khep_vehicle user manual

Version 0.7[1]
March 2000

Whit Schonbein
Philosophy – Neuroscience – Psychology Program
PNP Robot Lab
Department of Philosophy
Washington University
St. Louis, Missouri, 63130 USA

email: whit@twinearth.wustl.edu
web page: http://artsci.wustl.edu/~wwschonb
PNP Robot Lab web page:
http://artsci.wustl.edu/~philos/pnp/robotlab/robotlab.html

0. Introduction

The khep_vehicle program (kv) allows users to hand-code artificial neural network controllers for Khepera robots. This document provides an introduction to kv. A brief overview of connectionist networks and terminology is presented in section 1. Section 2 provides an introduction to the node types and node modifiers available in kv, and section 3 does the same for connection types. The basic syntax for specifying networks in kv is described in section 4. Section 5 describes the propagation of activation through networks in kv, and, finally, section 6 contains a brief discussion of running kv. Any updates to kv since version 0.7 are described in section 7, 'Updates'.

The author thanks the Department of Philosophy at Washington University for funding (summer 1999) contributing to the completion of this project.

License Agreement

[1] Note: This documentation may be out of date, and therefore not completely accurate. – Whit Schonbein, August 2003.

commercial) use, provided the modified code is not distributed without documentation of its modification. *Please* let Whit Schonbein know (whit@twinearth.wustl.edu) of any changes/improvements made.

1. Control Networks

It is assumed that the reader is familiar with the basics of connectionist architectures (see Rumelhart and McClelland, 1986, for an introduction). As in most connectionist models, networks in `kv` consist of *nodes*, arranged into *layers*, connected by links that have certain *weights* or values. The first layer of the network (layer 0) is the *input layer*, and the last layer of the network is the *output layer*. Layers occurring between the input and output layers are termed *hidden layers*, and the nodes that make up these layers are referred to as *hidden nodes*.

The input to any given hidden node is the sum of the weighted inputs to the node. So, for example, suppose nodes *j* and *k* feed into node *i* with weights of 5 and 2, respectively. If the output of node *j* is 1, and the output of node *k* is 2, the input to node *i* is $(1 \times 5) + (2 \times 2) = 9$. In general, the input to any given (non-input) node *i* is

$$input_i = \sum_j w_{ij} o_j$$

where $j = 0, 1, \ldots, n$ are nodes feeding into node *i*, o_j is the output of node *j*, and w_{ij} is the weight of the connection from node *j* to node *i*.

The output of a node depends on what sort of node it is; there are a number of different types of nodes in `kv`, and each has its own characteristic input-output function (activation function), as described below. Given some set of input values to a network, activation values (inputs and outputs of the nodes) are propagated forward through the network, generating outputs at the output layer. Typically, the Khepera's sensors supply input to the network, and outputs of the network are sent to the Khepera's motors.

Layers are numbered 0 through *m*-1, where *m* is the number of layers. Nodes are numbered 0 through *n*-1, where *n* is the number of nodes in the layer. Thus a pair consisting of a layer number and node number specifies a particular node. For example, (2,1) specifies the second node (node 1) of the third layer (layer 2).

2. Node Types and Node Modifiers

As previously mentioned, `kv` offers a number of distinct node types. They may be roughly categorized into *input* nodes, *output* nodes and *hidden* nodes.

2.1 Input Nodes

Input nodes are nodes that get at least part of their input from one of the robot's sensors. An input node cannot get input from more than one sensor. The node is said to be *bound* to the sensor from which it gets input. There are seven different types of input nodes. They are:

Infrared sensor node. An infrared sensor node is bound to one of the Khepera's eight infrared sensors. Sensors are numbered in accordance with figure 6 in the Khepera User Manual (K-Team SA, 1995, p. 6). So, for example, if node (0,1) is bound to sensor 2, this node (the second node of the first layer (layer 0, the input layer)) gets input from the front left sensor of the robot (sensor 2).

