The PDP++ Software Users Manual

Version 3.0 (2 May 2003)

Chadley K. Dawson, Randall C. O'Reilly, and James L. McClelland.

Welcome to the PDP++ Software Users Manual

Chadley K. Dawson, Randall C. O'Reilly, and James L. McClelland.

This describes version 3.0 of the PDP++ neural network simulation software.

Manual revision date: 2 May 2003.

1 Introduction to the PDP++ Software

As the field of connectionist modeling has grown, so has the need for a comprehensive simulation environment for the development and testing of connectionist models. Our goal in developing PDP++ has been to integrate several powerful software development and user interface tools into a general purpose simulation environment that is both user friendly and user extensible. The simulator is built in the C++ programming language, and incorporates a state of the art script interpreter with the full expressive power of *C++*. The graphical user interface is built with the Interviews toolkit, and allows full access to the data structures and processing modules out of which the simulator is built. We have constructed several useful graphical modules for easy interaction with the structure and the contents of neural networks, and we've made it possible to change and adapt many things. At the programming level, we have set things up in such a way as to make user extensions as painless as possible. The programmer creates new C++ objects, which might be new kinds of units or new kinds of processes; once compiled and linked into the simulator, these new objects can then be accessed and used like any other.

The system is very powerful, and as with all powerful things, one needs to learn how to handle it to get the most out of it. But the system is also engineered for ease of use by the beginner. Classic programs (such as backpropagation) and classic example simulations (like learning XOR) have been set up so the user can run them and explore their behavior with a few key strokes and mouse clicks. It's easy to set up new examples and modify various parameters and explore the effects. There are libraries of different unit and process types, so it is easy for you to put together novel simulations on your own.

This manual provides a complete overview of the PDP++ system, starting from the simplest kind of uses and progressing to a full description of all of the features so you can get the most out of the system. We strongly recommend reading the Chapter 3 [over], page 14 chapter in its entirety, and then stepping through the Chapter 4 [tut], page 20 chapter to get some hands-on familiarity with the system. If the software is not yet installed on your system, you must follow the instructions in the Chapter 2 [inst], page 4 chapter to install it. The remainder of the manual can be used as a reference as needed, or read through for a deeper understanding of how things really work.

Before you'll be able to use this system, or even understand this manual, you will need some background. First, you'll need to have a working understanding of the Unix operating system, the X11 window system, and your window manager (Motif, OpenWindows, or whatever you use). You'll probably also need to know something about connectionist/neural network simulation models. For general background, consider Chapters 1-4 of Parallel Distributed Processing, Volume 1 (Rumelhart, McClelland, and the PDP Research Group, 1986). While section BP of this tutorial presents back propagation as a procedure, it does not fully motivate it or discuss the reasons for its importance; for this we refer you to Chapter 8 of PDP Volume 1, to Chapter 4 of Explorations in Parallel Distributed Processing (McClelland and Rumelhart, 1988) or to Rumelhart, Hinton and Williams (1986).

New for 2.0: The PDP++ software provides the basis for a large number of cognitive-neuroscience oriented simulations in the new book "Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain", by O'Reilly and Munakata, MIT Press, 2000 (September). These simulations use the

`leabra++` algorithm, which is now provided with the PDP++ software in the `src/leabra` directory. The simulations can be downloaded for free from either the MIT Press website ('`http://www.mitpress.com`' – search for the book by title or authors) or from a link off of O'Reilly's we page at '`http://psych.colorado.edu/~oreilly`'. Of course, to get the most out of them, it is recommended that you purchase the book ;).

A mailing list is available for users of PDP++ to share ideas and help each other solve problems. This list will be monitored by the authors of the PDP++ software, but ABSOLUTELY NO GUARANTEE or even SUGGESTION of technical assistance is implied by the existence of this list. We have limited resources and are making the software available as a public service. Please respect this by not asking for help from the authors directly. To be added or removed from the mailing list, and/or to view archives of previous messages, see the instructions at the website, '`http://psych.colorado.edu/~oreilly/PDP++/PDP++.html`'.

For bug reports (which we will accumulate and fix), write to '`pdp++bugreport@cnbc.cmu.edu`'. Again, there is no guarantee that your bug will be fixed. However, sending in a complete description of the actions that lead to the bug so we can reproduce it is essential for anything to be done to fix it. Also see the '`TODO`' files in the top level and various sub-directories (ta and css) for things we are planning to do.

2 Installation Guide

This chapter provides a guide to installing the PDP++ software. There are two basic forms in which the software is distributed — the executable files only for use by an "end user", and the complete source code, for use by a "programmer" who will be compiling new additions to PDP++. We will refer to the executable files only distribution as the "end user's version", and the source code distribution as the "programmer's version", where the version refers to the manner in which the software is distributed, not to the software itself (it's all the same code).

For most systems, the end user's distribution is obtained in two parts, a tar file containing pre-compiled binaries for a particular system, and another which contains the manual, demos, default files, and other miscellaneous things. This is for the end user who will not need to compile new versions of the software to add new functionality to it. The relevant tar files are:

```
pdp++_version_bin_CPU.tar.gz
pdp++_version_ext.tar.gz
```

where *version* is the version number of the software release, and *CPU* is the cpu-type of the system you will be running on (see below).

For LINUX systems under RedHat or compatible distributions, an rpm version (with the '`.rpm`' extension) is available.

For MS Windows, a standard `setup` install file (with the '`.exe`' extension) is available.

For Mac OS-X, a standard "package" file (with the '`.pkg.sit`' extension) is available.

The currently supported *CPU* types (listed in rough order of level of support) are:

`LINUX` An Intel 386-Pentium machine running a modern glibc version of Linux (e.g., RedHat 7+).

`CYGWIN` An Intel 386-Pentium machine running the Windows operating system (using the Cygnus CygWin system as a compilation environment).

`DARWIN` A Mac running OS-X (aka Darwin), which is based on FreeBSD. Binaries are for the power PC (PPC) architecture.

`SUN4` A SUN sparc-station system running a modern 5.x Solaris version of the operating system.

`SGI` A Silicon Graphics workstation running a recent Irix 6.x release.

`HP800` A Hewlett Packard workstation running HP-UX version 10.x.

`IBMaix` An IBM RS/6000 machine running AIX v4.1.4 (4.1.x shouldwork)

If you don't have one of these machines, then you will have to compile the software from the source code using the programmers distribution.

The programmer's distribution is contained in one tar file that contains the source code along with the manual and other supporting files:

```
pdp++_version_src.tar.gz
```

These and any other files mentioned below can be obtained from our anonymous FTP servers:

```
CMU FTP Site:            ftp://cnbc.cmu.edu/pub/pdp++/
Colorado FTP Site:   ftp://grey.colorado.edu/pub/oreilly/pdp++/
European (UK) Mirror:    ftp://unix.hensa.ac.uk/mirrors/pdp++/
```

The Colorado site is updated most frequently.

2.1 Installing the End User's Version

After downloading the two tar files, '`pdp++_version_bin_CPU.tar.gz`' and '`pdp++_version_ext.tar.gz`', you need to decide where to locate the files. It is recommended that you put them in '`/usr/local/pdp++`', but they can be put anywhere. However, the `PDPDIR` environmental variable must then be set for all users to the location it is actually installed in. In addition if your CPU supports shared libraries (all unix versions, including LINUX, IBMaix, SUN4, HP800, SGI, but not DARWIN), you will need to insure that the `LD_LIBRARY_PATH` environment variable includes the path `PDPDIR/lib/CPU` where PDPDIR is the location of the pdp++ distribution, and CPU is your system type as described above (more details on this below). The following will assume that you are installing in '`/usr/local/pdp++`'.

Note: all of the PDP++ software is distributed in the gnu '`gzip`' format, and it also uses gzip to automatically compress and decompress the network, project, and environment files so that they take up less space on your disk. Thus, your system must have '`gzip`' installed before proceeding. It can be obtained from the GNU ftp server ('`gnudist.gnu.org`') or one of its mirrors, and is typically installed on most modern systems anyway.

Go to the '`/usr/local`' directory, and issue the following command:

```
gzip -dc <tarfile> | tar -xf -
```

or, on Linux or other systems having a gnutar program

```
tar -xzf <tarfile>
```

where <*tarfile*> is the name of the tar archive file. Note that the tar files will create the pdp++ directory, or load into it if it already exists. Thus, if you have an old version of the software, be sure to rename its directory something else before loading the new files.

LINUX users: There is a special '`.rpm`' file that will install the LINUX binaries and ext tar contents, including making links to the binaries in /usr/local/bin, and installing the libIV.so library (and links) in /usr/local/lib, and all of the ext-tra stuff in /usr/local/pdp++. To install this file, you need to be super-user, and then execute the following command:

```
rpm -Uvh pdp++-binext-VERSION.i386.rpm
```

Note that the PDP++ specific libraires are still installed in PDPDIR/lib/LINUX, so you still need to set the LD_LIBRARY_PATH to include this path.

Windows users (CYGWIN): There is a special '`.exe`' file that is an auto-installing executable distribution of both the bin and ext tar files described above. This should be used to install under windows. If you should install it in a location other than the default '`C:\PDP++`' directory, you should add a `set PDPDIR=path` in the '`C:\autoexec.bat`' file.

OS-X users: There is a special '`.pkg.sit`' file that is an auto-installing package file distribution of both the bin and ext tar files described above.

All further references to file names, unless otherwise stated, assume that you are in the PDPDIR directory (e.g., '`/usr/local/pdp++`').

The files will get loaded into the following directories:

'`bin`' binaries (executable files) go here

'`config`' configuration (for Makefile) and some standard init files are found here

'`css`' contains include files for commonly-used css scripts and some additional documentation, plus some demo script files

'`defaults`'
contains default configuration files for the various executables (see the manual for more information).

'`demo`' contains demonstrations of various aspects of the PDP++ software.

'`manual`' contains the manual, which is in texinfo format and has been made into a .ps, emacs .info, and html files.

'`src`' contains the source code for the software.

'`lib`' libraries (for dynamically linked executables) go here

'`interviews/lib`'
InterViews toolkit libraries (for dynamically linked executables) go here.

The binaries will get unloaded into '`bin/CPU`', where `CPU` is the system name as described above. The binaries are:

'`bp++`' The backpropagation executable (see Chapter 14 [bp], page 226).

'`cs++`' The constraint satisfaction executable (see Chapter 15 [cs], page 245).

'`so++`' The self-organizing learning executable (see Chapter 16 [so], page 256).

'`bpso++`' A combination of backpropagation and self-organization algorithms, so hybrid networks can be built.

'`leabra++`'
The Leabra algorithm developed by O'Reilly, which incorporates Hebbian and error-driven learning, together with a k-Winners-Take-All competitive activation function, into a single coherent framework, which is biologically based. See "Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain", by O'Reilly and Munakata, MIT Press, 2000 (September) and associated simulations Chapter 1 [intro], page 2 for details.

'`lstm++`' The long-short-term-memory algorithm by Hochreiter, Schmidhuber et al.

'`rns++`' The real-time neural simultation program developed by Josh Brown.

'`maketa`' The type-scanner used for programming the software. You can read about it in Chapter 18 [prog], page 282.

'`css`' A stand-alone version of the CSS script language system. It can be used as an interpreted C++ language system for any number of tasks.

You should either add the path to these binaries to your standard path, or make symbolic links to these files in '`/usr/local/bin`' or some similar place which most user's will have on their path already. For example, in a csh-like shell (e.g., in the '`~/.cshrc`' file that initializes this shell), add (for the LINUX CPU):

```
set path = (/usr/local/pdp++/bin/LINUX $path)
```

or to make the symbolic links, do:

```
cd /usr/local/bin
ln -s /usr/local/pdp++/bin/LINUX/* .
```

Configuring the Libraries

IMPORTANT: Most of the binaries are dynamically linked, which means that the '`pdp++_version_bin_CPU.tar.gz`' file installed some dynamic libraries in the '`PDPDIR/lib/CPU`' directory and in the '`PDPDIR/interviews/lib/CPU`' directory. When one of the PDP++ programs is run, it will need to know where to find these dynamic libraries. Thus you must set the `LD_LIBRARY_PATH` environmental variable (using `setenv` under csh/tcsh) to point to both of these locations. For example, under LINUX with the standard PDPDIR:

```
setenv LD_LIBRARY_PATH /usr/local/pdp++/lib/LINUX:/usr/local/pdp++/interviews/lib/LINU
```

It is a good idea to put this setting in your initialization file for your shell (i.e. '`~/.cshrc`').

It might be easier, especially if you want to use idraw or other programs available under interviews, to copy the '`PDPDIR/interviews/lib/CPU/libIVhines.so*`' (or .sl for HP800) file into your '`/usr/local/lib`' or somewhere else that is already on your dynamic linker's path.

