviors, such as "follow wall" and "avoid obstacle", work as expected but also more complex behaviors like "room cleaning" (finding and pushing objects towards walls) as shown in figure 22. The dotted line shows how the Khepera has moved over time.

The behaviors were also modified in various ways, and it was observed that the modifications had similar effects on behavior on simulated as well as and real Kheperas.

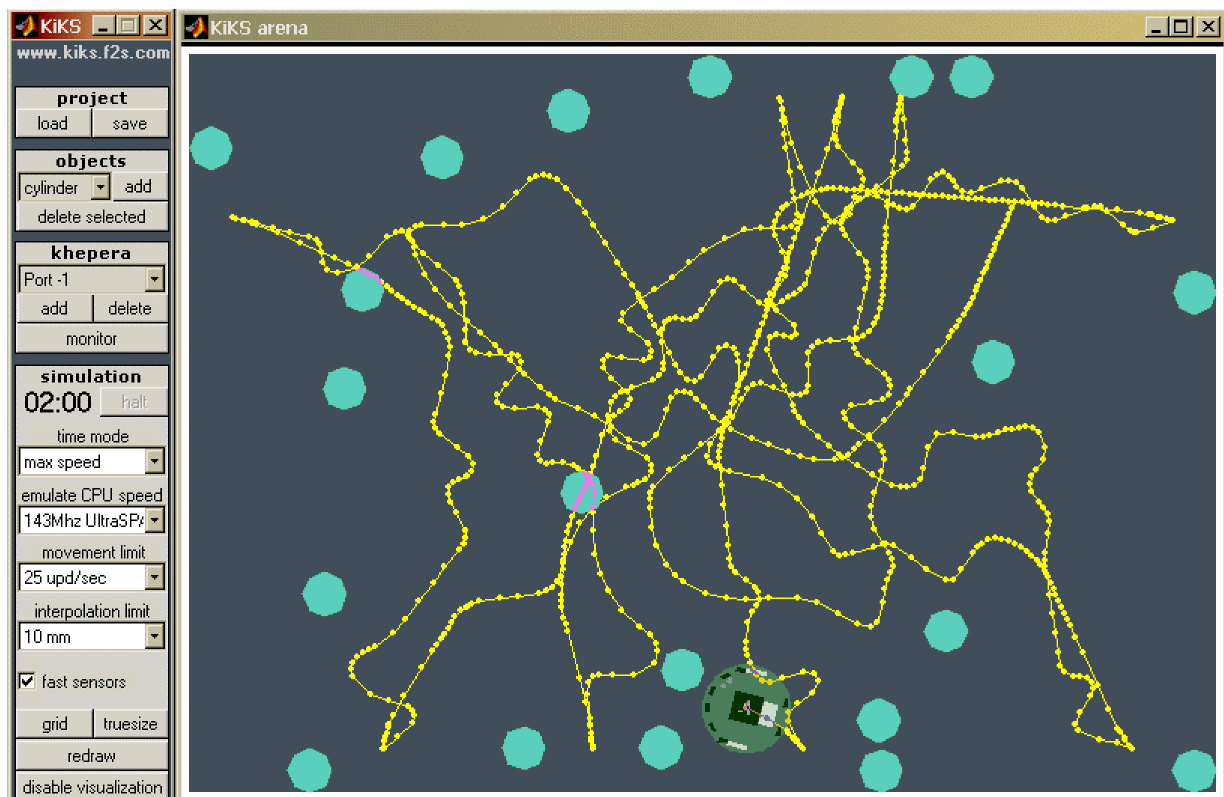


Figure 22

As a second task, a program that is a complete implementation of a Khepera soccer behavior was tested in the simulator. Students attending the course "Artificial Intelligence 2" in the spring 2000 developed the program for use on a real Khepera. It should be noted that this program was the winning submission in the first Khepera soccer tournament at the university, and that the author of this paper did not participate on the course or development of the program. In order for the program to work with KiKS, all instances of "kopen", "ksend", and "kclose" in the program were replaced by "kiks_kopen", "kiks_ksend", and "kiks_kclose".

Also, since the ball physics in KiKS still had a few bugs, all "try-catch" statements in the program were removed to prevent any run-time error that might occur in the simulator from being handled by the soccer program. The program algorithms, including all individual behaviors, were not modified in any way.

Then, a suitable soccer arena was drawn in grayscale .tif format and imported to KiKS using the newly written "kiks_tif2arena" command. A ball was added to the arena, a simulated Khepera was placed in the upper right corner of the arena, and the program was started.