The value returned by the robot in response to a query regarding the infrared sensors ranges between 0 and 1000 (roughly), where 0 indicates nothing is detected, and 1000 indicates that something is strongly detected. The activation function of an infrared sensor node scales this input value to an output value ranging between –1 and 1, where 1 indicates that nothing is sensed, and –1 indicates that something is strongly detected. The default activation function is $f(x) = x(-1/500) + 1$, which means that the output is zero $x = 500$. At the time of writing, this parameter can be changed by modifying the `ir_zero_crossing` variable in `kv7.cc` and re-compiling. Future versions of `kv` may (i) support similar modifications in other input node types, and (ii) allow this value to be changed at runtime (as opposed to compile-time).

Ambient sensor node. As with an infrared sensor node, an ambient sensor node is bound to one of the Khepera's eight ambient light sensors. The ambient light sensors are simply the infrared sensors functioning in a passive capacity, i.e., in ambient mode the sensors do not emit infrared light. In this mode of operation, the sensors return values between 0 (light) and 500 (dark). The output of an ambient sensor node is scaled to a value between –1 (dark) and 1 (light), with the output equaling zero when the input is 250. Note that at the time or writing, there is no variable analogous to `ir_zero_crossing` for ambient sensor nodes.

Movement sensor node. The Khepera are equipped with two movement sensors, one for each wheel. These sensors can detect how fast and in what direction a wheel is turning. A movement sensor node is bound to one or the other movement sensors.

Vision turret pixel node. Khepera robots are equipped with an expansion bus. One of the pre-fabricated modules designed for the Khepera is a vision turret (K-Team SA, 1995b). The vision turret consists of a single array of 64 light sensitive pixels arranged in a line horizontal to the horizon. They are numbered zero to 63, from left to right across the visual field, and taken together have a field of vision of approximately 36 degrees. The turret has an optimal viewing range from 5 to 50 centimeters in front of the robot – smaller and larger distances are out of focus. Each pixel can report 255 gray-levels.

Vision turret nodes are bound to individual pixels of the vision turret array. The input to a vision turret node is a value between 0 and 254, inclusive, where 0 indicates low light intensity (e.g., darkness), and 254 indicates a high light intensity (e.g., bright light). This input value is scaled it to a value between 0 and 1, where 1 corresponds to a high light intensity, and 0 to a low light intensity. Note that if one desires an output in the range of –1 to 1, simply give

the node a bias of –0.5, and give connections from the node a weight of 2. Similar manipulations allow for other output ranges to be obtained.

Vision turret light intensity node. The vision turret also has the ability to report the average light intensity across the entire visual array. A vision turret light intensity node receives input from this form of sensor information. The input to the node is a value between 0 and 254, inclusive, where 0 indicates low light intensity, and 254 indicates high light intensity. This input value is scaled to a value between 0 and 1 as per vision turret pixel nodes.

Vision turret maximum light intensity node. The vision turret can return the index of the pixel in the vision array that has the highest light intensity value of the entire array. A vision turret maximum light intensity node takes as input the index (0 – 63) of the node with the maximum intensity, and outputs that same value.

Vision turret minimum light intensity node. This node type is identical to the vision turret maximum light intensity node, except that it outputs the index of the node with the least light intensity value in the array.

Input nodes typically appear in the input layer of a network, but this is not strictly required; `kv` supports input nodes in any layer of the network.

2.2 Output Nodes

There is currently only one type of output node supported by `kv`, namely motor output nodes. A motor output node is bound to one or the other of the two motors of the robot. Motors are designated ‘left’ and ‘right’ when viewing the robot from above, with the forward sensors at twelve o’clock. The input to the node is the sum of the weighted outputs of nodes feeding into the node. The activation function of the node is the identity function. So a motor output node simply passes its input to the motor to which it is bound. A positive value causes the associated motor to move forwards; larger values correspond to faster forward speeds. Likewise, a negative value causes the associated motor to turn in reverse, and more negative values correspond to faster reverse speeds. See K-Team SA (1995a) for more details.

2.3 Hidden Nodes

Hidden nodes are distinguished according to their distinctive *activation functions*. Currently, there are five different activation functions implemented in `kv`. They are:

Sigmoid. A hidden node utilizing the sigmoid activation function generates an output according to the function

$$f(x) = 2 * \left(\left(\frac{1}{1 + e^{-x}} \right) - 0.5 \right)$$

where x is the input to the node. In this way the input is 'squashed' to a value between -1 and 1.