ADDITIONAL STEPS FOR libIV: First, note that we are now using (as of version 3.0) the version of InterViews maintained by Michael Hines as part of the NEURON detailed neural simulation package. This library is now called '`libIVhines`'. The latest version should always be available at '`ftp://www.neuron.yale.edu/neuron/unix/`', with the current version being '`iv-15.tar.gz`' (also available on the PDP++ ftp sites).

The interviews library in '`PDPDIR/interviews/lib/CPU`' is called libIVhines.so.3.0.3 (or possibly other numbers instead of 3.0.3), but on several unix systems (including LINUX, SGI, SUN4) the linker also wants to see a libIVhines.so.3 and a libIVhines.so as different names for this same file. Therefore, you need to do the following in whatever directory you end up installing libIVhines.so.3.0.3 (even if you keep it in the original location, you need to do this extra step):

```
ln -s libIVhines.so.3.0.3 libIVhines.so.3
ln -s libIVhines.so.3.0.3 libIVhines.so
```

(replace 3.0.3 and .3 with the appropriate numbers for the libIVhines file you actually have). Note that the .rpm install under LINUX does this automatically.

Manual

The manual is distributed in several versions, including a postscript file that can be printed out for hard-copy, a set of "info" files that can be installed in your standard info file location and added to your '`dir`' file for reading info files in gnu emacs and other programs, and a directory called '`html`' which contains a large number of '`.html`' files that can be read with '`Netscape`', '`Mosaic`' or some other WWW program. Point your program at '`pdp-user_1.html`' for the chapter-level summary, or '`pdp-user_toc.html`' for the detailed table of contents.

Help Viewer Configuration

There is now a `Help` menu item on all of the objects (under the `Object` menu), which automatically pulls up the appropriate section of the '`html`' version of the manual, using '`netscape`' by default.

Under Windows (CYGWIN), the default help command is setup to use '`C:/Program Files/Internet Explorer/iexplorer.exe`' — if you prefer netscape or any other HTML browser, this should be changed to the full path to that executable.

Under Mac (DARWIN), the default help command is '`open -a \"Internet Explorer\" %s &`' – this should work for most systems, although you might want to use the new browser whatever it is called.

These defaults can be changed in the `Settings` menu of the `PDP++Root` object, see Section 9.5 [proj-defaults], page 126 and Section 6.6 [gui-settings], page 63 for details.

The latest version of the manual is also available on-line from:

`http://psych.colorado.edu/~oreilly/PDP++/PDP++.html`

MS Windows Configuration

Memory Configuration

The PDP++ executables are compiled using an environment called cygwin developed by Cygnus Solutions (now owned by Red Hat Software). The default cygwin configuration has an upper limit of 128MB, which should work for most simulations. However, you might want to increase this limit if you are exploring larger simulations.

Open regedit (or regedt32) and find the key HKEY_CURRENT_USER\Software\Cygnus Solutions\Cygwin\

If this does not exist, you must create a new Key called Cygnus Solutions, and then another within it called Cygwin.

Then, create a new DWORD value under this key called "heap_chunk_in_mb" that contains the maximum amount of memory (in Mb) your application needs (watch the hex/decimal toggle – you'll probably want to set it to decimal). For example if you wanted to set the memory limit to 256Mb, just enter 256. Exit and restart all cygwin applications (e.g., pdp++).

Taskbar Configuration

Because the simulator uses many windows, the windows taskbar often does not adequetly display the names of the windows. This can be remediated by dragging the top of the bar up, allowing more room for each icon. Another approach is to grab the taskbar, drag it to the right edge of the screen, drag the left edge to widen it, and then set it to auto hide (right click on the taskbar, and select Properties to expose this option, or go to the Start menu/Settings/Taskbar). When all this has been done, the window list can be exposed bringing the pointer to the right edge of the screen.

Mac OS-X Configuration

PDP++ depends on having an XWindows (X11) server running on your mac. Apple now has their own version of the XFree86 X11 server, which runs very smoothly under the standard OSX window manager. This is the recommended solution. Read more about it and download from:

`http://www.apple.com/macosx/x11/`

Basically, everything works just as under unix because this is a fairly standard unix setup after all is said and done. The manual should therefore provide all the info you need.

Other Configuration

See Section 9.5 [proj-defaults], page 126 for instructions on how to setup customized startup files if you want to change some of the default properties of the system.

Happy simulating!

2.2 Installing the Programmers Version

Read over the instructions for installing the end-user's version first. This assumes that you have unloaded the '`pdp++_version_src.tar.gz`' file in something like '`/usr/local/pdp++`'.

IMPORTANT: Whenever you are compiling, you need to have the environmental variable `CPU` set to reflect your machine type (see above). Other machine types can be found in the interviews '`config/InterViews/arch.def`' directory. These are (in addition to the above): `VAX`, `MIPSEL`, `SUN3`, `SUNi386`, `SUN`, `HP300`, `HP200`, `HP500`, `HP`, `ATT`, `APOLLO`, `SONY68`, `SONYmips`, `SONY`, `PEGASUS`, `M4330`, `MACII`, `CRAY`, `STELLAR`, `IBMi386`, `IBMrt`, `IBMr2`, `LUNA68`, `LUNA88`, `MIPSEB`, `MOTOROLA`, `X386`, `DGUX`, `CONVEX`, `stratus`, `ALPHA`.

The C++ compiler types that are supported are modern gnu g++/egcs compilers (anything released after 1999 seems to work), and proprietary system compilers, which we refer to as '`CC`' compilers.

There are a couple of other libraries that the PDP++ software depends on. These need to be made before PDP++ itself can be compiled. Please ensure that all of the following are installed properly:

1) The '`readline`' library, which will have already been installed if '`gdb`' or perhaps other gnu programs have been installed on your system (or if you are using Linux). Look for '`/usr/lib/libreadline.so`' or '`/usr/local/lib/libreadline.so`'. If it isn't there, then download a version of it from one of the gnu ftp server sites (e.g., '`gnudist.gnu.org`'), and compile and install the library.

2) If using g++, and not on a Linux-based system, you need to make the the '`libstdc++`' library in the libg++ distribution. **NOTE:** PDP++ now requires the `sstream` header file, which defines the `stringstream` class, which is a much improved replacement for the `strstream` class. **For g++ version 2.9x, this header file might be missing or broken (e.g., under RedHat 7.3, it is present but broken!).** A good version of this file is present in `PDPDIR/include/new_sstream_for_gcc_2.9x.h`, and should be installed in the appropriate location (e.g., for RH 7.3, do the following as root: `cp include/new_sstream_for_gcc_2.9x.h /usr/include/g++-3/sstream`.

I haven't done this install in a while so the following is likely out of date:. It seems that in the latest distribution of libg++ both of these are installed in '`/usr/local/lib`' automatically, but if they are not there, '`libiostream.so`' is made in the '`libio`' directory in the libg++ distribution (do a `make install` to get the properly installed or copy it yourself), and '`libstdc++.so`' is made in the '`libstdc++.so`' directory. CC/cfront users should have their iostream code linked in automatically via the standard C++ library that comes with the compiler.

3) Install the InterViews library, which provides the graphics toolkit used by PDP++. We have collaborated with Michael Hines, developer of the Neuron simulation system, in developing an improved version of the InterViews library – **You must install the Hines version of InterViews!**. The source code for this version is:

```
iv-15.tar.tz
```

available on our ftp servers, and the latest version should be available from Michael Hines' ftp site at: '`ftp://www.neuron.yale.edu/neuron/unix/`'. Some miscellaneous information about interviews can be found in the '`PDPDIR/lib/interviews`' directory.

An alternative to compiling interviews yourself is to download pre-compiled interviews libraries from us. These are available for the dominant form of compiler (CC or g++) for the platforms on which the binary distribution is available (see list above in Chapter 2 [inst], page 4) and are provided as '`pdp++_version_ivlib_CPU_CC.tar.gz`' for the CC compiler and '`..._g++.tar.gz`' for g++. The include files, which are necessary to use the libraries to compile PDP++, are in '`pdp++_version_ivinc.tar.gz`'. These create a directory called '`interviews`' when extracted, which means this should be done in the '`/usr/local`' directory so that the interviews directory is '`/usr/local/interviews`'. Alternatively, these can be installed elsewhere and the `IDIRS_EXTRA` and `LDIRS_EXTRA` makefile variables set to point to this directory (see below). We install ours in '`/usr/local/lib`' – follow the directions in the End Users's install version described above to do this.

4) Once you have the above libraries installed, the next step is to configure the makefiles for the type of compiler and system you have. These makefiles are located in '`PDPDIR/config`'. The actual makefiles in a given directory (e.g., '`src/bp/Makefile`' and '`src/bp/CPU/Makefile`') are made by combining several makefile components. '`config/Makefile.std`' has the standard rules for making various

things, '`config/Makefile.defs`' has the standard definitions for everything, and then '`config/Makefile.CPU`' overrides any of these definitions that need to be set differently for a different CPU type (i.e., "local" definitions). There are several sub-steps to this process as labeled with letters below:

a) If you have installed the software in a location other than '`/usr/local/pdp++`': You need to change the definition of `PDPDIR` in **both** the top-level Makefile ('`PDPDIR/Makefile`') and in the definitions makefile '`PDPDIR/config/Makefile.defs`'.

There are makefiles in '`PDPDIR/config`' for the supported CPU types listed above, and the two different supported compilers (g++, CC). The makefiles are named '`Makefile.CPU.cmplr`', where cmplr is either '`g++`' or '`CC`'. The actual makefile used during compiling for a given machine is the one called '`Makefile.CPU`', where CPU is the type of system you are compiling on (e.g., LINUX, SUN4, HP800, SGI, etc.). If you are compiling on a machine for which a standard makefile does not exist, copy one from a supported machine for the same type of compiler. Also, see the notes below about porting to a new type of machine.

b) Copy the appropriate '`Makefile.CPU.cmplr`' makefile (where cmplr is either g++ or CC depending on which compiler you are using), to '`Makefile.CPU`' (again, CPU is your machine type, not 'CPU'). For some architectures there is just one '`Makefile.CPU`' since only one type of compilier is currently supported. In this case you can just leave it as is.

c) You should put any specific "local" definitions or modifications to the makefiles in the '`config/Makefile.CPU`'. This will be included last in the actual makefiles, and any definitions appearing here will override the standard definitions. To see the various definitions that might affect compiling, look at '`Makefile.defs`', which contains all the "standard" definitions, along with descriptive comments. The following are items that you will typically have to pay attention to:

i) `IOS_INCLUDES` and `IOS_LIB`: In order to be able to access via the CSS script language the functions associated with the standard C++ iostream classes, the type-scanning program '`maketa`' needs to process the iostream header files: '`streambuf.h`', '`iostream.h`', '`fstream.h`', and '`strstream.h`'. These files are scanned in the '`src/ta`' directory, as part of the building of the type access library '`libtypea.a`'. These header files are different depending on the compiler being used. For CC compilers, the `IOS_INCLUDES` variable should be set to `CC-3.1`.

These header files are typically located in '`/usr/include/CC`', which is where the `CC-3.1` versions of these files in '`src/ta`' point to via symbolic links. Thus, if your headers are located elsewhere, you will need to change these symbolic links, or just copy the header files directly into the '`ios-CC-3.1`' subdirectory in '`src/ta`'. For g++ users, `IOS_INCLUDES` variable should be set to `g++-2.8.1` (for g++ 2.9x) or `g++-3.1` (for g++ 3.x). As of version 3.0, these g++ iostream headers are never actually included in the compile process itself, and are only scanned via the maketa program. Therefore, they have been dramatically edited to expose only the relevant interface components. As such g++-3.1 should work for all subsequent releases of g++ (hopefully!).