The simulated Khepera located the ball and the goal, moved closer to the ball, made a few twisting and jerking moves and kicked the ball in the entirely wrong direction. When the ball hit the wall, the simulator crashed.

The bug that caused the crash was located and fixed. However, the reason why the "orbit around ball" behavior didn't work was not to be found in the simulator, but proved to be a known bug in the soccer program. According to students participating at the tournament, the same problem had occurred on the real Khepera when the soccer tournament was held, but since programs were run one at a time (making it more of a soccer penalty tournament) and each soccer program was allowed to run for five minutes, this program had still been able to score enough goals to win.

Running the behavior for a few minutes shows that this also holds in the simulated environment.

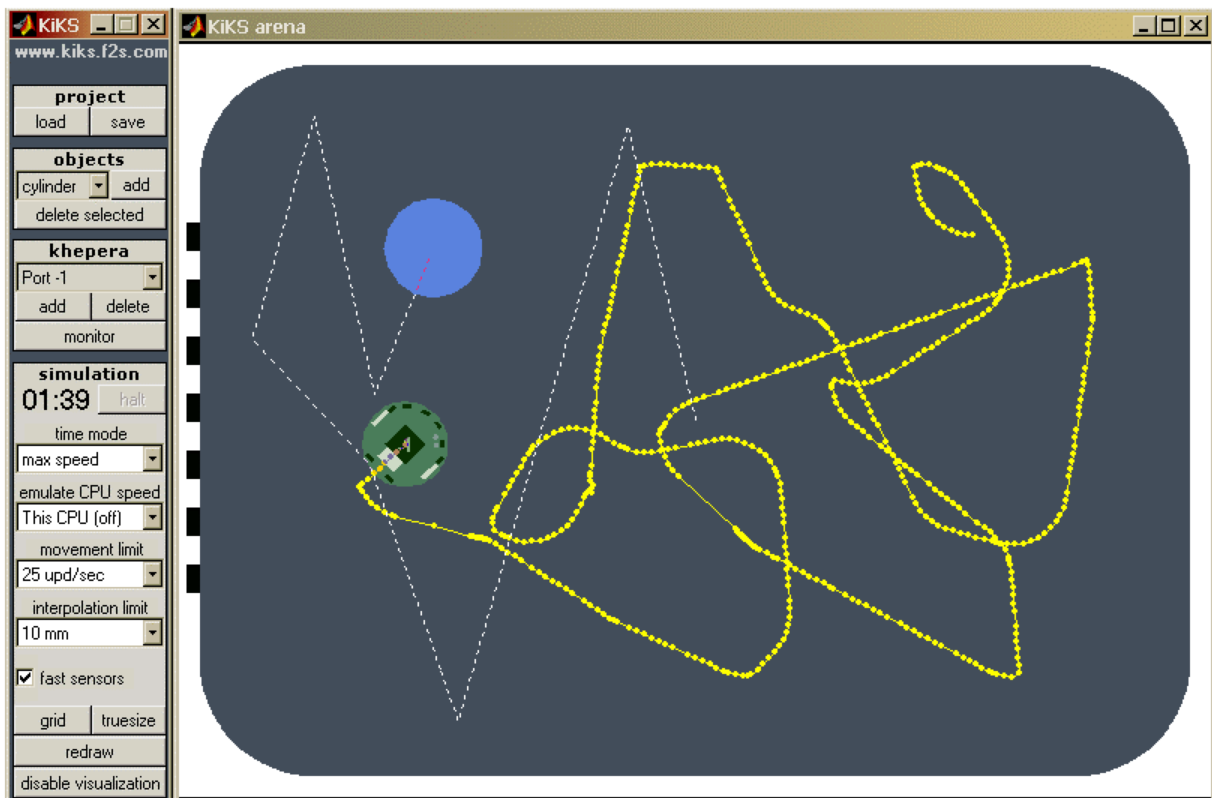


Figure 23

In figure 23, the program is run on the final version of KiKS. After first kicking the ball straight up into the upper wall, the Khepera manages to score a goal on the second try. The dotted line shows how the Khepera has moved, and the thinner dashed line shows how the ball has moved.

Because of time constraints, no complex behavior has been completely developed on a simulated Khepera and then tested on a real Khepera. It would be particularly interesting to examine if KiKS can be used for genetic algorithms, and how well a genetically trained behavior would “survive” the transition from the simulator to the physical world.