Threshold (above). `kv` supports four types of threshold functions. In the present case, the activation function is

$$f(\mathrm{x}) = \begin{cases} 1 \text{ if } \mathrm{x} > t \\ 0 \text{ if } \mathrm{x} \leq t \end{cases}$$

where t is a threshold value specified by the user. In other words, the node fires if the input is greater (non-inclusive) than a specified value. This is sometimes called the 'above-threshold' function.

Threshold (below). The below-threshold activation function outputs 1 if the input is less (non-inclusive) than a specified threshold t, and 0 otherwise:

$$f(\mathrm{x}) = \begin{cases} 0 \text{ if } \mathrm{x} > t \\ 1 \text{ if } \mathrm{x} \leq t \end{cases}$$

Threshold (in range). Also known as the 'between-threshold' or 'within-range threshold' function, this activation function outputs a value of 1 if the input falls within a (non-inclusive) range specified by the user:

$$f(\mathrm{x}) = \begin{cases} 1 \text{ if } \mathrm{x} > t_1 \text{ and } x < t_2 \\ 0 \text{ otherwise} \end{cases}$$

where t_1 is the lower bound of the desired range, and t_2 is the upper bound.

Threshold (out of range). Also known as the 'outside-range threshold' function, this activation function outputs a value of 1 if the input falls outside a range (non-inclusive) specified by the user:

$$f(\mathrm{x}) = \begin{cases} 0 \text{ if } \mathrm{x} > t_1 \text{ or } x < t_2 \\ 1 \text{ otherwise} \end{cases}$$

where t_1 and t_2 are the lower and upper bounds, respectively.

Identity. The identity activation function simply outputs the input to the node.

2.4 Node Modifiers

In addition to multiple node types, `kv` also has a small set of *node modifiers*. A node modifier is a way of 'tweaking' a node so as to give it additional features. Some of these adjustments affect the behavior of the network while others do not. They are:

Noise. A node modified with the noise modifier has a random value between –1 and 1 added to its *output* at each time step.

Bias. A biased node has a user-specified bias (constant) value added to the *input* of the node at each time step.

Probe. When a node is probed, information about the functioning of the node is displayed during runtime, e.g., input value, output value, etc.

Clamp. Clamping a node sets the *output* value of the clamped node to a user-specified value before the network is run. This is extremely useful (for example) in cases where two sub-networks are meant to function exclusively of one another, and the user needs to guarantee that one of the two networks is active at run time.

3. Connection Types

3.1 Excitatory and Inhibitory Connections

`kv` supports two types of inter-node connections, *excitatory* and *inhibitory*. Let the node from which a connection arises be called the *source* node, and the node into which a connection feeds the *target* node. Let *c* be a connection with weight *w* between a source node and a target node.

If *c* is an excitatory link, then *c* contributes to the total input of the target a value equal to the output value of the source node multiplied by *w* (recall that the target node may have other connections feeding into it in addition to *c*). Note that *w* may be positive or negative, as can the output of the source node. Therefore *c*'s contribution to the total input of the target node may be positive or negative.

In contrast, suppose *c* is an inhibitory link. In this case, the following situation holds. If the output of the source node is *positive*, then the output of the target node is 0, *regardless of the value of* w*, or of the inputs of any other connections to that target node*. That is, an inhibitory connection effectively squelches the output of its target node, provided the input to that connection is positive.

3.2 Recurrent and Feedforward Connections

kv supports *recurrent* connections as well as *feedforward* connections. A feedforward connection is a connection from a node in an earlier layer to a node in a later layer. A recurrent connection is a connection from a node in a later layer to a node in an earlier layer. Both feedforward and recurrent connections may be excitatory or inhibitory.

Furthermore, kv supports *intra-layer* connections as well as inter-layer connections. An intra-layer connection is a connection from one node to another within the same layer, or between one node and itself. The latter is sometimes referred to as a *self-recurrent* connection, since the connection feeds back to the same node.

Section 5 provides information on how kv handles the propagation of activation in recurrent and intra-layer connections.

4. Specifying Network Architectures

Perhaps the easiest way to introduce the format for the specification of network architectures in kv is through an example. Section 4.1 presents a simple obstacle avoidance architecture. Section 4.2 provides a generalized summary.