Note that you can use the `make force_ta` action to force a re-scan of the header files. A `make opt_lib` is then neccessary to compile this type information into the library. Finally,

the `IOS_LIB` variable should be blank by default for both g++ and C++ users (for newer g++), but for older g++ configurations it was necessary to set it to `-liostream`.

ii) `IDIRS_EXTRA` and `LDIRS_EXTRA` can be used to specify locations for other include and library files, respectively (for example, the cfront compiler may need to be told to look in -I/usr/include/CC for include files). Use these if you have installed any libraries (e.g., InterViews) in a non-standard location.

iii) `MAKETA_FLAGS` should be set to `-hx -css -instances` by default, but it is also often necessary to include the include path for the iostream and other C++ library files. For example, the LINUX makefile has the following: `LIBG++_INCLUDE_DIR = -I/usr/include/linux -I/usr/include/g++-3`, and then `MAKETA_FLAGS = -hx -css -instances $(LIBG++_INCLUDE_DIR)`.

iv) `cppC`: this is the c-pre-processor, needed for the '`maketa`' program to process header files. Although the system default preprocessor, usually installed in '`/usr/lib/cpp`', should work, '`maketa`' was developed around the gnu cpp program, and so if you run into difficulties using the system cpp, install the gnu one (included as part of the gcc compiler). Note that `make install` of gcc/g++ does not apparently install this program by default, so you have to manually copy it from either your gcc compile directory or '`/usr/lib/gcc-lib/<machine>/cpp`'. You often need to include a define for the system architecture (e.g., `-D__i386__` for Linux on intel chips, or `-Dsparc` for suns) in the cppC command.

v) `VT_xxx` and `TI_xxx`: these specify the virtual-table instantiation and template-instantiation (respectively) files, which are needed by different compilers. CC typically requires the VT_xxx files along with the +e[01] flags to only make one copy of virtual tables, while g++ requires the TI_xxx files to only make one copy of the templates. The TI_xxx files are included by default, so you will need to define them to be empty to override this default: `TI_INST_SRC = , TI_INST_OBJ = , TI_INST_DEP = .`

vi) Porting to a non-supported machine: There are a small set of system-dependent definitions contained in '`src/ta/ta_stdef.h`', which are triggered by defines set up in the makefiles. `NO_BUILTIN_BOOL` should be defined if the c++ compiler does not have a bulitin `bool` type, which is the case with most cfront-based CC compilers, but not g++. `CONST_48_ARGS` determines if the `seed48` and `lcong48` functions take const arguments or not. In addition, different platforms may require different defines than those that are flagged in '`ta_stdef.h`'. In this case, you will have to edit '`ta_stdef.h`' directly. Please send any such additions, and the corresponding '`config/Makefile.CPU`' along with any notes to us ('`pdpadmin@crab.psy.cmu.edu`') so we can put them on our web page for others to use, and incorporate them into subsequent releases.

vii) For some more information about the makefiles, see Section 18.1 [prog-make], page 282.

5) The standard makefiles use gnu's '`bison`' instead of '`yacc`' for making parsers. If you don't touch any of the .y files in the distribution, you won't need either. If you plan on messing around with the guts of the maketa type scanner or CSS, then you will probably want to install the latest version of '`bison`'.

6) The dependency information, which is essential if you are going to be editing the main body of PDP++ code, but not necessary for a one-pass make of the system, is not made by

the default `make world` action. If you want to make this dependency information, do it with a `make depend` after a successful `make world`. Also, note that the automatic dependencies are made by calling '`gcc`' in the standard configuration. If your local C compiler supports the `-M` flag for generating dependency information, then this can be used instead. Just change the definition for `CC` in your '`Makefile.CPU`'. If you don't have gcc and your local C compiler doesn't support this, you can edit the end of the '`Makefile.std`' and change it to use the '`makedepend`' program, which we have not found to work as well, but it is an option.

7) On some systems, the standard '`make`' program is broken and will not work with our complex makfile system. This is true of the SUN4 system and IBMaix, and may be true of others. In this case, you will have to install the GNU make program, and use it to compile the software. If you get inexplicable errors about not being able to make certain things (seems to be the .d dependency files in particular that cause a problem), then try using GNU make (again, available at '`gnudist.gnu.org`' or mirrors).

8) If your CPU supports shared libraries (most do now), you will need to insure that the `LD_LIBRARY_PATH` environment variable includes the path `PDPDIR/lib/CPU` nad `PDPDIR/interviews/LIB/CPU` where PDPDIR is the location of the pdp++ distribution, and CPU is your system type as described above. **This is important even during the compile process, because the maketa program will need to access shared libs**.

After setting appropriate definitions, go back up to the PDPDIR and just do a:

```
make world
```

this should compile everything. This will make makefiles in each directory based on your CPU type, and then compile the various libraries and then the executables.

Most likely, the make will at least proceed past all the basic directory initialization stuff that is part of '`make world`'. Thus, if the compile stops after making the makefiles and after making the '`maketa`' program, you can fix the problem and re-start it by doing '`make all`' instead of '`make world`'.

If you run into difficulties during the compile process, the programming guide might contain some useful information for debugging what is going wrong: Chapter 18 [prog], page 282.

3 Conceptual Overview

This chapter provides a general introduction to the PDP++ software and conceptual overview of its major components. The primary objective here is to explain the important design decisions that went into the software, so that you will be able to get the most out of it. This is oriented towards those who have some familiarity with neural networks, and who want to find out how things are done in this software. However, it is also useful for all users to be aware of the rationale for the way things are set up, so that it becomes clearer where to look for ways to solve problems that might arise, or accomplish various simulation objectives.

3.1 Overview of Object-Oriented Software Design

The PDP++ software is based on the principles of object-oriented software design. The effects of this are pervasive, and an understanding of these basic principles can help the user to understand the software better.

First of all, everything in the software is an object: units, connections, layers, networks, etc. While this sounds intuitive for these kinds of objects, other kinds of objects, like the object that controls the way a network is viewed, or the ones that control the way the network is trained, are somewhat less intuitive. However, if you just think of an object as representing some discrete piece or aspect of the software, it makes more sense. Essentially the PDP++ software is just a big collection of objects that work together to let you get things done.

The primary advantage of an object-oriented design is its *flexibility*. Because the functionality is broken up into object-sized pieces, these pieces can be combined in many different ways to accomplish many different tasks. However, this power comes at the price of *simplicity*—since everything is accomplished by the interactions between multiple objects, setting things up to perform any given task requires knowing which objects to use and how to configure them. The very distributed nature of the processing makes it less easy to identify one single thing that needs to be operated on in order to do something. Its like the difference between a neural network and a conventional computer program, in some sense. In a computer program, things proceed in a relatively linear, straightforward fashion from one step to the next, whereas in a neural network, processing is distributed among many different units that all work together to get something done.

While the tradeoff between flexibility and simplicity is inevitable to a certain extent, we have endeavored to make the commonly-used tasks simpler by including menus or buttons that do all of the right things in one simple step. However, as you become a more advanced user and want to perform tasks that have not been "pre-compiled" in this way, expect to encounter some of the difficulties associated with object-oriented software.

3.2 Overview of the Main Object Types

This section describes the main ways in which the many tasks involved in neural network simulation have been divided up and assigned to different types of objects. It is essential to understand this division of labor in order to use the software effectively.

There are four central components to a neural network simulation as we see it:

- The **Network**, including layers, units, connections, etc. (see Chapter 10 [net], page 131).
- The **Environment** on which the network is to be trained and tested (see Chapter 11 [env], page 163).
- The **Processing** or **Scheduling** of training and testing a network, which determines how long to train the network, with what environment, etc. (see Chapter 12 [proc], page 184).
- The **Logging** of the results of training and testing. One can view these results as a graph, a table of numbers, or a grid of color chips that represent values in a graphical similar to that used in the viewing of networks (see Chapter 13 [log], page 210).

Since all of these components work together in a given simulation, we have grouped them all under a **Project** object (see Chapter 9 [proj], page 119), which also has some additional objects (defaults and scripts) that make life easier. Thus, when you start a simulation, the first thing you do is create a new project, and then start creating networks, environments, etc. within this project.

New for 2.0: There is now a graphical interface for the project-level organization of objects, which can really help make complex projects more managable, especially for dealing with the processing aspects of things. Just enlarge your project window on existing projects to see the new view.

New for 3.0: The complexity of dealing with distributed objects is partially simplified through the introduction of a **Wizard** object that automates the contstruction of typically-used network simulation objects. See Section 5.5 [how-wizard], page 51 for details. Also new are principal components analysis (PCA) and multidimensional scaling (MDS) techniques for analyzing network representations (see Section 11.9 [env-analyze], page 179), and much imporoved graph log features for displaying overlapping traces of data, including a spike raster plot. In addition, distributed memory parallel processing support, both for connection-level and event-level parallelism, was added.

Of the different simulation components, Processing (aka Scheduling) is probably the least intuitive. Indeed, we might have decided instead that networks knew how to train themselves, or that environments would know how to train networks. The main reasons we chose to make Processing a separate major category of function in the software are: 1) The ways in which one trains and tests a network are common to different types of networks and learning algorithms. 2) Many different kinds of training and testing processing can be performed on the same network and environment. 3) Processing depends in part on the nature of the network and the nature of the environment, so that putting this task in either of these separately would lead to strange dependencies of networks on environments or vice versa.

Thus, one can think of the processing function as being similar to that of a movie director, who coordinates the actors and the camera operators, set designers, etc. to produce a finished product. Similarly, the processing coordinates the network with the environment and other elements to direct the overall process of training and testing. For more information on how processing is implemented in the software see Section 12.2 [proc-sched], page 186.

3.2.1 The Objects in Networks

For the most part, the objects that make up a network are fairly intuitive, consisting of Layers (see Section 10.2 [net-layer], page 135), Units (see Section 10.4 [net-unit], page 143), and Connections (see Section 10.5 [net-con], page 146). However, there is an additional type of object which describes the pattern of connectivity between all of the units in one layer to those in another layer, which is called a Projection (see Section 10.3 [net-prjn], page 137). The Projection is very useful because it allows one to specify connectivity at a broad level, instead of individually connecting each unit with other units. It also seems to be a conceptually real entity—one often thinks about the connectivity in terms like, "the hidden layer is fully connected to the input layer."

The use of Projection objects is illustrative of a general principle regarding network structure, which is that the larger-scale objects (Layers and Projections) contain *parameters* for how to construct the smaller-scale objects (Units and Connections). This allows one to create Layers and Projections, fill in some parameters regarding the type of connectivity and number and type of units, and execute general-purpose `Build` and `Connect` (see Section 10.1 [net-net], page 131) functions which translate the parameters into an actual network with interconnected units.

While it is possible to manually build a network unit-by-unit and connection-by-connection, it is usually much easier to use the parameters-and-build/connect method. However, some people find the distinction between specifying and actually building difficult to work with, in part because only after you execute `Build` and `Connect` does your actual network reflect the parameters you have entered. We have tried to lessen these difficulties by providing graphical representations of the parameters, so that you can see what you are specifying at the time you do it, not only after you have built the network.

3.2.2 The Objects in Environments

We decided to call the training and testing data that is presented to a network an `Environment` (see Chapter 11 [env], page 163) to capture the idea that a network is like an organism that exists and interacts within a particular environment. Thus, the environment object describes a little world for the network to roam around in. It does not, however, specify the path that the network takes in its travels (i.e., the order in which the events are presented to the network)—this is determined by the scheduling objects, which can be thought of as "tour guides" in this context. To switch metaphors, an environment should be thought of as a library, which contains lots of data, but does not tell you which books to check out when.

The `Environment` is composed of `Event`s (see Section 11.2 [env-event], page 166), which represent a collection of stimuli or `Pattern`s that go together. Thus, in the context of a typical backpropagation network, an `Event` consists of an input pattern and a target pattern for a single training example. `Pattern`s contain an array of values which get mapped onto the units in a particular layer of the network.

In order to represent *sequences* of events that should be presented sequentially in time, events can be grouped together (see Section 11.3 [env-seq], page 169, see Section 3.5 [overgroup], page 19). Note, however, that the scheduling objects need to be made aware of the

fact that the events in a group represent a sequence of events in order for these to actually be presented to the network in the right order (see Section 12.4.1 [proc-special-seq], page 195).

Environments can also be defined which dynamically create events on-line, possibly in response to outputs from the network. This must be done under the supervision of an appropriate Schedule process, which can coordinate network outputs with the parameters that determine how the environment creates events.

New for 2.0: The Environment interface has been completely rewritten, and now enables interactive configuration of event structures, just like in the Network interface.

3.2.3 The Objects Involved in Processing/Scheduling

In order to make the scheduling aspect of the software reasonably flexible, it has itself been decomposed into a set of different object types. Each of these object types plays a specific role within the overall task of scheduling the training and testing of a network.

The backbone of the scheduling function is made up of a structure hierarchy of objects of the type `SchedProcess` (see Section 12.2 [proc-sched], page 186), which reflects the hierarchy of the different time grains of processing in a neural network:

```
Training (loop over epochs)
    Epoch (loop over trials)
        Trial (present a single pattern)
        .
        .
    .
    .
```

Thus, there is one object for every grain of processing, and each process essentially just loops over the next-finer grain of processing in the hierarchy. Because each grain of processing is controlled by a separate process, it is possible to alter both the parameters and type of processing that takes place at each grain *independently*. This is why this design results in great flexibility.

Since this time grain structure forms the backbone of scheduling, all other aspects of scheduling, and anything that depends on time grain information in any way, are oriented around it. For example, setting the training criteria becomes embedded in the Training level of processing. The updating of displays is associated with a particular time grain of processing (e.g., telling the system that you want to see the activations updated in your view of a network after every trial means associating the view with the Trial grain of processing). Similarly, the generation of information to be logged, and the update of the display of this information, is tied to a particular time grain.