It is a fair assumption that behaviors that in some way rely on the simplistic physics in KiKS will not work well on a real Khepera. For example, if a simulated Khepera traveling straight forward collides with a corner, it will stop completely, not rotating at all, and the wheels will keep spinning at only 10%-15% of the motor speeds. The collision can be quickly detected if the 'K' command is frequently used to see if the wheels have been forced to slow down (indicated by the lE and rE variables). However, a real Khepera will most likely not stop in a situation like this, but instead slide and rotate depending on the angle of collision. If an internal map is used to keep track of the

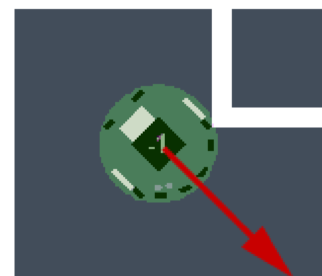


Figure 24

position of the Khepera and this difference between the simulator and the real world is not kept in mind, problems could arise since a simulated Khepera does not lose orientation when colliding in the same way as a real Khepera does.

Compared to other Khepera simulators, KiKS has its biggest drawback in speed. A simple "follow wall" behavior usually runs at between 150% and 300% of wall clock time, depending on the speed of the computer and simulator settings. If the simulator was rewritten in C and made as a standalone application, it could probably run a lot faster. There is a compiler included in Matlab, which converts Matlab functions to C code and compiles them to something called mex files (on the Windows platform, mex files are ordinary dll files). However, no speedup has been gained by compiling KiKS – on the contrary, when compiling kiks_ksend and all referenced functions into one dll file, the simulator goes 3-4 times *slower* than when running KiKS in standard interpreted mode. It is not clear why this happens, but it most likely has something to do with the fact that the compiled code still has to run in the Matlab environment to be able to use neural network functions and certain matrix functions. This is also the reason why the Matlab compiler cannot create a standalone application out of KiKS.

5. Acknowledgements

I wish to thank my thesis advisor Thomas Hellström for his help and feedback during this project. Also, thanks to K-team for helping me promote this software on their home page and newsgroup.

6. References

[1] K-Team: *Khepera user manual*, Lausanne, 1999 (available for download at http://www.k-team.com/download/khepera/documentation/KheperaUserManual.pdf)

[2] The Mathworks: *Matlab Help desk*, 5/31/2000, http://www.mathworks.com/access/helpdesk/help/helpdesk.shtml, 9/19/2000

[3] John Vince: *Virtual Reality systems*, University press, Cambridge, 1995

[4] Ronald C. Arkin: *Behavior-based robotics*, The MIT Press, Cambridge, 1998

[5] Stuart J. Russel, Peter Norvig: *Artificial Intelligence: A modern approach*, Prentice Hall International Editions, London, 1995

[6] K-Team: *K213 vision turret user manual*, Lausanne, 1999 (available for download at http://www.k-team.com/download/khepera/documentation/K213Manual.pdf)

Appendix A: KiKS user guide

KiKS is an abbreviation for "**K**iks **i**s a **K**hepera **S**imulator". The program simulates one or more Kheperas connected to the computer via serial link in a very realistic way. Simulated Kheperas are controlled in the same way as real, physical Kheperas. A complex time system that even takes the speed of the simulated serial link into account makes sure that KiKS is well suited for time critical real-time applications, a feature that as of the writing of this document is not found in other Khepera simulators available.

KiKS is written completely in Matlab, and requires Matlab r11.1 (or newer versions). KiKS can be downloaded from http://www.kiks.f2s.com/dl_kiks.php.

In order for the demo behaviors included with KiKS to work, you also need to have the kMatlab modules from K-team installed.
They can be downloaded from http://www.k-team.com/download/khepera.html.

The kMatlab package contains a set of three core commands (kopen.dll, ksend.dll, kclose.dll) that are used to communicate with real Kheperas from Matlab. Also, a number of Matlab scripts that takes care of message parsing and error handling when controlling Kheperas are included, and although they are not in any way required for KiKS to work, these scripts are used in the demos simply because they are very handy.

Note that in this document, text written using `fixed width font` are Matlab commands.

Installing KiKS

Unpack the KiKS zip file to a location of your choice. A "kiks" directory will be created.

Start up Matlab.

Change directory (using the "`cd`" command) to the kiks directory.

Run `kiks_setup` to add the 'kiks\' and 'kiks\system\' directories to the Matlab path (you can also do this manually using the File/Set Path... menu in Matlab).

Calibrating KiKS

In order for you to get the most out of this simulator, it needs to run a few tests on your computer.

- In Matlab, change directory to kiks\.

- If you have a Khepera connected to the computer, run `kiks_calibrate(port,baud)` where 'port' is the serial port the real khepera is connected to (0=serial port 1, 1=serial port 2) and 'baud' is the baud rate of the real khepera. The calibration may take a few minutes, don't use the computer and run as few applications as possible during the calibration.