4.1. An Example: Obstacle Avoidance

The following example serves as a simple introduction to network specification in kv. Suppose our goal is to have a robot move forward while avoiding obstacles. One solution is depicted in figure 1. In this example, layer 0 consists of two input nodes.

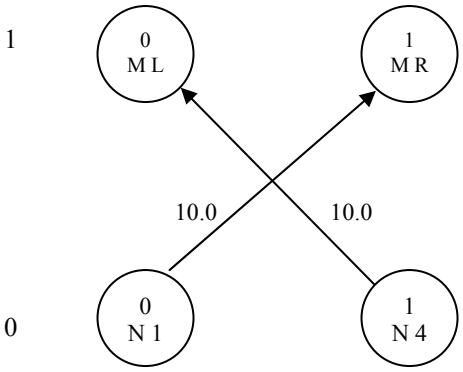

Figure 1. A simple obstacle avoidance network.

Node (0,0) is an infrared sensor node bound to sensor 1, and node (0,1) is an infrared sensor node bound to sensor 4. Node types are specified according to a single-letter code, in this case 'N', which indicates an infrared sensor node (a table summarizing all available node codes appears in section 4.2). Nodes (1,0) and (1,1) are motor output nodes (code 'M'), the former bound to the left motor ('L'), the latter to the right motor ('R').

The network has two connections, the first between nodes (0,0) and (1,1), the second between (0,1) and (1,0). Each connection has a weight of 10.0. Figure 2 depicts a configuration file specifying this architecture.

```
s
2
L 0 2
L 1 2

W 0,0 1,1 10.0
W 0,1 1,0 10.0

N 0,0 N 1
N 0,1 N 4
N 1,0 M L
N 1,1 M R

*

avoid.net : a simple obstacle avoidance architecture.
```

Figure 2: `kv` network configuration file for the obstacle avoidance network depicted in figure 1.

Network configuration files can have any filename, although it is recommended that some distinguishing extension be used (in this case, ".net"). A line-by-line explanation of this file is as follows:

The first line of the file ('s') is not used at the time of writing. However, it is required by `kv` for proper execution. Hence every configuration file must have an 's' (lowercase) as the first line[2].

The second line of the file specifies the number of layers in the network (2).

The next section of the file specifies the number of nodes per layer. So, for example, the third line of the file begins with an 'L', which indicates that the next two values will be a layer number (in this case 0) followed by the number of nodes in that layer (in this case 2). Recall that, supposing a network has *n* layers, layers are numbered from 0 to *n*-1. If layer numbers are specified that fall outside this range, behavior is unpredictable at best (most likely `kv` will crash with a segmentation fault). Furthermore, the number of nodes per layer must be specified for all layers of the network. Failure to do so results in unpredictable behavior.

The next section of the file, those lines beginning with 'W', specifies the connectivity of the network. So, for example, the first line indicates that there is a

[2] For the curious, 's' stands for 'saved', and contrasts with 'n' for 'new'. If a file has an 'n' as the first line, and provided that no weight values are specified in the connections section of the file, `kv` will assign random weights (between –10 and 10) to each connection in the network. Behavior is untested if 'n' is used and weight values are provided by the user.

connection (a 'weight', 'W') between node (0,0) and node (1,1), and that this connection has a weight of 10.0.

Those lines beginning with 'N' provide node type specifications. The first of these lines specifies that node (0,0) is an infrared sensor node (type 'N') bound to sensor 1. The second line specifies that node (0,1) is an infrared sensor node bound to sensor 4. The third that node (1,0) is a motor node bound to the left motor ('L'), and the fourth that node (1,1) is a motor node bound to the right motor.

Finally, the file concludes with an asterisk ('*') and a comment. `kv` ignores everything after the concluding asterisk, providing the user with space to comment on the network configuration contained in the file.

The network can be run as follows. Assuming `kv` is compiled and the robot is attached to the computer according to the user manual (K-Team SA, 1995a), and that the file is called 'avoid.net', it can be executed by entering, at the command line,

```
kv avoid.net 1000
```

where the '1000' is the number of time-steps to run the network. The robot should move forward and turn away from (light colored) objects. More information on running `kv` is given below, in section 6.