The other principal type of object used for scheduling is the *statistic* (of type `Stat`, see Section 12.5 [proc-stat], page 199), which is responsible for computing (or simply recording) data of any kind that is either relevant for controlling scheduling (e.g., stopping criteria), or is to be displayed in a log at a particular time grain. Thus, statistics are the source of any data that is to be logged, and provide a general way of specifying when to stop processing based on the values of computed variables (i.e., sum-squared error). Again, because these statistic objects can be mixed and matched with all sorts of different schedule processes, this creates a high level of flexibility.

Because they are intimately tied to the time grain of processing, and can directly affect its course, statistics actually "live" on schedule processes. Thus, whenever a statistic is created, it is created in a particular process that governs a given time grain of processing. Furthermore, statistics can create copies of themselves that reflect their *aggregation* over higher levels of processing. For example, the sum-squared error statistic (see Section 12.6.1 [proc-stats-se], page 202) is created at the Trial level of processing, where it can actually compute the difference between the output and target for a given pattern right after it has been presented. However, in order for these individual pattern-level sse values to be accumulated or aggregated over time, there is another sse statistic at the Epoch level of processing whose only job is to add up the Trial level sse values.

3.3 Separation of State from Specification Variables

Another major distinction that is made in the software reflects the separation of *state* variables (e.g., the current activation of a unit) from *specification* variables (e.g., the activation function to be used in computing activation states, and its associated parameters). This distinction is made because it is often desirable to have many objects use the same specifications, while each retain a distinct set of state variables. For example, all of the units in one layer might use a particular learning rate, while units in another layer have a different one. By separating these specifications from the state variables, it is easy to accomplish this.

Thus, associated with many different kinds of objects is a corresponding `Spec` object (e.g., a `UnitSpec` for `Unit` objects, etc). While this separation does sometimes make it a little more difficult to find the parameters which are controlling a given object's behavior, it makes the software much more flexible. Furthermore, we have put most of the specification objects (specs) in one place, so it should soon become second nature to look in this place to set or change parameters. For more information about specs, see Section 8.4 [obj-spec], page 115.

Specs are also used to control the layout of events and patterns in the environment (throught the use of `EventSpec` and `PatternSpec` objects). Thus, one first configures these specs, and then creates events according to them. A new interface makes configuring these specs much easier than before.

3.4 Scripts: General Extensibility

It is impossible to anticipate all of the things that a user might want to do in running and controlling a simulation. For this reason, we have built a powerful script language into the PDP++ software. This script language, called CSS for C-Super-Script or C^C, is essentially a C/C++ interpreter, and thus does not require the user to learn any new syntax apart from that which would be required to program additions to the software itself. This shared syntax means that code developed initially in the script language can be compiled into the software later for faster performance, without requiring significant modifications.

CSS scripts are used in several places throughout the software. They have been added to some of the basic objects to enable their function to be defined and extended at run time. Examples of these include a **ScriptEnv**, which allows an **Environment** to be defined dynamically and have events that are generated on the fly, potentially in response to the

output of the network itself (see Section 11.11 [env-other], page 182). The **ScriptPrjnSpec** allows arbitrary patterns of connectivity to be defined (see Section 10.3.3.5 [net-prjn-misc], page 143). The **ScriptStat** allows arbitrary statistics to be computed and recorded in the data logs during training and testing (see Section 12.6.8 [proc-stats-misc], page 207).

In addition to these targeted uses, scripts can be used to perform miscellaneous tasks or repetitive functions at any level of the software. It is possible to attach a set of scripts to a given process, run them automatically when the process starts up, and even record actions performed in the user interface as script code that can be replayed later, edited for other uses, etc. For more information on these objects, see Section 9.6 [proj-scripts], page 128.

Finally, the script language provides a means for interacting with a simulation when the graphical interface cannot be used, such as over dial-up lines when working from home, etc. Anything that can be done in the GUI can be done in the script language, although it is not quite as easy.

3.5 Groups: A General Mechanism for Defining Substructure

Another way in which the PDP++ software has built-in flexibility is in the ability to create substructure at any level. By substructure, we mean the grouping of objects together in ways that reflect additional structure not captured in the basic structural distinctions made in the software (e.g., as described in the previous sections). For example, it is possible to group events together to reflect sequential information. Also, one can imagine that certain layers should be grouped together to reflect the fact that they all perform a similar function, and should be treated as a group. Similarly, units within a layer can be divided into subgroups that might reflect different parameter settings, etc.

The basic operations of the software are written so as to be insensitive to this additional substructure (i.e., they can "flatten out" the groups), which allows the substructure to be used without requiring special-case code to handle it.

Substructure is defined by creating subgroups of objects. Thus, everywhere the user has the opportunity to create an object of a given type, they also have the opportunity to create a subgroup of such objects.

For more information, see Section 8.2 [obj-group], page 109.

4 Tutorial Introduction (using Bp)

In this chapter, Section 4.1 [tut-bp], page 20, Section 4.2 [tut-using], page 24 and Section 4.3 [tut-config], page 33 provide a tutorial introduction in three parts. Section 4.1 [tut-bp], page 20 provides a brief overview of a connectionist learning algorithm, back-propagation, and a simple but classic learning problem that can be solved by this model, called XOR. It is worth reading this section even if you are familiar with these problems, since it introduces some of the conceptual structure behind the simulator and some important terminology, such as the names of many of the variable names used in the simulator. Section 4.2 [tut-using], page 24 shows you how to use the PDP++ system to run backpropagation on the XOR example, indicating how to control the learning process, modify options and parameters, save and load results, etc. Section 4.3 [tut-config], page 33 shows you how to to configure the backpropagation model for an example of you own. To illustrate the configuration process we choose another classic problem, the Encoder problem.

4.1 Backpropagation and XOR

Backpropagation is the name of a learning algorithm used to train multi-layer networks of simple processing units (Rumelhart, Hinton & Williams, 1986). In the simple case we consider in this tutorial, we restrict our attention to multi-layer feed-forward networks. Such networks consist of several layers of units, the first (one or more) of which are the input layer(s) and the last of which are the output layer(s). Each layer consists of some number of simple connectionist units. Units in lower-numbered layers may send connections to units in any higher-numbered layer, but in feedforward networks they cannot send connections to units in the same layer or lower-numbered layers.

The network learns from events it is exposed to by its training environment. Each event consists of an input pattern for each input layer and a target output pattern for each output layer. Henceforth we will consider the case of a single input and output layer. The goal of learning is to adjust the weights on the connections among the units so as to allow the network to produce the target output pattern in response to the given input pattern. Weights changes are based on calculating the derivative of the error in the network's output with respect to each weight.

Training is organized into a series of epochs. In each epoch, the network is exposed to a set of events, often the entire set of events that comprise the training environment. for each event, processing occurs in three phases: an activation phase, a back-propagation phase, and a final phase in which weight error derivatives are calculated.

Training begins by initializing the weights and biases to small random values. The process of learning then begins, and continues for some number of epochs or until a performance criterion is reached. Typically this criterion is given in terms of the total, summed over all of the events in the epoch, of some measure of performance on each event; the sum squared error, described below, is the most frequently used measure.

We now consider in each of the three phases involved in processing each pattern.

4.1.1 Activation Phase

In this phase, the activations of the input units are set to the values given in the input pattern, and activation is propagated forward through the network to set the activations of units at successively higher-numbered layers, ending with the output layer. The idea is that the connection weights determine the outcome of this process, and the purpose of this pass is to determine how closely this outcome matches the target. During this pass, we calculate two quantities for each unit: First the net input it receives from the units that project to it, and second the activation, based on the net input. The net input is simply the sum, over all of the incoming connections to the unit, of the weight of the connection times the activation of the sending unit, plus a bias term:

$$net_i = \sum_i a_j w_{ij} + b_i$$

This sum may run over all of the units in all of the lower-numbered layers, but the connectivity may be restricted by design. the term w_ij signifies the weight to unit i from unit j, and the term b_i signifies the bias associated with unit i. The activation is simply a monotonically increasing sigmoidal function of the net input. The simulator uses the logistic function:

$$a_i = \frac{1}{1 + e^{-net_i}}$$

The activation ranges from 0 to 1 as the net input ranges from minus infinity to infinity. As an option other ranges can be used, in which case the function becomes

$$a_i = min + (max - min)\frac{1}{1 + e^{-net_i}}$$

4.1.2 BackPropagation Phase

In this phase, a measure of the error or mismatch between the target and the actual output is computed, and the negative of the derivative of the error with respect to the activation and with respect to the net input to each unit is computed, starting at the output layer and passing successively back through successively lower numbers of layers. By default the error measure used is the summed square error:

$$ss = \sum_i (t_i - a_i)^2$$

where the index i runs over all of the output units. The derivative of this with respect to the activation of each output unit is easily computed. We actually use the negative of this derivative, which we call *dE/da_i*. For output units, this quantity is

$$\frac{dE}{da_i} = t_i - a_i$$

We then compute a quantity proportional to the negative derivative of the error measure with respect to the net input to each unit, which is

$$\frac{dE}{dnet_i} = \frac{dE}{da_i} f'(net_i)$$

where *f'(net_i)* is the slope of the activation function given the net input to the unit.

This quantity is then propagated back to successively lower numbered layers, using the formulas

$$\frac{dE}{da_j} = \sum_i w_{ij} \frac{dE}{dnet_i}$$

and

$$\frac{dE}{dnet_j} = \frac{dE}{da_j} f'(net_j)$$

Here subscript i indexes units that receive connections from unit j; we can see these formulas as providing for the backward propagation of error derivative information via the same connections that carry activation forward during the activation phase.

4.1.3 Weight Error Derivative Phase

In this phase, we calculate quantities called *dE/dw_ij* and *dE/db_i* for all of the weights and all of the biases in the network. These quantities again represent the negative derivative of the Error with respect to the weights and the biases. The formulae are:

$$\frac{dE}{dw_{ij}} = \frac{dE}{dnet_i} * a_j$$

and

$$\frac{dE}{db_i} = \frac{dE}{dnet_i}$$

After these error derivatives are calculated, there are two options. They may be summed over an entire epoch of training, or they may be used to change the weights immediately. In the latter case, the variable *dE/dw_ij* accumulates the value of this derivative over all of the patterns processed within the epoch, and similarly for *dE/db_i*.

When the weight change is made, there are two further values calculated. First, the exact magnitude of the change to be made is calculated:

$$\Delta W_{ij} = (lrate * \frac{dE}{dw_i j}) + momentum * deltaW_{ij}$$

This expression introduces a learning rate parameter, which scales the size of the effect of the weight error derivative, and a momentum parameter, which scales the extent to which previous weight changes carry through to the current weight change. Note that in the expression for *deltaW_ij* the value of *deltaW_ij* from the previous time step appears on the right hand side. An analogous expression is used to calculate the *deltaBias_i*. Finally, the weight change is actually applied:

$$w_{ij} = \Delta W_{ij}$$

Once again, an analogous expression governs the updating of *bias_i*.

4.1.4 The XOR Problem

XOR is about the simplest problem requiring a hidden layer of units between the inputs and the outputs. For this reason, it has become something of a paradigm example for the back propagation learning algorithm. The problem is to learn a set of weights that takes two-element input patterns, where each element is a 1 or a 0, and computes the exclusive-or (XOR) of these inputs. XOR is a boolean function which takes on the value '1' if either of the input bits is a 1, but not if both are 1:

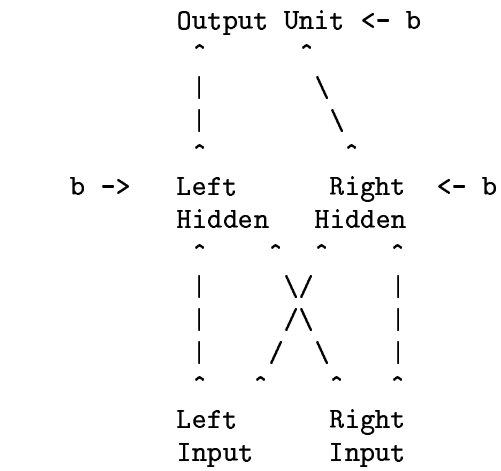
```
Table [XOREvents]:

        Input  Output

         0 0     0
         0 1     1
         1 0     1
         1 1     0
```

The output should be 1 if one of the inputs is 1; if neither is 1 or if both is 1, the output should be 0.

The environment for learning XOR consists of four events, one for each of the input-output pairs that specify the XOR function. In the standard training regime, each pattern is presented once per epoch, and training continues until the total squared error, summed across all four input-output pairs falls below a criterion value. The value is usually set to something like .04. With this criterion, all the none of the cases can be off by more than about .2.

Various network configurations can be used, but for this example we will use the configuration shown in Figure [XORNet]. In this configuration, the input layer consists of two units, one for each input element; the output layer consists of a single unit, for the result of the XOR computation; and the hidden layer consists of two units. The output unit receives connections from both hidden units and each hidden unit receives connections from both input units. The hidden and output units also have modifiable biases, shown in the figure as with arrows labeled with 'b' coming in from the side.