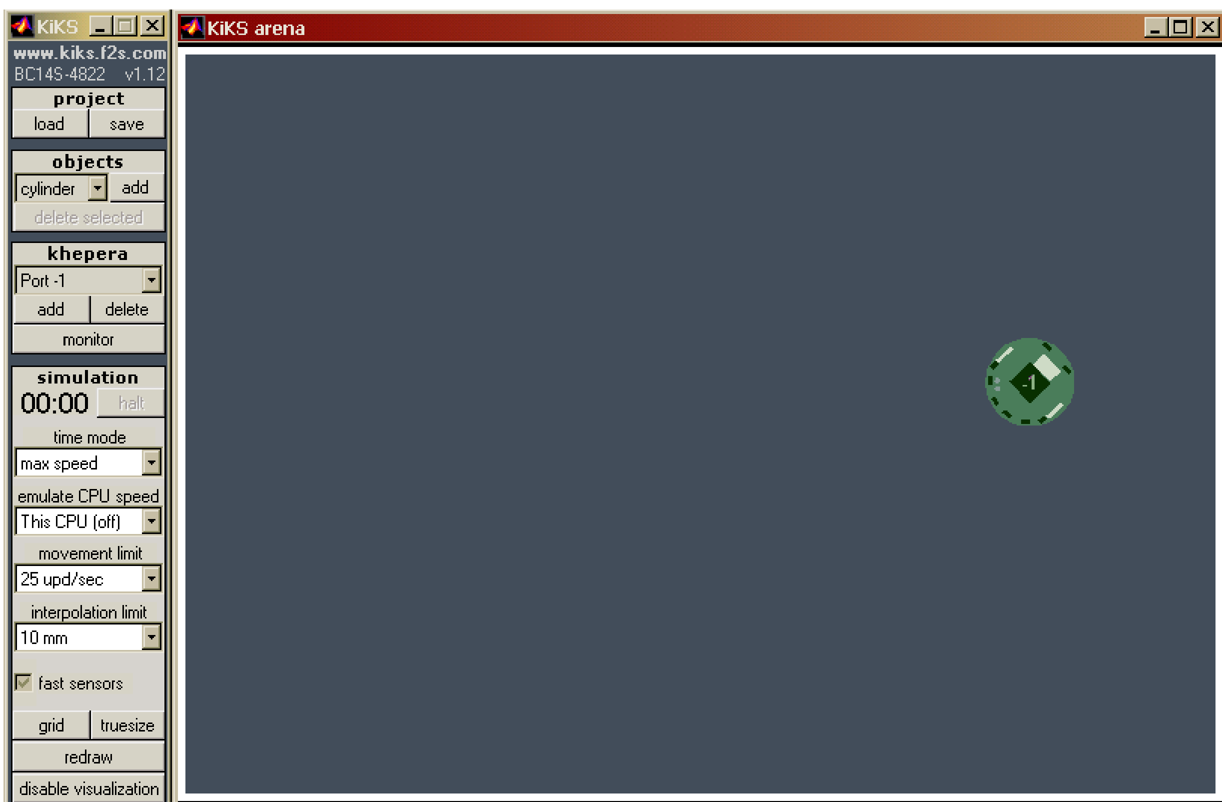
- If you do not have a Khepera connected to the computer, run `kiks_calibrate` without any arguments.

Starting up KiKS

To start up KiKS, type the following at the Matlab prompt:

```
kiks;
```

KiKS should start up and two windows resembling the ones below should appear.



You can move simulated Kheperas by placing the mouse pointer over the black square on the Khepera, pressing and holding the left mouse button, and moving the mouse. To rotate a simulated Khepera, simply repeat this procedure but over any of the dark green areas on the Khepera.

KiKS is now ready to be used. Try running the simple avoid-obstacle behavior in kiks\demos\avoid\avoid.m.

```
cd demos\avoid\
avoid;
```

Creating the simulated environment

An ARENA matrix is used to specify the simulated environment.

ARENA should be a matrix describing the arena or a string containing the filename of a project.
If ARENA is empty, a default arena is created and a Khepera is spawned.
If ARENA is an m*n matrix, an arena corresponding to the matrix is created and one (or more, if start positions are defined) Khepera(s) will be spawned.
Each matrix element must be one of the following:
<0=Khepera start position (absolute value defines the start angle)
0=no obstacle
1=wall
2=pushable object
3=light source
One matrix element corresponds to 1 square millimeter. The environment may be of any size.

You may also provide a COLORMASK matrix. This matrix tells KiKS what color the walls defined by the ARENA matrix are. The COLORMASK matrix must be the same size as the arena matrix. For each '1' element in the ARENA matrix at position [x,y], set [x,y] in the COLORMASK matrix to the color you want the wall to have.