4.2. General Configuration File Specifications

Network configuration files are divided into roughly five sections. They are (1) the layers, (2) the weights, (3) the nodes, (4) the node modifiers, and (5) comments (Figure 3). The first five subsections describe these five sections of the configuration file. However, there is one piece of errata to be mentioned: *The first line of a* `kv` *network configuration file is always a lowercase 's'*. This instructs `kv` to *not* generate random

```
s
---------------------------------------------------------------
Section 1: Layer Specifications
---------------------------------------------------------------
Section 2: Connection Specifications
---------------------------------------------------------------
Section 3: Node Specifications
---------------------------------------------------------------
Section 4: Node Modifiers
---------------------------------------------------------------
Section 5: Comments
```

Figure 3: Basic network configuration file structure.

weights for the connections in the network. Failure to follow this convention results in untested behavior.

4.2.1 Specification of Layers

The first main section of a network configuration file specifies the number of layers and the nodes per layer in the network. The general syntax is

```
<number of layers>
L 0 <number of nodes>
L 1 <number of nodes>
      .
      .
      .
L <number of layers – 1> <number of nodes>
```

`<number of layers>` and `<number of nodes>` are positive integers. The capital 'L' indicates that the line contains information concerning the layers of the network. Recall that layers are numbered from 0 through the number of layers in the network less one. The number of nodes must be specified for all layers in the network. So, for example, the following segment declares a network of three layers, with 2 nodes in layer 0, 1 node in layer 1, and two nodes in layer 2:

```
3
L 0 2
L 1 1
L 2 2
```

There are two important restrictions on the *output* layer of networks in `kv`. First, `kv` is designed to support a maximum of two output nodes (one for each motor) at the output layer. If more than one output node is given for a particular motor, `kv` sends the output value of the node with the *higher* index. For example, suppose a network was declared with three motor output nodes in the output layer: The first (node 0) is bound to the left motor, the second (node 1) to the right, and the third (node 2) to the left. In this case, `kv` will ignore the output of node 0, and send only the output of node 2 to the left motor.

Second, error messages will be displayed if any non-output (i.e., non-motor output) nodes appear in the output layer. If one wishes to have *no* output to the motors at all, then simply include an output layer with 0 nodes in the network configuration.

4.2.2 Specification of Weights

The second section of the configuration file specifies the connections and their weights for the network. The basic format for declaring a connection and weight is

```
W <source layer>,<source node> <target layer>,<target node> <weight
value> [optional 'i' if connection is inhibitory]
```

Each declaration should have its own line in the configuration file. So, for example, the following segment declares two connections. The first is a connection from node (0,1) to

node (1,0) with a weight of 5.0, and the second is an *inhibitory recurrent* connection from node (2,1) to node (1,0) (the weight has been arbitrarily set to 1.0):

```
W 0,1 1,0 5.0
W 2,1 1,0 1.0 i
```

4.2.3 Specification of Nodes

The third section of the configuration file specifies the node types and type-dependent parameters. The basic format for declaring a node type is

```
N <layer>,<node> <node type> [any type-dependent options]
```

Tables 1 and 2 summarize node types and their various type-dependent options.

Table 1. Node types and their associated codes.

Code	Type	Notes
N	Infrared Sensor Node	*Syntax*: `N <sensor>` *Summary*: `<sensor>` is an integer between 0 and 7, inclusive. Sensors are numbered in accordance with the numbering in the Khepera User Manual (K-Team SA 1995a). The infrared sensors on a Khepera robot return a value between zero and 1000, with 1000 indicating an obstacle in very close proximity to the sensor. Barring any recurrent connections, the infrared sensor node takes as its input this raw sensor value, and scales it to a value between –1 and 1, where –1 corresponds to an object in close proximity, and 1 to nothing being detected. As a default, the node outputs zero when the raw sensor value is 500. *Example*: `N 0,1 N 4` (The first 'N' indicates the line contains node information. The node is 0,1, it is an infrared sensor node (type N), and is bound to sensor 4.)
A	Ambient Light Sensor Node	*Syntax*: `A <sensor>` *Summary*: `<sensor>` is a value between 0 and 7, inclusive. Sensors are numbered in accordance with the numbering in the Khepera User Manual (K-Team SA 1995a). The infrared sensors on a Khepera robot can also function as ambient light sensors (when used in a passive manner). In this mode the sensors return a value between 0 (light) and 500 (dark). The ambient light sensor node scales this input to a value between –1 (dark) and 1 (light), with the zero crossover at an input of 250. *Example*: `N 0,1 A 2` (The first 'N' indicates that the line contains node information. The node is 0,1, and it is an ambient light sensor node (type A) bound to sensor 2).
V 1	Vision Turret Pixel Sensor Node	*Syntax*: `V 1 <pixel>` *Summary*: `<pixel>` is an integer between 0 and 63, inclusive, corresponding to the pixel numbering specified in the K213 Vision Turret Manual (K-Team SA, 1995b). The vision turret consists of an array of 64 pixels, each of which returns a value between zero and 254, inclusive. Zero corresponds to a low light intensity (typically black), and 254 to high light intensity (typically white). Barring any recurrent connections, the vision turret pixel sensor node takes as its input this raw sensor value, and scales it to a value between –1 and 1, where 1 corresponds to a high light intensity,