Figure [XORNet]:



4.2 Using the Simulator to run BP on the XOR Example

The brief description of back propagation and XOR just given in section BP introduces key concepts and terminology, including the actual names of the variables used in the simulator. With that background, you are ready to turn to the simulator itself.

4.2.1 Notation for Tutorial Instructions

Before we go any further, we will need a few notational conventions, so let us introduce some. We use *underlined* type for things that you will type. We use `typewriter` for labels and prompts, such as the prompt in the PDP++ shell or a label on a type-in menu boxes. **bold** is used to denote an object type name, like **BpUnit**. We use *italic* for menu items or buttons you will click on or select using the SELECT (usually left) mouse key. For multi-step menu selections we will use statements like 'select *A/B/C*'. This means that you should press and hold the select key on menu item *A*. When a popup menu appears move the pointer to item *B*. When a further popup menu appears, move the pointer to *C*, and then release the select key. Finally, we use double quotes to designate strings that name objects such as files, layers, units, etc.

4.2.2 Starting up PDP++

The XOR demo project that we will be using is located in a directory where the PDP++ software was installed. You should go to that directory (typically '`/usr/local/pdp++`') and then change to the '`demo/bp`' directory, which is where the project files are.

To start using the back propagation package within the PDP++ system, you simply type *bp++* to your unix command interface. After a short time, the `bp++>` prompt will appear in your command interface window, and a small menubar window will appear somewhere on the screen. This is the `PDP++:Root` window, and it contains three menus. The first is called *Object*, the second is called *.projects*, and the third is called *.colorspecs*. The first menu's name alerts you to the fact that this window is associated with an object, whose

name appears as the label on the window. The other menu names refer to the children of Root, which consist of projects and colorspecs. You can think of the '.' in front of the names as indicating that the projects and colorspecs are children of the current object.

You are going to open a pre-defined **Project** and then run it. To open the project, select *.projects / Open In / Root*. A file-selector window will appear. It gives a list of directories accessible from your current directory and files in the current directory that might contain a project. You can either type the name of the project file in the `enter filename` box, or double click on an item in the the list to select it. We want the project file "xor.proj.gz". This file is saved in a compressed format, which is why it has the ".gz" extension. Double click on it, and the project will be loaded in (the simulator automatically uncompresses files stored in the ".gz" compression format).

After a little time for initialization, the Root menu will move to the right, and you should see two new windows on your screen: A **Project** window and a window called a **NetView**. Several other windows will be iconified, we will turn our attention to them in a moment.

4.2.3 Object Hierarchy: The Root Object and the Project

As a result of loading the "xor.proj" file, we created a number of objects, all of which can be thought of as organized into a hierarchical tree, with Root at the top of the tree. The Root window is associated with the Root object, which is a generic object that basically just serves as a place holder for other objects. We opened the XOR project in Root, and so an object representing the entire XOR project has been created as a child of Root. The **Project** window is associated with this project; in fact we could create a second project under root, and if we did a second project window would be created. Typically, though, we just have one project. Within this project, we have both a network and an environment, as well as some other things we will consider later in this tutorial. Within the network we have layers of units; within the units we have projections from other layers, which contain connections.

Each object can be identified by a string that specifies its location in the object hierarchy. Root sits at the top of the object hierarchy; our project is a 'child' of root. It's identifier string is `.projects[0]`. In general, the identifier string for an object specifies its location in the object hierarchy as a pathway specification that begins with a '.', followed by fields separated by dots. Each field contains a type-name and an index. One can trace the pathway to a particular object down from Root (implicit in the first dot) through intermediate fields separated by dots, ending with the field specifying the type and token number of the object itself. The path for the project is short since it is the first (and only) project that is a child of Root. This is all exactly like a directory hierarchy, except we have objects instead of files and the separator is '.' instead of '/'.

Objects can have proper names as well as type names and indices; these names can be used to access the object directly and to give it a mnemonic label that can be used in referring to it. The windows you see give the full identifier string and if a name has been assigned, they give the name as well in parentheses.

Within the Project window, we have several menus accessible. As before, one of these menus refers to the Project Object itself (the *Object* menu). The other menus refer to the children of the project, which are again organized into several types. The main ones we will

care about are the networks, the environments, the processes, and the logs. The defaults, specs, and scripts are for customizations that we will not consider for the moment. We could access the network through the *.networks* menu, but we already have a window up on the screen that gives us access to the network, called the **NetView**.

4.2.4 Network and NetView Window

The **NetView** window (see Section 10.6 [net-view], page 149) is an essential window for us since it provides us with access to the network that we have created for learning the XOR problem. The **NetView** window has two sets of menus along with lots of pushbuttons and a display area containing a graphical display of the network. Lets take a look at the menus first. The left hand set of menus relate to the **Network** itself. The right hand set of menus relate to the **NetView**. The **NetView** is an object, actually a child of the **Network**, that provides a view of the Network. We can operate on either one, and we have two separate sets of menus.

The left hand menus allow access to the **Network** itself and the Networks' children: its *.views* and its *.layers*. Most of these menus have the same basic structure as we have already seen, but there is one new menu, namely the *Actions* menu. This menu allows us to perform actions on the network itself, such as removing things from it, initializing it, etc. Some of the other actions we can perform are accessible through the buttons to which we will turn in the next paragraph. The right hand menus allow us to access the **NetView**. This is indicated by the label *View:* on this set of menus. These menus operate on the visual display of the network, or any operation that interacts with the visual display (e.g., the *Selections* menu operates on things that are selected in the visual display).

The **NetView** itself, the main body of the NetView window, provides a graphical depiction of our network. You can think of the layers of the network arranged vertically, with the units within each layer laid out on a two-dimensional sheet. The input layer is located at the bottom of the netview, with the two input units arranged side by side. In networks with many units per layer the units can be laid out in an X by Y array. The hidden layer is next above the input layer, and we can see the two hidden units, again laid out side by side. The output layer, above the hidden layer, contains just a single unit, aligned to the left of the netview. At present, each unit is displaying its activation, as indicated by the depressed appearance of the `act` button, together with the yellow highlight. The activation is shown in color in accordance with the color bar shown at the right of the NetView, and by the numeric value printed on it. Either way we see that the activation of all of the units is 0.

The **Network** has been initialized so that each unit has a random bias weight and so that there are random weights on the connections between the units. We can examine these biases and weights by displaying them on the netview. First, to look at the bias weights, we can click on the *bias.wt* button. Once we do this, the units will assume colors corresponding to their bias weights. The biases on the input units are not used and can be ignored.

We can also view weights either coming into or going out of a selected unit. The former are called 'receiving weights'. To view the receiving weights of the left hidden unit we first click on the button labeled *r.wt*. This activates the *View* button, and turns the cursor into a little hand, indicating that we may now pick a unit to view its receiving weights. We can now click on the left hidden unit, and its receiving or incoming weights are displayed on

the units from which they originate; the selected receiving unit is highlighted. We see that this unit receives incoming weights only from the two input units, both somewhat positive. We can proceed to click on other units to view their incoming weights as well. Note that the input units have no incoming weights. To see the outgoing or 'sending' weights from a unit, we simply click the *s.wt* button, then click the unit whose sending weights we wish to view. You can verify that the values of the sending weight from the left input unit to the right hidden unit is the same value as the receiving weight to the right hidden unit.

Since all we've done so far is initialize the network's weights there isn't that much else we can look at at this point. However, if we click on the *act* button, the network will be set to display the activations of units later when we actually start to run the network. Do this now.

4.2.5 The Environment

A second type of object contained in our Project is the **Environment**. The environment can be thought of as a special object that supplies events (training examples) on request. In real life, if our network were embedded in the brain of some organism the training experiences would arise from the world. In the simulator we will imagine that we sample events from the environment when a process requests them. In general the environment can itself be an arbitrarily complex process, but in the present case, the environment simply contains a list of 'events', each of which in turn consists of an input-output pattern pair. If we double click on the iconified **EnviroView** window, we can inspect the objects corresponding to each of these Events. The window that pops up displays the names of all four of the events that make up the 'XOR' problem. You can inspect any one of these events by clicking on the appropriate button, and if you do you will see that it consists of two patterns, an input and an output pattern. You can display all four together, by first clicking the left mouse button on the first event button, then clicking the middle mouse button on the other event buttons in order. The color scale indicates the correct interpretation of the colors that indicate the activations of the units. Once you check this out you can go ahead and iconify the **EnviroView** window again.

4.2.6 Processes

Since everything has been properly initialized, and the various displays are all laid out for our use, we are ready to try to teach the network the XOR function. We train networks, and test them, by running **Process** objects that operate on them.

Process objects are described at length in Chapter 12 [proc], page 184. We will only say enough about them here to give you a general sense of what they are and what they do. You can think of processes as objects that consist of several parts: an initialization part, a loop part, and a finishing-up part. When a process is run, it first carries out its initialization process; then it loops for a specified number of iterations or until some performance criterion is met, and then it finishes up and exits. For many processes, the loop consists of simply calling the process's sub-process over and over again. At the bottom of the process tree, we call routines that actually interact with the network. Processes also spawn sub-processes that calculate statistics, such as the sum of squares statistic that measures how well the network has solved the learning problem that has been posed to it. Some statistics, called

`loop_stats`, are calculated at the end of each step in the process loop; others are calculated only when finishing up.

In the case of backpropagation, we have a three-level process hierarchy. At the bottom of the hierarchy is the **TrialProcess**. This process takes an event, consisting of an input pattern and a target pattern, from the **Environment**, and then carries out the three phase process described previously (see Section 4.1 [tut-bp], page 20). The processing is actually implemented by looping through the **Layer**s in ascending numerical order; then looping, within each layer, through the **Unit**s, and calling functions associated with each unit to compute the quantities as previously described. All of the variables mentioned there are explicitly stored in each unit, and can be inspected once they have been computed by selecting the appropriate button in the **NetView**. At the end of the trial, the sum of squares statistic is calculated; in this case it isn't much of a sum since there is just one output unit.

The Trial process is actually called as a sub-process of the **EpochProcess**. All the Epoch process does is loop through the set of pattern pairs in the **Environment**, calling the trial process to process each pattern pair. Before it starts, it initializes its statistic, the sum of the trial-wise sum-squared-error (`sum_sum_SE_Stat`), to 0. As it loops, it computes the `sum_sum_SE_Stat` by aggregating the `sum_SE_Stat` statistic over each of the training trails, so that at the end of the epoch, it reflects the sum over all of the trials in the epoch. At the end of the epoch, it passes the `sum_sum_SE_Stat` up to the training process.

The Epoch process is called by the **TrainProcess**. This process initializes the epoch counter before it starts. It stops when the counter reaches a specified maximum value, or when the latest (or last) sum-squared-error value (`lst_sum_SE_Stat`) falls below its stopping criterion. The stopping criterion is actually a property of the Train process's **SE_Stat** statistic, `lst_sum_SE_Stat`. This statistic simply copies the last `sum_sum_SE_Stat` from the epoch process, and compares it to the criterion, passing a special return value to the Train process when the criterion is met. If so, the process exits from the loop at that point, and training is stopped.

4.2.7 Logging Training and Testing Data

Before we start to train our **Network**, and see if it can learn the XOR problem, we will need a couple of tools for visualizing the network's performance as it learns. To allow for this, we have created two objects called **Log**s, one for recording the state of various network statistics and variables as they are computed during the training process, and one for recording the activations of the hidden and output units, when the network is tested with the learning process switched off. The data recorded in these logs are displayed in the **GraphlogView** and the **GridLogView** respectively. Double click on both of these to open them up.

The **GraphLogView** is set to display the sum of the sum-squared-error measure (`sum_sum_se`), summing over all output units and all of the training patterns presented within each epoch. The Epoch number will be the X axis and the summed squared error will be the Y axis. When we start to run the network this will be updated after every epoch as we shall see.

The **GridLogView** is set to display detailed information about each pattern whenever a special test process is run. During the test, each event in the environment is presented once, and the View displays the epoch number, the Event name, the sum_squared error over all output units (though in this case there is only one), and the activations of the hidden and output units that occur when the input pattern associated with this event is presented.

4.2.8 Running the Processes

To run a process, we first create a 'control panel' window for it. Select *.processes / ControlPanel / Train* in the **Project** window; this creates a control panel for running the **TrainProcess**. At this point, if you were to click on *Run* (don't do it now!), the network will proceed to run epochs, until either the stopping criterion is met or `max` epochs (shown in the `max` typein box) are completed.