The easiest way to create an arena with a colormask is to draw a grayscale (256 colors) picture using e.g. Photoshop, saving the picture as .tif, and use the command

```
[arena,colormask]=kiks_tif2arena('filename')
```

where filename is the name of the .tif file.
All completely black areas (RGB=#000000) will be regarded as floor by this function, and the rest of the colors should be used to draw the walls. So if you want a black wall, draw it using RGB #010101.

There are a few sample pictures in the demos\football\ directory, and the m-file demos\football\field.m demonstrates how to use kiks_tif2arena.

Here are a few examples at how to define arenas.

```
kiks(zeros(500,500));
```

will start up KiKS and create an empty arena of 500x500mm size. The sequence

```
ARENA=zeros(400,600);
ARENA(1:100,1:100)=1;
kiks(ARENA);
```

will start up KiKS and create a 400mm high and 600mm wide arena with a 100x100mm obstacle in the top left corner and the code sequence

```
[ARENA,COLORMASK]=kiks_tif2arena('myfile.tif');
kiks(ARENA,COLORMASK);
```

Will convert the picture myfile.tif into an arena and a colormask and start up KiKS.
If ARENA is a string, KiKS attempts to load a project using kiks_load(ARENA).

```
kiks('project');
```

The 'room cleaning' demo in kiks\demos\clean\clean.m is an example of this.

Project files are created by starting up KiKS, optionally with an arena and a colormask specified, placing Kheperas and objects in the arena using the GUI, and clicking 'save' under 'project' in the KiKS GUI.

You can switch arenas while KiKS is running by using the `kiks_arena(ARENA,COLORMASK)` command.

The KiKS graphical user interface

load/save project loads/saves arena and all objects and Kheperas in it.

add, **delete** should be self-explanatory. Pushable objects are 1.5cm in diameter, the light sources are 1 watt and slightly elevated (i.e. the Khepera cannot collide with lights). Select and move the objects using the mouse.

add Khepera adds a simulated Khepera. **Delete Khepera** removes the Khepera selected in the list. Press **monitor** to view selected Khepera status.

halt simulation - IMPORTANT! ***Never*** press ctrl-C to stop KIKS. Use this button instead for the same result, but in a more simulator-friendly way.

Simulation time mode - 'max speed' is recommended. 'wall clock' may not work well on slow computers.

emulate CPU speed - lets you examine how different CPU speeds affect Khepera behaviour. Emulating a slower CPU than the one present in the computer will result in faster simulation.

maximum position updates each simulated second - fewer is faster and less accurate.

movement interpolation - less is slower and more accurate

The **"fast sensors"** checkbox is checked by default and toggles real-time neural network calculations on/off. Checked is a lot faster, unchecked is more accurate.

grid turns on/off 1cm grid. **Truesize** resizes the arena so 1 mm=1 screen pixel

redraw removes Khepera and object 'trace lines'

enable/disable visualization toggles visualization on/off.

Controlling simulated Kheperas

KiKS simulates one or more Kheperas connected to serial ports, and you use three commands to communicate with Kheperas:
kiks_kopen, **kiks_ksend**, and **kiks_kclose**.

These commands work just like the kopen.dll, ksend.dll, and kclose.dll modules that K-team provide - in fact, you can use the kiks_k* commands to control real Kheperas aswell as simulated Kheperas.
When calling kiks_kopen with a negative serial port#, KiKS is activated and a working environment for a simulated Khepera is created. If, however, the serial port# >= 0 kiks_kopen will simply forward the call to kopen.dll.
kiks_ksend and kiks_kclose use the 'ref' array returned by kiks_kopen/kopen to detect if the call should be forwarded to ksend.dll/kclose.dll or if the call should be sent to a simulated Khepera.

A small code sample:

```
port=-1;                              % simulated Khepera
baud=9600;                            % use 9600 baud
ref=kiks_kopen([port,baud,1]);        % open the port for communication
kiks_ksend(['B' 13],ref)              % read version
kiks_kclose(ref);                     % close the port
```

Simply setting `port` to 0 or 1 (depending on which serial port your Khepera is connected to) will read the version of the real Khepera.

Kiks_ktime and kiks_pause are two new functions that only work in conjunction with kiks_kopen/kiks_ksend/kiks_kclose. Kiks_time(num) returns elapsed time since the serial port #num (simulated or real) was opened. If the port has not been opened, kiks_ktime will return [].
Kiks_pause(time) is intended to be used instead of pause(time). It allows for the simulator to make use of CPU time that would otherwise be spent doing nothing at all.