		and –1 to a low light intensity. *Example*: `N 0,7 V 1 13` (The 'N' indicates that the line contains node information. The node is 0,7, and it is vision turret node of type 1 (pixel node) bound to pixel 13).
V 2	Vision Turret Ambient Light Intensity Sensor Node	*Syntax*: `V 2` *Summary*: The vision turret is equipped with the ability to return a value indicating the average light intensity across the array. The range of this value is the same as for the vision turret pixel (V1). The vision turret ambient light sensor node takes as its input this raw sensor value, and scales it to a value between –1 and 1 in the same fashion as V1 type nodes. *Example*: `N 0,3 V 2` (The 'N' indicates that the line contains node information. The node is 0,3, and it is a vision turret node of type 2 (ambient light intensity sensor node). No further specification is required).
V 3	Vision Turret Maximum Light Intensity Node	*Syntax*: `V 3` *Summary*: The vision turret is equipped with the ability to return the index of the pixel with the highest light intensity value. The output of the node is simply the index of the pixel (i.e., the node has the identity function as an activation function). *Example*: `N 0,4 V 3` (See example for node type V 2).
V 4	Vision Turret Minimum Light Intensity Node	*Syntax*: `V 4` *Summary*: This node is identical to V 3, only provides the index of the pixel with the least light intensity. *Example*: See example for type V 3.
E	Motor Sensor Node	*Syntax*: `E <sensor>` *Summary*: `<sensor>` is either '`L`' (for the left motor) or '`R`' (for the right motor). The motor sensors on a Khepera return a value between –127 and 127, with a negative value indicating the wheel is turning backwards, and a positive value indicating the wheel is turning forwards. The larger the magnitude, the faster the wheel is turning. The motor sensor node takes as its input this raw value and scales it to a value between –1 and 1, with negative values indicating backwards movement, and positive values indicating forward movement. *Example*: `N 0,0 E L` ('N' indicates the line contains node information. The node is 0,0, it is a motor sensor node (type E), and is bound to the left motor sensor).
M	Motor Node	*Syntax*: `M <motor>` *Summary*: `<motor>` is either '`L`' (for the left motor) or '`R`' (for the right motor). Currently the only output node available in `kv`, each motor node in a network (there should be two at the most) is associated with (or 'bound to') either the left or the right motor. The motor node implements an identity function, and simply passes its input to its associated motor. *When used as the source node for a recurrent connection, the motor node's output is scaled to a value between –1 and 1, so that its functionality is similar to that of a motor sensor node (E).* The value passed to the associated motor is unaffected. *Example*: `N 3,0 M L` ('N indicates that the line contains node information. The node is 3,0, it is a motor output node (type M), and it is bound to the left (L) motor.
H	Hidden Node	A hidden node is a node that is neither an input node (N, A, V1, V2, V3, V4, E) nor an output (M) node. Hidden nodes may implement a variety of functions. These are summarized in a separate table.

Hidden nodes have different possible activation functions, and these must be specified in the network configuration file. Table 2 provides a summary of the codes and function types.