Rather than start training right away, let's run through a test before any training is carried out. We can do this using the "Test" process, which is actually just another **EpochProcess** that is set up to run one epoch without learning. Select *.processes / ControlPanel / Test* in the **Project** window, then simply press *Step*. Now, you will see some action. First, assuming that *act* is selected in the **NetView**, you will see the activations of all of the units as the first pattern is processed. The input units are both off (0) in the first pattern, and should have the color associated with 0 in the color scale. The hidden and output units all have activations between about 0.4 and 0.6. The same information is displayed in the **GridLogView**. You should see the epoch number, the Event name, the sum-squared error for this event, and a display showing the activation of the hidden and output units for this event. If you press *Step* 3 more times you can step through the remaining test patterns. At the end of stepping you should have the results from all four cases visible in the GridLogView.

Now let's run 30 epochs of training and see what happens. To do this, just click on *Step* in the "Train" control panel, which is pre-set to run 30 epochs in each step (as indicated by the value of `n` in the control panel `step` field). When you do this, assuming that indeed the *act* button has been clicked in the **NetView**, you will see the state of activation in the network flicker through all four patterns thirty times. At the end of each epoch the total sum of squares will be displayed in the **GraphLogView** — the results are pretty dismal at first, with a sum of sum-of-squares-error (`sum_sum_se`) oscillating a bit very close to 1.

After 30 epochs you can run through another test if you want, either stepping through the patterns one at a time as before with the *Step* button or running a complete test of all of the items with the *Run* button. If you hit *Run* the results will flicker by rather fast in the **NetView** but again all four cases will be logged in the **GridLogView**. You can see that the activations of the hidden and output units are very similar across all four cases.

You can proceed to taking successive steps of 30 epochs, then testing, if you like, or, if you prefer, you can simply let the learning process run to criterion, and then run a test only at that point. To do the latter, just hit *Run* in the "Train" control panel. You can click off the *Display* toggle in the **NetView** (near the upper left) — this will speed things up a bit, even though the **GraphLogView** will still be adding new data after each epoch.

As with the example of learning XOR from the original PDP handbook, not much happens to the sum of sum-squared-error (`sum_sum_se`) until about epoch 160 or so. Now the error begins to drop, slowly at first, then faster... until at about 270 epochs the stopping

criterion is reached, and the problem is solved. Hit *Step* in the "Test" control panel, to step through a test at this point. You should be able to see either in the **NetView** or in the **GridLogView** that the units that should be off are close to off (less than about .1) and the units that should be on are most of the way on (activation about .9).

If you would like to let the network learn to get the activations even closer to the correct values, you'll need to change the stopping criterion. This criterion is associated with the sum of sum-squared-error statistic (`lst_sum_SE_Stat`) in the **TrainProcess**. To access it, select *.processes / Edit / Train / lst_sum_Train_SE_Stat* from the .project window. The label "lst_sum_Train_SE_Stat" indicates that this **SE_Stat** (squared-error statistic) is associated with the **TrainProcess**. It originates at the trial level, but its `sum` over training trials within the epoch is passed to the epoch level, and the `lst` (last, latest) value at the epoch level is passed to the Train process.

An edit dialog box will come up when you *Edit* it. This may seem a bit daunting at this point — we're letting you see how much there is under the hood here — but bear with us for a moment. All you need to do is find the row of this dialog labeled `se`. You'll see its current value, a toggle indicating that it is set to serve as a stopping criterion, a criterion-type (here less-than-or-equal or `<=`), and to the right of this a numeric criterion value (which should be 0.04). Click in this numeric field, and set it to 0.01.

Before the new value takes effect, it must be propagated from the dialog box to the actual internal state of the simulator. As soon as you have started to make change, the dialog will highlight the *Apply* button to indicate that you need to press this button to apply any changes that have been made to the state of the simulator. Once you have the value field showing the new value that you want to use, click *Apply* and the change will be implemented. Then click *Ok* to dismiss the window Note that *Ok* also performs an "Apply" if necessary, so you can actually just do this in one step. However, often you'll want to keep a dialog around after you make changes, which is where the *Apply* button is useful.

Now hit *Run* again in the "Train" control panel, and the network will run around 70 more epochs to get its performance to the new, more stringent criterion.

4.2.9 Monitoring Network Variables

It is often useful to monitor the values of network variables, such as the activation of the output units, in your log files. For example, you may wish to watch how the network gradually differentiates its output in response to the four input patterns. We can do this by creating what is called a **MonitorStat** statistic.

Conceptually, a monitor statistic is a special type of statistic that simply monitors values already computed by the processes that are applied to the network. For example, activations are calculated by the **TrialProcess** during training. To record these activations in a log we need to monitor them and that's what a Monitor Statistic will do. Think of it as an electrode for recording data at specified times inside your network.

To create a Monitor statistic, first select the units and the variable you want to monitor in the **NetView**. To monitor with activation of the output unit, for example, we would make sure `act` is selected as the variable, and then we would select the output unit. In this case we can select either the unit or the whole layer, it won't make any difference. To select the layer, make sure the *Select* button is highlighted in the NetView and then click over the

output layer itself. The layer's border will be highlighted. Now Select *Monitor Values / New* (located in the middle-left region of the netview), and a pop-up will appear with some necessary information displayed.

The popup indicates that you are creating a type of Statistic called a **MonitorStat**, and it indicates which process will contain this Statistic. In this case, the `In Process` button will indicate that the statistic will be placed at the end of the "TrainTrial" process, which is what we want (we already have something monitoring activations in the "TestTrial" process). The `For Variable` should be `act`. The `Using Operator` in this case is `COPY`, which is also what we want – we just want to copy the values of all the selected units into the statistic. The last aspect of this is conceptually the most challenging. What we need to do is to specify that we want to `Create Aggregates` of this variable at higher levels. In this case, what we really want to do is just copy the activation from all 4 patterns to the Epoch level, so we can display the activations of the output unit for the different events in our Graph. So click the *CreateAggregates* toggle, then click *Ok*. Another popup will then appear, asking you what form of aggregation you want. In this case you must select *COPY*, to indicate that we are copying each event's output unit activation into the epoch graph, not averaging or summing or anything like that. Do this, click *Ok*, and you are done.

Before you restart the simulation, you'll need to reinitialize the weights. We do this in the Train Control Panel. You reinitialize with the same starting weights by clicking *Re Init*, or you can initialize with new starting weights by clicking *New Init*. Either way, the network is reinitialized and the epoch counter is set to 0. Click *Re Init* this time so we can see trace the learning process over the very same trajectory as before. To run the simulation again we can simply press *Run* or *Step* in the "Train" process control panel. You'll see the new results graphed on the same graph along with the earlier sum of sum-squared-error results.

The graph may seem a little confusing at first. Overlaying the `sum_sum_se` results will be four new lines, which will seem to be oscillating rather extremely. Buttons identifying these new lines will be shown on the left of the View. Note that the color of the graph line associated with a variable that is displayed is shown to the left of the Button, and the color of the axis that is associated with the variable is displayed on the right. All four new lines are associated with a new Y-axis, which you can see covers a rather narrow range of values (about .46 to .54). This axis is tied to the entire group of activations that are being monitored, and it is auto-scaled to the largest and smallest value in the display.

You may want to fix the scale to something reasonable like 0-1. To do this, click with the *right* mouse button — the *Edit* button – on the lead (top) element of the set of new elements (it may be labeled `output.act` or `cpy_output.act_0`. A popup will appear. You want to set `min mode` and `max mode` to `FIXED` so click where it says `AUTO GROUP` and select `FIXED`, then enter 1.0 in the max field next to `range`. To make this take effect, click *Update View*, then click *Ok*. The new Y axis will now span the 0-1 range, and the oscillations will appear more moderate on this scale. This axis adjustment can be done while the simulation is running.

In any case, you'll see that after thirty epochs or so the oscillations level off. The simulation appears to be flatlined until about epoch 120, when the activations of the output unit begin to differentiate the four input patterns. (The outputs to patterns 1 and 2, which are the 01 and 10 patterns, are nearly identical and the latter lies on top of the former much of the time). In any case, you should see that the network tends to produce the strongest

output for the '11' pattern (pattern 3) until about epoch 210, well after it has completely nailed the '00' case. After epoch 210 the performance changes rapidly for the 10, 01, and 11 patterns (patterns 1, 2, and 3) until about epoch 270 or so, where they begin to level off at their correct values.

4.2.10 Changing Things

Our network has succeeded in learning XOR, but none too quickly. The question arises, can backpropagation do a better job? The first thing that might occur to you to try is increasing the learning rate, so let's try that.

Before you can change the learning rate, you must understand how such variables are handled in the simulator. The simulator is set up so that it is possible to have different learning rates for different weights. This would, for example, allow us to have one learning rate for the weights from the input layer to the hidden layer, and a different learning rate for the weights from the hidden layer to the output layer. To allow this, parameters such as the learning rate are stored in structures called *specifications* or **Spec**s. Each bias weight and each group of receiving weights (i.e., all the weights coming to a particular unit from a particular layer) has associated with it a pointer to a specification, and parameters such as the learning rate are read from the specification when they are used. Our network has been set up so that all of the weights and biases share the same specification. This specification is called a connection specification or **ConSpec**, and since it is the one we are using in this XOR network we have called it "XORConSpec".

One can look at this specification by selecting in the **Project** window *.specs / Edit / XORConSpec*. When this pops up we can see that there are actually several variables associated with this Spec, including `lrate`, `momentum`, and `decay`. We can change any one of these, but for now click in the `lrate` field and change the value to, say, 1.0. Then click *Apply* or *Ok* (which applies and closes the pop-up).

To see what happens with this higher learning rate, go ahead and *Re Init* on the Train Control panel as before. Before you hit *Run* you may wish to *Clear* the **GraphLogView**.

One can modify the learning process in other ways. For example, one can change the `momentum` parameter, or introduce `decay`. These parameters are also associated with the **ConSpec** and can be modified in the same way as the `lrate`. For more information about these parameters, see Section 14.1.2 [bp-con], page 227.

Another parameter of the Backpropagation model is the range of values allowed when initializing the connection weights. This is controlled by the `var` parameter of the `rnd` field of the **ConSpec**. The default value of 0.5 can be increased to larger values, such as 1.0, thereby giving a larger range of variation in the initial values of the weights (and biases).

One can also modify the learning process by selecting different options in the **EpochProcess**. For example, as initially configured, the XOR problem is run using `SEQUENTIAL` order of pattern presentation, and the mode of weight updating is `BATCH`. `SEQUENTIAL` order simply means that the patterns are presented in the order listed in the environment. The other two options are `PERMUTED` (each pattern presented once per epoch, but the order is randomly rearranged) and `RANDOM` (on each trial one of the patterns is selected at random from the set of patterns, so that patterns only occur once per epoch on the average).

The options for weight updating are `BATCH`, which means that the weights are updated at the end of the epoch; `ON LINE`, which means that the weights are updated after the presentation of each pattern, `SMALL BATCH`, which means that the weights are updated after every `batch_n` patterns, and `TEST`, which means that the weights are not updated at all.

These characteristics of the processing are managed by the "Epoch_0" process that is subordinate to the "Train_0" process— and so to change them select *.processes / Edit / Epoch_0* from the **Project** window menu bar. The dialog that pops up indicates the current values of these options (sequential and batch) next to their labels (`order` and `wt update`). Click on the current value of the option and the alternatives will be displayed; then select the one you want, and click *Apply* or *Ok* when done.

4.2.11 Saving, restoring, and exiting.

You can save the state of your simulation or virtually any part of it. This is done by selecting *Object / Save* or *Object / Save As* on the appropriate menu. For example, to save the whole project using its existing project name, you would select *Object / Save* from the **Project** window. To save just the network in the file "myxor.net" you would select *Object / Save As* from the **NetView** window and then enter the filename "myxor" in the filename field of the popup before clicking *Ok*. The program will actually save the file as "myxor.net.gz"; the ".gz" suffix indicates that the file has been compressed using '`gzip`'.

To restore the state of the simulation or some part of it, you can just select *Object / Load* in the appropriate window. This will load the saved copy of the object over whatever is presently there, effectively destroying whatever was there before. When you first start up the simulator, you can load a project by selecting *.project / Open In / Root*. This creates a project as a constituent of root and loads the project saved in the file.

At the end of the day, once you've done whatever saving you want to do, you can exit from the simulator by selecting *Object / Quit* from the Root window.

Now you know how to run the software and you are done with this part of the tutorial. The next section of the tutorial steps through the process of actually building a simulation from scratch instead of loading a "canned" one in.

4.3 Configuring the Encoder Problem

Running a canned exercise may be fun and informative, but most people like to use simulation models to work on something new of their own. In this short section we will give you a quick overview of the essentials of creating your own project. We'll use the 4-2-4 encoder problem, first considered by Ackley, Hinton and Sejnowski in their seminal article on learning in Boltzmann machines. We'll stay with the back propagation learning algorithm, though, since it learns the problem much faster than the Boltzmann machine.