NOTE: Since Matlab doesn't support threads or any other kind of background processes, the simulated world is updated **only** when kiks_kopen, kiks_ksend, kiks_kclose, or kiks_ktime is called.

So, if you are familiar with using kMatlab to control Kheperas, getting used to KiKS should be very easy. It's only a matter of using kiks_kopen, kiks_ksend, and kiks_close instead of using kopen,

ksend, and kclose.

Look in the **KiKS\demos** folder for a few code examples.

Supported Khepera commands

KiKS supports all commands found in the Khepera user manual except for the following:

- Configure PID controller ('F')
- Read A/D input ('I')
- Set PWM ('P')
- Read a byte on the extension bus ('R')
- Write a byte on the extension bus ('W')

Appendix B: Modelling with neural networks

Introduction

This appendix is not meant to be an in-depth description of all techniques and theories involved in the area of neural networks. Instead, it is intended to provide some general information about neural networks and is aimed at those who have no or little previous knowledge of neural networks.

As the name implies, a neural networks consists of a number of interconnected neurons, where each neuron is a computing element.

A neuron has a number of input links and output links. Also, each input link has a weight.

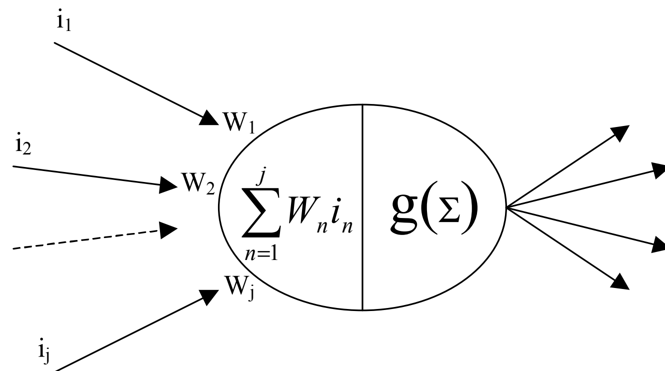


Figure 25

Figure 25 illustrates how a neuron computes its output by applying an **activation function** g(x) to the weighted sum of the input values. Only one output value is calculated even if there is more than one output link from the neuron, and the same value is thus sent on all output links. Additionally, the neuron can contain a **bias**. A bias can be viewed upon as a value that is added to the product of inputs and weights, so the output value is g(∑+bias).

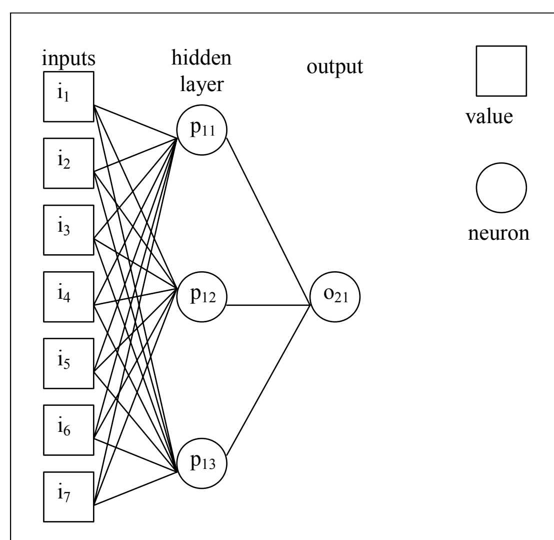


Figure 26

There are different ways of connecting the neurons, but the main distinction is made between feed-forward networks and recurrent networks.

In feed-forward networks, the neurons are usually grouped in layers as figure 26 shows. Each neuron receives its input from all neurons in the preceding layer and consequentially sends its output to all neurons in the following layer (except for output neurons, whose output lines used to output the resulting data from the network).

In particular, the links in feed-forward networks are unidirectional and output is never sent backwards in the network. In recurrent networks, neuron output can be sent backwards in the neural network – activation is sent back to the neurons that caused it. This allows neurons in recurrent networks to have internal state (aside from the input weights), but also means that computation in recurrent networks can be much less orderly than in feed-forward networks and therefore much harder to understand and train.

In this paper, only feed-forward neural networks have been used and hence, recurrent networks will not be further discussed.

Using neural networks to approximate a function

The general idea behind neural networks is that if we know a set of input and output values of a function (called the training set), the neural network can be adapted to associate each input with its output by adjusting the synaptic weights and neuron biases. This process is called *training the network*. The most common training method is back-propagation [5:p.578].