Table 2. Hidden Node Function Types

Code	Type	Notes
T 1	Above-Threshold Function	*Syntax*: `T 1 <threshold>` *Summary*: Node outputs 1 if its input is greater than `<threshold>`, where `<threshold>` is a real number. *Example*: `N 1,2 H T 1 0.15` (The node is 1,2, it is a hidden unit (type H) using a threshold function (type T) of the first sort (type 1, the above-threshold function), and the threshold is 0.15. Hence the node outputs a 1 if the input to the node is greater than 0.15)
T 2	In-Range (Between) Threshold	*Syntax*: `T 2 <lower bound> <upper bound>` Summary: <lower bound> and <upper bound> are real numbers. Node outputs 1 if its input is greater than <lower bound> and less than <upper bound> Example: N 1,2 H T 2 0.15 1.0 (Node 1,2 is a hidden node (tyfires if the input to the node is greater than 0.15 and less than 1.0)
T 3	Out-of-Range (Outside) Threshold	Syntax: `T 3 <lower bound> <upper bound>` Summary: Node outputs 1if its input is less than `<lower bound>` or greater than `<upper bound>` Example: `N 1,2 H T 3 0.15 0.75` (fires if input is less than 0.15 or greater than 0.75)
T 4	Below-Threshold Function	Syntax: `T 4 <threshold>` Summary: Node outputs 1 if its input is less than `<threshold>` *Example*: `N 1,2 H T 4 0.60 (Node outputs 1 if input is less than 0.60)`

4.2.4 Specification of Node Modifiers

Node modifiers are not standardized in `kv`. Their syntax is as follows (see section 2.4 for further discussion).

Noise. The following declaration is used to add noise to a node:

```
N <layer>,<node> n
```

So, for example, the line

```
N 1,1 n
```

adds noise to node (1,1), the second node of the second layer.

Bias. The following declaration is used to add a bias value to a node:

```
N <layer>,<node> b <bias value>
```

So, for example, the line

```
N 1,2 b 1.3
```

adds a bias of 1.3 to the input of node 1,2 (the third node of the second layer) at each time step.

Clamp. The following declaration is used to clamp a node to a particular value before running the network:

```
C <layer>,<node> <value>
```

where `<value>` is a real number. So, for example, the line

```
C 1,0 1.0
```

clamps the value of node 1,0 (the first node of the second layer) to 1.0. That is, the output value of node 1,0 will be 1.0 the first time step.

Probe. The following declaration is used to probe a node:

```
N <layer>,<node> p
```

For example, the line

```
N 1,5 p
```

probes the sixth node of the second layer.

4.2.5 Comments

The configuration file *must* contain an asterisk, on its own line, after the layer, weight, node, and node modifiers specifications. Everything after this asterisk is ignored by `kv`. This provides arbitrary space for comments on the network specified in the file. Keeping commented network configuration files greatly reduces the time taken to relocate particular networks.

5. Propagation of Activation in `kv`

As previously mentioned, nodes in a network take on certain input and output values depending on their inputs and their activation functions. These values are referred to as

the *activation* of the nodes[3]. The first nodes to become active are the nodes of the input layer. The activation propagates forward through the network until the output layer is reached, according to the following algorithm:

```
For each layer
      For each node
            Calculate input to node
            Generate output of node
      Advance to next node in layer
Advance to next layer
Send values of any output nodes to motors
```

This algorithm encapsulates one pass through the network, from input layer to output layer. Each pass through a network is one *time step*. Thus networks in `kv` are updated *synchronously*: All nodes are updated at each time step. Note that the actual amount of time it requires to complete a pass through the network is a function of how many nodes and connections there are in the network, as well as how often `kv` must query the vision turret (see below).

At the end of each pass through the network, the values of any output nodes are sent to their associated motors. Keep in mind the two constraints on the output layer (section 4.2.1): (1) kv is designed to support a maximum of *two* motor output nodes (one for each motor), and (2) non-output nodes cannot appear in the output layer.

There are a number of other important aspects of the activation propagation algorithm that must be mentioned. These include details on the handling of intra-layer and recurrent connections, and vision node updates.

Recurrent connections are connections from a node in a higher level to a node in a lower level. Suppose, for example, at time *t* node (1,0) receives input *exclusively* from node (2,2) due to a recurrent connection with weight *w*. Then node 1,0 receives as its input the output of node (2,2) *from time step t-1* multiplied by *w*. In general, any node that has a recurrent connection feeding into it receives as input from that node the source node's output from the previous time step (multiplied by the weight of the recurrent connection).