In the 4-2-4 encoder problem, the goal is to learn to associate an input pattern with itself via a small hidden layer (in this case containing 2 units). There are 4 input units and 4 output units, and there are 4 training pattern-pairs, each involving one unit on, the same unit, in both the input and the output:

```
Input   Output
1000 -> 1000
```

```
0100 -> 0100
0010 -> 0010
0001 -> 0001
```

What we need to do is set up a network with four input units, four output units, and two hidden units; then create an environment containing the set of training patterns; then create the processes that train and test the network, then create Logs and LogViews for displaying the results of training and testing.

To begin, you just start up the executable for bp by typing `bp++` at your unix prompt. This starts the program, giving you the `bp++` prompt as your command-line interface to the program, and giving you a Root window as the seed of the GUI you are about to build for yourself. You're going to do the whole thing through the GUI. You can keep a script of what we've done so you can repeat the same steps (or even edit them and do something slightly different) later. This is optional, however; many people prefer simply to save the state of their configured project in a .proj file.

To create your project, select *.projects / New / Project* in the Root window, and then just click *Ok* on the confirmation box that appears. A project menu will appear.

4.3.1 Recording a Script File of Your Actions

If you would like to make a script file of your actions, you should start it up before you start building the network. The script will record all your actions, which means you can later look at this script and see what steps you took (and replay them). However, because one inevitably makes mistakes along the way, using a script to save everything is not the best idea — instead, we simply save a project file at the end with everything stored as it is currently configured.

To start recording your script, select *.scripts / New / Script* , and *Ok* the creation with the *right* mouse button (this tells the program you want to edit the object that you create). An edit dialog window for the script object should automatically appear. If you forgot to use the right mouse button, you can manually bring up an edit dialog by choosing *.scripts / Edit / Script_0*. You will want to associate a file with this script, so click on the menu next to `script file` (where it says `--- No File ---`) and select *Open*. A File Selection Window will appear with an empty edit field near at the top under the text "Enter Filename:". Fill in the name you want to give your script ("my424" would do) and press the "Open" button at the bottom of the File Selection Window or press <Enter>. Once this is done you still have to click *Apply* back on the script dialog to confirm your file name. Now hit *Record* and your future actions will be recorded in the script. The mouse pointer should change to an arrrow with the letters "REC" beneath it to indicate that you are in record mode. To stop recording you can press the *StopRec* button in this window. Since we do not need this window until later you can iconify it now.

4.3.2 Making a New Network

Now, lets make the **Network**. Select *.networks / New / Network* and *Ok*. You now have a blank **Network** object, and a **NetView** window in which to view it.

Click *New Layer(s)*, set the `Number` to 3 (by editing the number field or using the increment/decrement buttons), and hit *Ok*. Your three layers will appear, but at this point

none of them contain any units. The **NetView** is now in *ReShape* mode (the corresponding mode button is highlighted). So, if you select "Layer_0", you can start to reshape it. Click and hold in the line you see above the layer label (this line is a layer-frame with no space in it yet) and drag the mouse to the right; boxes will appear corresponding to unit-spaces. When you have 4, stop and let go. You've created a frame for four units. Click on the line above "Layer_1" to create a frame for the two hidden units; drag right until two boxes appear. Finally click on the line above the "Layer_2" label, and create a frame for the four output units. You now have frames but no units; to create the units, in all three layers at once, just click on *Build All*. The units should all now be visible, and the default *act* variable selected, indicating that your are viewing their activation states, which should all be zero.

You can move the hidden layer to be centered, by selecting *Move* mode then dragging the hidden layer with the mouse. Note that things move in "chunks" of the size of a single unit, so you will have to move the mouse a certain amount before the layer itself moves. It's a good idea to go back to Select mode after moving by hitting *Select*.

Also, clicking the *Init* button will adjust the display of the network to fill the available space, and generally fix any display problems. It is automatically performed most of the time, but you can use the *Init* button if the display doesn't look right for some reason.

Now we need to make connections. Actually, you first make **Projection**s then create connections within the projections. A projection is to a connection what a layer is to a unit; it represents a bunch of connections that connect units in two layers. In any case, you need to create two projections. One to the hidden layer from the input layer, and one to the output layer from the hidden layer. In each case:

- Select the receiving layer with the left mouse button. You want to see the frame around all of the units highlighted. This should occur on the first click; if that's not what you see, keep clicking until it is; (you're cycling the selection through the layer, all the units, and the unit under the mouse).
- Add the sending layer to the selection using the "extend-select", which can be done with either the middle mouse button, or the Shift key plus the left mouse button, depending on whether you come from a Unix or a Mac background. Now both the sending and receiving layers should be highlighted. If you make a mistake, click the left mouse button in the background area of the NetView between the layers to unselect the layers and start over with the previous step.
- When you have 2 layers selected, click *New Prjn(s)*, which will be highlighted. A Projection arrow should appear between the two layers. Note that the arrow head of the projection arrow is unfilled (outline) indicating the the actual connections have not been created yet.

After you do this for both projections you will have two unfilled arrows. Now click *Connect All* and your projections will be filled with connections to each unit on the receiving end from each unit on the sending end. Now the arrow heads of the projections will be solid indicating that their connections have indeed been created. You may verify this by clicking on the *r.wt* button and using the finger pointer to view the weights of some of the hidden and output units. When you are finished looking around, return to viewing the activations by pressing *act*.

And your network is complete.

4.3.3 Making a New Environment

Now we can create the **Environment** and the training patterns. Locate the Project window (which you'll note has been updated to reflect the existence of your new Network object), and select *.environments / New / Environment* and *Ok*. Your **EnviroView** window will appear. Then click *New Event*. Set the number to create to 4, then click *Ok*. You should see Panel buttons for all four events ("Event_0".."Event_3"). Left-click the first one to select it for viewing, then extend-select (middle-click or Shift + left-click) the rest in order, and all of them will appear to the right of the buttons. They automatically have an input pattern the same size as the input layer and an output or target pattern the same size as the output layer, and all the values of all of the patterns are initialized to 0. Both input and output patterns are displayed with the top four boxes of each event represent the output target values and the lower four boxes representing the input values. You can now set the 1 bits of each pattern by clicking on the color spot next to the value field labeled '1', then clicking on the appropriate elements of the pattern. They will turn to the appropriate color, indicating the specified value. You have to click *Apply* for the change to take effect. Now you have your environment.

You can also use this **EnviroView** window to configure the layout of the patterns within the events according to the **EventSpec** and **PatternSpec** specs. The default layout (which works for our present purposes) is to have one pattern for the first layer in the network, which serves as an input, and another pattern for the last layer in the network, which serves as an output. However, these defaults will not always be appropriate. To see how to change them, hit the *Edit Specs* button at the top left of the window, and then after the display changes, hit the *EventSpec_0* button. You will see two grids for the two patterns (input and output). You can move, reshape, and edit these patterns if you need to. The text within each pattern shows some of the critical settings in terms of whether an pattern is an `INPUT` or `TARGET`, what layer it goes to, etc. We'll just leave everything as-is for now, but if for example you wanted to change the network to be a 8-3-8 encoder, then you'd need to reshape these patterns to be 8 units wide instead of 4. Doing so would automatically stretch the corresponding events that use this spec.

For now, just hit *Edit Events* to return to the events display, and iconify the window.

4.3.4 Setting Up Control Processes

While the **Network** and the **Environment** constitute the bulk of the custom aspects of a simulation, the way in which the network is trained can vary depending on what kinds of tasks you are performing, etc. The training and testing of a network is determined by the processes, and the results of running these processes can be viewed in logs that record data over time.

Processes come in hierarchies, but there are sensible defaults built into the program that create sensible hierarchies. We will create two hierarchies, one for training the network, and one for testing. Should your process hierarchy become accidentally misaligned with unnecessary Train, Epoch and Testing processes or spurious Batch or Sequence processes, you may wish to start the process creation procedure over from scratch by choosing *.pro-*

cesses / Remove / All. In this case, the default naming of the processes may use a different numbering convention than the one described below, but hopefully you will be able to follow along.

To create the **TrainProcess**, select *.processes / New / TrainProcess* from the Project window. A New Object Popup window will appear indicating that 1 Train Process should be created. In addition, the *Create Subprocs* toggle will be checked; leave this, it will do the right thing and create a subordinate Epoch and Trial process under the Train process for you automatically. Use the right mouse button to click *Ok* so you can edit the Train process. If you forget to use the right mouse button, you can edit the train process by selecting *.processes / Edit / Train_0.* You could at this point rename the train process to something more descriptive, (e.g., "Training") by changing its name field, but we'll assume you stay with the default, which is just Train_N, where N is the index of this instance of a train process; if this is the first process you are creating, the process will be called Train_0. One thing you can notice here is the max epoch value associated with the Train process. By default it has value 1000, which is reasonable for many relatively small scale problems — if it doesn't learn in 1000 epochs, it may not learn at all. We are done with editing the Train Process at this point, so you should press *Ok* and dismiss the Edit Dialog.

Now, take a look at your Project window, which shows all of the major objects within your project. You should see the network and environment, and the three processes in the process hierarchy you just created. You can automatically view or iconify the network or environment windows by double-clicking on their icons. Also, you can see how everything is connected up through the processes by clicking on a process and hitting show links — select the yellow *Trial_0* icon and then hit the *Show Links* button. You should see that this process works on the network (solid pink line) and the environment (solid green line), and that it updates the network display (dashed pink line). You can also see the statistics that are computed at each level of processing (more on this below).

We can now edit the sub-process Epoch_0, either clicking with the right mouse button on the *Epoch_0* object in the project view, or by choosing its name from the *.processes / Edit / name* menu. Again, you could rename them at this point; but we'll assume you stay with the default name of Epoch_0. The main reason to edit the Epoch process is to specify the presentation order and what mode of weight updating you want by setting the values of the *order* and *wt update* fields in the edit dialog for Epoch_0. The defaults, `PERMUTED` and `ONLINE` may not be what you want (`SEQUENTIAL` `BATCH` is the "standard" combination).

By default, the TrialProcess for the Bp algorithm creates a squared-error statistic. You can see this statistic in the project view, or by pulling down the edit menu, and moving the mouse over any of the processes — it shows up in a sub-menu off of the these processes. In general, statistics are created at the **TrialProcess** level, because that is the appropriate time-grain for the computation of most statistics. For the squared-error, this is when an individual pattern has been presented, the activations propagated through the network, and a difference between the target and actual output values is available for computing the error. However, one often wants to see an *aggregation* of the error (and other statistic) over higher levels of processing (e.g., a sum of the squared-error over an entire epoch of training). This is done by creating aggregates of the statistic, and it is why all of the processes contain a squared-error statistic in them – the ones in the Epoch and Train processes are aggregations of the one computed in the TrialProcess.

While you could procede to the next step now, we will make a small detour in order to show you how to create statistics, since you may in the future want to create a different type of statistic than the default squared-error statistic. We will simply remove the existing statistic, and then perform the steps necessary to recreate it. To remove the statistic, you can click on it (under the *Trial_0* process) and hit *Rmv Obj(s)* button in the project view, or select *.processes / Remove / Trial_0 / sum_Trial_0_SE_Stat*, and confirm that it is *Ok* to close (remove) this item. Note that the SE_Stat has been removed from all of the processes — a statistic will always remove its aggregators in higher-level processes when it is removed.

Now we will perform the steps necessary to re-create the statistic we just removed. You can do this by either selecting the *Trial_0* object in the project view, and hitting *New Stat*, or by selecting *.processes / New Stat / SE_Stat* in the **Project** window (*New Stat* is near the bottom of the menu). The popup that appears indicates the type and number; it also indicates which process the stat will be created in. As mentioned above, statistics are generally created at the **TrialProcess** level, because that is the appropriate time-grain for the computation of most statistics. Given that there is only one Trial process (Trial_0) at this point, the program will guess correctly and suggest to create the SE_Stat at this level. The popup also allows you to choose where within the Trial process to create the statistic – with the two options being Loop and Final. Most statistics will know where to create themselves, so the *DEFAULT* selection should be used, which means it is up to the stat's own default where to go. You'll want to create aggregates (summing over trials at the epoch level, and keeping track of the latest value at the overall Train process level), so leave the *Create Aggregates* toggle checked. Click *Ok*.

Now you get to answer another question: What aggregation operator do you want for the next (epoch) level? The default `LAST` (keep track of just the last value) is not what we need; you want to `SUM` this stat at the next level, so select `SUM`, and then click *Ok*.

Note that the stat is also being aggregated at the levels above the epoch (i.e. the **Train** level). It will keep track of the last value at the **Train** level. Note that at each level each statistic has a name refelcting the process and the nature of the aggregation. So the Train level stat is called "lst_sum_Train_0_SE_Stat" indicating it is the last value of the sum of an **SE_Stat** and that it is attached to the Train level. We have now re-created the squared-error statistic. You could have skipped over this process, but now you know how to do this in the future.