In theory, a feed-forward neural network with one hidden layer can approximate any continuous function of the inputs and a neural network with two hidden layers can approximate any function at all. However, there is no good theory to characterize functions that can be approximated with a small number of neurons [5:p.572], which means that there is no way to know how large the neural network has to be to approximate a given function. One might think that an easy way to get by this problem is to always use a very large neural network, but apart from the computing speed aspect the problem with this approach is that neural networks are subject to **overfitting** when there are too many weights in the network. Overfitting means that the neural network "memorizes" and becomes specialized at the data used to train the network, but performs badly when trying to approximate the function with input data that was not in the training set.

In practice, this means that in order to find a suitable neural network you have to start out with either a small or a large network, and adjust it until you have found a network that appears to be optimal, or at least performs well enough. This task becomes even more complex considering there are also a number of different transfer functions that can be used in the neurons.

Neural networks in Matlab

Provided the "neural network toolbox" is installed, Matlab provides very powerful functions for creation and training of neural networks. Since the neural networks used when simulating the proximity sensors have 7 unknowns, it is difficult if not impossible to visualize the function in an

understandable way. Instead, an example of a neural network that approximates a function with one unknown, namely the sinus function, will be used to show how to use neural network in practice in Matlab.

First, we need a training set. The training set should consist of inputs and outputs that describe the function to be approximated as well as possible.

```
input=[0:pi/8:pi*2];
output=sin(input);
```

Then, we need to specify the neural network to be used. As previously mentioned, there is no specific rule or theory that tells us how the network should be constructed, but since the function to be approximated is a trigonometric function the hyperbolic sigmoid transfer function **'tansig'** should be suitable. A complete listing of the different transfer functions in Matlab is available at http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/tabls12a.shtml#8665.

We know that there should be one input and one output in the neural network, which means that there should be only one neuron in the final (output) layer. Let's try creating and training a one-layered network with one 'tansig' neuron. Note that when the network is created, the weights are given random values by Matlab. The training function `traingdx`, which uses a back-propagation training method, will be used in the examples

```
net = newff(minmax(input),[1],{'tansig'},'traingdx'); % create network
net.trainParam.epochs = 300;
net=train(net,input,output);     % train network for no more than 300 epochs
test_input=[0:pi/32:pi*2];       % create input data for testing
Y=sim(net,test_input);           % let the network calculate output data
plot(test_input,Y,'b-');         % plot the network test input/output
hold on;
plot([0:pi/32:pi*2],sin([0:pi/32:pi*2]),'k:'); % plot the sinus function
plot(input,output,'ko');         % place rings at training data
xlabel('x');
ylabel('y');
```

Copying and pasting the two code segments into Matlab should yield a graph that resembles figure 27.

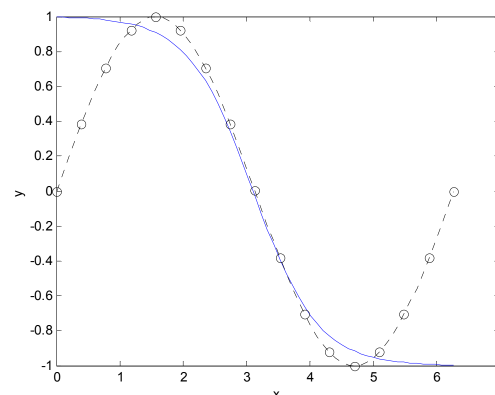


Figure 27

The dashed line is a plot of the sinus function, the solid line is a plot of the neural network output, and the rings are placed to indicate the input and

output data used when training the network.

Obviously, the network does not perform very well. This should not come as a surprise, since the network in fact only consists of the 'tansig' function applied to the product of one weight and one input with an added bias. Taking a closer look at the network structure allows us to find out the values of the weight and the bias.

```
weight=net.IW
bias=net.b
```

To confirm that this is true, try plotting the tansig function with these values. It should overlap the neural network plot.

```
plot([test_input],tansig(weight{1}*test_input+bias{1}),'r--');
```

To gain a better result, we need more neurons. Since there should be only one output neuron, we need to add a hidden layer. A suitable network is

```
net = newff(minmax(input),[4 1],{'tansig' 'purelin'},'traingdx');
```

which makes the final code for the entire example

```
input=[0:pi/8:pi*2];
output=sin(input);
net = newff(minmax(input),[4 1],{'tansig' 'purelin'},'traingdx');
net.trainParam.epochs = 300;
net=train(net,input,output);
test_input=[0:pi/32:pi*2];
Y=sim(net,test_input);
plot(test_input,Y,'b-');
hold on;
plot([0:pi/32:pi*2],sin([0:pi/32:pi*2]),'k:');
plot(input,output,'ko');
xlabel('x');
ylabel('y');
```