Self-recurrent and intra-layer connections work in the same way. In the former case, the input a node receives from a recurrent connection with itself is a function of the output of that node at the previous time step, multiplied by the weight of the connection. In the latter case, the input a node receives from another node in the same layer, *regardless of whether or not the node is of a higher or lower index than the target node*, is a function of the output of the source node from the previous time step, multiplied by the weight of the connection.

There is one exception to this general rule: Output values of motor output nodes sent through recurrent connections from motor output nodes are scaled to a value between –1 and 1, so that they behave like motor sensor nodes (type E). For example, suppose at time *t* motor output node *N* sends a value of *x* to the left motor. Furthermore

[3] Note that in most connectionist literature, the term 'activation' is reserved for the sum of the weighted inputs (i.e., the total input) to a node. This corresponds with the general idea that a node is 'stimulated' to the point of firing (i.e., generating an output): When the node is activated to a sufficient degree, it fires. Nothing hinges on this technicality in the current document, so it is ignored.

there is a recurrent connection from node *N* to some node in a lower level. Rather than use the output value *x* of *N* for computing the input to the lower level node, `kv` first scales *x* to a value between –1 and 1 before computing the input to that node. This is done because motor output nodes tend to output rather high values, compared to values contributed to the network by sensor nodes. In this way the recurrent input of motor nodes is brought more into line with the general input values of the network.

Finally, a note about sensor readings in `kv`. First, to speed up the operation of `kv`, infrared and ambient sensors are only queried once per time step, and then only if there is a node of the relevant type. So, for example, if there are multiple infrared sensor nodes in the network, `kv` will query the robot for sensor values only when the first infrared sensor node is encountered during the propagation of activity; that is, infrared encountered after the initial infrared sensory node get their values from the same set of values initially obtained – the robot is not queried again during the time step. Furthermore, if there are no ambient sensor nodes (for example) in the network, `kv` will never query the robot for the values of those sensors. See the `khep_core` documentation for more information.

Second, querying the vision turret for sensor values is time-consuming. If the vision turret is queried every time step, the robot often fails in its task, because time better spent navigating is spent gathering data from the vision turret. In order to minimize this drain on resources, the vision turret is queried only once every *x* time steps (provided the network contains at least one vision turret node), where *x* is some positive integer. The value of *x* corresponds to the `vision_sample` variable in `kv7.cc`, and can be set at compile time.

6. Running `kv`

Assuming `kv` has been compiled, the syntax for running the program is:

```
kv7 <filename> <time steps> [-b <baud rate>]
```

where `kv7` is the name of the executable, `<filename>` is the name of the file containing the specification of the network architecture to be run, and `<time steps>` is the number of time steps for which the network is to be run. There is currently one optional parameter, '-b', which allows the user to specify the baud rate of the serial port interface. The default baud rate is the fastest supported by the Khepera. So, for example,

```
kv7 avoid.net 1000 –b B9600
```

runs the network specified in the file '`avoid.net`' for 1000 time steps at 9600 baud (note that baud rate specifications require a 'B' to be prefixed to the integer value). Consult the code in `main.cc` for more information.

During execution, `kv` displays the current time step in increments of 100, as well as displaying any information from probed nodes.

Pressing `Ctrl+C` will stop the program. Note that in the absence of any further commands, the Khepera will continue to execute the last command received. If the program runs to completion, `kv` issues a stop command to the robot.

7. Updates

- Version 0.7a : Added command line serial port selection. Syntax is `-p <port>`, where `<port>` is one of the serial port devices. For example

  ```
  kv avoid.net 1000 -p /dev/ttyS2
  ```

  Runs a robot on serial port 3 (COM3 in DOS) for 1000 time steps using the architecture specified in the file avoid.net. Note that the full path for the device must be specified.

- Version 0.7a: Activation function of ambient sensor nodes can be manipulated via the `a_zero_crossover` variable at compile-time.

References

K-Team SA (1995a). Khepera User Manual. Lausanne, France.

K-Team SA (1995b). Khepera vision turret user manual. Lousanne, France.

Rumelhart, D. E. and J. McClelland (1986). Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Cambridge, MA, MIT Press.

Whit Schonbein
March 2000