Figure 28 shows a graph of the neural network plotted against the sinus function and training values. Since the weights are given random values upon network creation, the training may not always result in good performance. Because of this, it is usually a

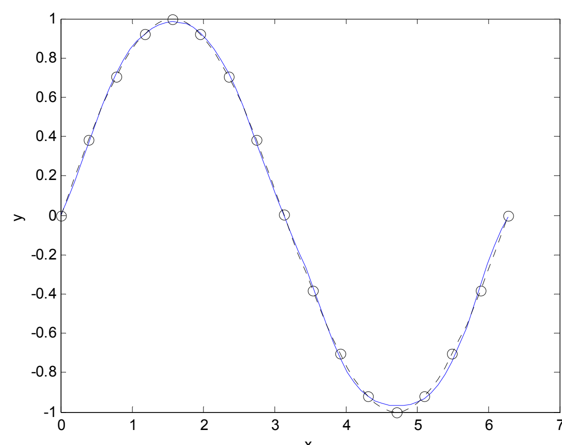
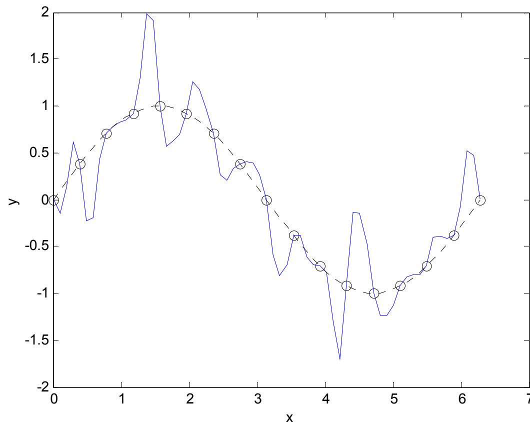


Figure 28

good idea to create and train several networks with identical topology and choose the one with best performance.

For an extreme example of overfitting, try having 30 neurons in the first layer.

```
net = newff(minmax(input),[30 1],{'tansig' 'purelin'});
```



As you can see, the error is close to 0 at the points used as training data but the neural network performs horribly at input values that are not very close to the test inputs.

Figure 29

Early stopping

One method for preventing overfitting is called **early stopping**. With this technique, the available data should be divided into three subsets: the training set, the test set, and the validation set.

As in the previous example, the training set is used for updating the weights and biases in the neural network, and the test set is used to evaluate the neural network after training is complete. During the training of the network, the performance on the validation set is monitored. Normally, the error on the validation set will decrease in the beginning of the training, but start to increase as the network begins to overfit the data. Early stopping means that training is stopped as soon as the error on the validation set starts increasing.

Early stopping does not generally result in dramatically better performance on the test data, but is a good way to prevent training

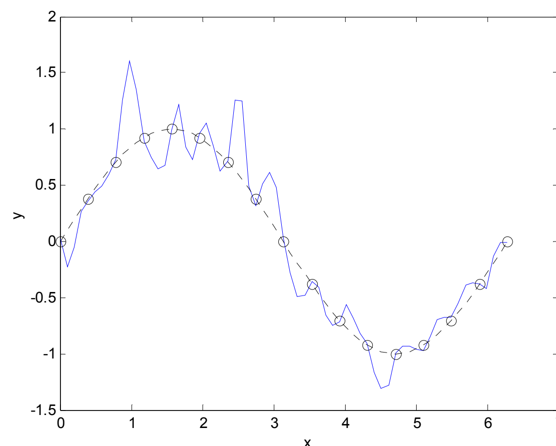


Figure 30

networks for unnecessary many epochs. In figure 30, a neural network with 30 neurons has been trained with early stopping using the Matlab code below.

```
input=[0:pi/8:pi*2];
output=sin(input);
v_input=[pi/16:pi/8:pi*2];  % create
v_input=v_input+pi/32*randn(size(v_input));
v_output=sin(v_input);      % the
vv.P=v_input;               % validation
vv.T=v_output;              % set
net = newff(minmax(input),[30 1],{'tansig' 'purelin'},'traingdx');
net.trainParam.epochs = 300;
net=train(net,input,output,[],[],vv);   % train network with early stopping
test_input=[0:pi/32:pi*2];              % create input data for testing
Y=sim(net,test_input);              % let the network calculate output data
figure; plot(test_input,Y,'b-');    % plot the network test input/output
hold on;
plot([0:pi/32:pi*2],sin([0:pi/32:pi*2]),'k:'); % plot the sinus function
plot(input,output,'ko');  % place rings at the values used as training data
xlabel('x');
ylabel('y');
```