Finally, you need to set the stopping criterion for learning, which is not set by default. You can do this by editing the "Train_0_SE_Stat". You find it by right-clicking (or selecting and hitting *Edit*) the object in the project view, or by doing *.processes / Edit / Train_0 / lst_sum_Train_0_SE_Stat*. Once this pops up you will find the `se` value of this stat near the bottom of the window. This field has 1 parameter `val` which indicates the current value of the statistic, and 4 parameters which control when the stopping actually occurs. To set the criteria you need to click on the first stopping paramater, `stopcrit` which should turn on with a checkmark symbol indicating that we are indeed using this statistic as a stopping criteria. The default relation `<=` on the next parameter to the right is appropriate here so ther is no need to adjust its value. To the right of the relation parameter is the value to which the statistic is measured against. In our case, the `val` of this SE statistic is compared with this value to determine if it is less than or equal to it. A reasonable value for this parameter is .04 (.01 for each pattern you are testing, summed over patterns). The next

field labeled `cnt` can be used to set a stricter stopping criterion where the condition must be met `cnt` times in a row instead of just once. We will just leave the value at 1. Click *Ok* when you are done.

You will probably also want to create a Test process hierarchy. A test sweeping through all of your patterns is an **EpochProcess**. So, select *.processes / New / EpochProcess*. After you click *OK* (use right mouse button to edit) in the popup window, go ahead and edit this process (if you didn't use right-click, select *.processes / Edit / Epoch_1*). Set `wt_update` to `TEST` and `order` to `SEQUENTIAL`. Click *Ok*. Note that a default SE_Stat was automatically created for your test process. There is no need to set any stopping criteria.

Now, you probably want to monitor some aspect of network function in your test. Let's look at the activation of the output units. To do this:

- Select the units in the output layer in the **NetView** (either the whole layer, or all the units in the layer).
- Select *act* to display on the **NetView**.
- Click *Monitor Values / New* (located in the left-middle of the **NetView** display).
- In the pop-up, set the process to "Trial_1". You'll find it under *BpTrial* when you click on the `In Process` value field. Everything else is as we want it so click *Ok*.

Also, it is useful to create an EpochCounterStat for the test process by selecting *.processes / NewStat / EpochCounterStat*, then, when the pop-up appears, for *In Process*, select *Trial_1*, then click *OK* (or select *loop_stats* under the *Trial_1* object in the project view, and then pick *EpochCounterStat* from the dialog). When the program asks you how to aggregate this statistic, it will remember the value you entered last time (`SUM`), but we will want to use `LAST` this time. This will allow you to record information about how many epochs the network has been trained on each time you run a test.

Now, so you'll be able to run the net and test it, you can create control panels for training and testing. In the project view, select the *Train_0* object and hit *Ctrl Panel*, and then the same for *Epoch_1*, or do *.processes / Control Panel / Train_0* for the training panel and ... */ Epoch_1* for the testing panel. In the Train_0 control panel, you might set the `step n` field to something like 30, so that when you click step it will run for 30 epochs.

4.3.5 Creating Logs For Viewing Data

Before you run the network, you need to create logs of the results of training and testing and views that allow you to examine the contents of these logs. We'll create a Log with a **GraphLogView** to follow the **SE_Stat** over epochs and Log with a **GridLogView** to follow the activations of the output units at test.

To create the **GraphLogView**, select *.logs / New / GraphLog*, and hit *Ok* in the New dialog. It will automatically prompt you for which process you want to update this view, which should be *Epoch_0*. This will cause the **GraphLogView** to get information from this process about what is being monitored there: the **SE_Stat**, summed across output units and across patterns in the epoch. By default the epoch number will be used at the X axis, so this GraphLogView is now ready.

To create the **GridLogView**, select *.logs / New / GridLog*, and specify that *Trial_1* should update this log. Again, the header for the data that are being monitored is automatically retrieved.

As an advanced example of setting up a grid log, we now describe how you may be able to set up a special GridLog associated with your Test process (Epoch_1) that displays the weights in your network at the end of each test that you run. This may or may not be something you really need to do depending on your goals. But the example shows some advanced features that are useful and powerful, so we go through it to expose you to them.

1a. Create a monitor stat for each projection. Make sure *r.wt* is the current variable displayed, and that you are in `Select` mode instead of `View` mode. Now select the projection in the netview which connects the hidden and output layers. Then click *Monitor Values / new*. For *In Process*, select *EpochProcess / Epoch_1*, and for *Loop/Final* select *FINAL*, then *OK*. We selected *Final* here since we want to log the values of the weight at the end of each epoch.

1b. Repeat step 1 selecting the input->hidden layer projection instead.

2. Create a new GridLog with *.logs / New / GridLog*, then *OK* the popup, and select *Epoch_1* as the updater for this log.

3. Now comes the interesting part. We're going to re-arrange this display so it displays the weights in a way that better reflects the network's structure. To do this we are going to change the geometry and layout of the weight matrices for each projection. In the GridLog there is a "header" at the top associated with each column in the log (the sum-squared error, the EpochCounterStat (labeled *epoch*) and each of the two projections you are monitoring, each labeled *wt*). We can manipulate these headers with the mouse to rearrange their layout.

3a. Use the middle button (or shift plus left button) to reshape the layout. We first want to reshape the hidden-to-output weights to be 2 units wide by 4 units tall, so that each row of units will be the weights from one of the four output units. Move the mouse to the right hand side of the first *wt* grid, and middle-click and hold it down while dragging upwards and to the left into a shape that is 4 tall and 2 wide — it will not let you configure a shape that doesn't hold all 8 of the weight values, so you need to keep that in mind as you configure. It may take a few tries to get this right.

3b. Next we want to reshape the input-to-hidden weights to be 4 units wide by 2 tall, so that gain one row represents the weights for one receiving unit. Do this using the right mouse button. You can also use the left mouse button to move the columns around, and if you want to relabel the columns, that can be done by right-mouse-button clicking on the headers.

4. Then, press *Run* in the test epoch process control panel to see the weights! To verify the weigh values and understand how they correspond to those in the NetView, click on *r.wt* in the NetView, (make sure you are in *View* mode), and select the first hidden unit. The weights from the input units should be the same as those in the GridLog for the first row of the bottom set of weights. Similarly, the second hidden unit's weights are those in the second row of the bottom set of weights. The next set of weights are best viewed using *s.wt* to look at the *sending* weights from the hidden units to the output units. The sending weights for the first hidden unit are shown in the top row of the top set of weights in the

GridLog. Those for the second hidden unit are in the second row. This will all be much clearer in a fully trained network!

4.3.6 A Few Miscellaneous Things

Before you actually run the project, we'll mention a couple of final things you will want to do.

You can watch the state of the network in the **NetView** while it learns and during testing. The NetView is automatically updated by the Trial processes in both the training and testing process hierarchies. (You can have it updated by other processes by selecting *View:Object / Add Updater / <process>* in the NetView, or by clicking on a process and a network in the project view, and hitting *Add Updater*). Both of these Trial processes will send the **NetView** an update signal, so you can see the values of whatever state variable you'd like to look at updated after each trial of training or testing. You might select `act` as the variable to display in the NetView, toggle `Auto Range` off, and set the `max` and `min` on the color scale to 1 and -1. That way the meanings of the color in the color scale stay fixed as the various patterns are presented.

If you are recording the project using a script, you might want to turn off the scripting process at this point, since the network creation process is complete. De-iconify the script and click *StopRec*. You may *Edit* your script (which should pull up an editor and allow you to view the script file), to see what steps were taken, etc.

Regardless of whether you recorded a script, you will want to save the state of the project as it is now, so that it can simply be re-loaded from a project file, just like the XOR example. Select *Object / Save As* in the *Project* window, and specify a file name for saving (a ".proj.gz" will automatically be added to the end of the file name).

4.3.7 Training and Testing Your Network

Finally, you are ready to run your project.

Now you can run through a Test — just hit *Run* in the Epoch_1 control panel. You will see the four patterns flicker through the **NetView**, and you will see them logged in the **GridLogView**. The outputs all look pretty much the same, at this point. But now you can start to train the network. Click on *Step* or *Run* in the Train control panel, and you are off and running.

You can play around with parameters, etc. as described in the previous tutorial on running XOR.

There is one very important aspect of PDP++ which has not yet been demonstrated in the tutorial, which we can step through at this point. This is the ability to apply different parameters to different parts of the network. For the purposes of demonstration, we will assume that we want to have one learning rate for the input-to-hidden weights, and another learning rate for the hidden-to-output weights. The basic idea about how to do this is to create two different **ConSpec** objects, one of which applies to the input-to-hidden weights, and another of which applies to the hidden-to-output weights. An important facility in PDP++ is the ability to create a *child* spec that automatically inherits a set of parameters from a *parent* spec, and has a set of unique parameters. Thus, for the present example,

we would want to create a child ConSpec that inherits everything from the parent except for the learning rate, which will be unique to it. Thus, we will have two conspecs that are identical except for the learning rate. The advantage of this setup is that if you should decide to manipulate other parameters such as momentum, weight decay, etc, the two specs will automatically get the same values of these changed parameters.

To create the new child ConSpec, locate the Project window, and do *.specs / New Child / BpConSpec_0* (note that *New Child* is at the bottom of the menu). This will bring up a New object dialog, specifying that the new BpConSpec will be created as a child of the existing one. Select *Ok* with the right mouse button, so that the new child will be edited. Then, click the mouse into the *lrate* field, and enter a new learning rate parameter (e.g., .01). Notice that the little check-box next to this field was checked when we clicked in lrate. This indicates that this is a *unique* parameter, while the other, non-checked boxes are *inherited* parameters. To see how this works, let's edit the parent spec. At the top of the edit dialog, select *Actions / Find Parent*, which will bring up the edit dialog for the parent con spec. Notice that the parent does not have any of the unique check boxes, since it does not inherit from anything else. Now, change another parameter, like *momentum*, in the parent dialog, and press *Apply*. The *Revert* button on the child spec is highlighted, so press it. Notice that the child dialog displays the new (changed) momentum value, but retains the unique learning rate parameter entered before. Elaborate hierarchies of specs can be created, and the patterns of inheritence and unique parameters provides a very clear indication what is specialized about specific spec relative to the others, and what it has in common.

Before closing the edit dialogs, it is a good idea to label them with mnemonic names – call the parent "input-to-hidden cons" and the child "hidden-to-output cons", for example. Then click *Ok* on both edit dialogs.

Having created the specs with different learning rates, we now need to specify that one spec applies to one set of connections, and another applies to the other set. As it is now, the parent spec applies to all projections, since it is the default that was created automatically when the connections were created. The best way to set specs is using the NetView, which has a *Selections* menu that operates on selected items in the NetView (it is on the view half (right side) of the window, since it operates on the items selected in a particular view). Thus, make sure you are in *Select* mode, and select the projection arrow that goes from the hidden to the output layer. Then, select *Selections / Set Con Spec*. This will bring up a popup dialog, where you should select *BpConSpec / hidden-to-output cons*, and click *Ok*. This has now set the selected projection to use this con spec instead of the default. To verify the status of the network, click in the backround of the NetView to unselect everything, and then choose *Selections / Show Con Spec*. In the dialog, select *hidden-to-output cons*, and click *Ok*. The correct projection arrow will be selected in the NetView, indicating that it is using that con spec. You can repeat this to verify that the other projection is still using "input-to-hidden cons". Thus, the *Selections* menu allows you to both set and verify which objects in the network are using which specs. Notice that you can set many other properties using this menu much in the same way we just did for ConSpecs on projections.

At this point, you might want to look at some of the other demo projects available. These are located in the '`demo`' directory where the XOR example project was. Check out the '`README`' files for further information.

5 How-to Guide

This chapter provides step-by-step instructions on how to perform various commonly-used operations. It contains pointers to the other sections of the manual that provide more information about each step of the operation, so look for something close to what you want to do if you can't find it exactly. Also, many questions can be answered by looking at the relevant section of the manual.

It also contains a description of the `Wizard` object, which automates many of these commonly-used tasks.

5.1 Questions about Processes

How to I set a stopping criterion for training?

The stopping criterion is part of the statistic process that has the data on which the criterion is based. Thus, if you want to stop training based on sum-squared-error, the **SE_Stat** (see Section 12.6.1 [proc-stats-se], page 202) is where to go to set the criterion. The criterion is always set in the statistic that is located in the `loop_stats` of the process you wish to stop. So, if you wan to stop training, look in the **TrainProcess** (see Section 12.3.2 [proc-levels-train], page 191) for the statistic.

How do I record the activations of the units in my network?

This is accomplished by creating a **MonitorStat**, which gathers information from the network and makes it available for logging. The easiest way to do this is through the **NetView** (see Section 10.6 [net-view], page 149), where you can select the objects to be monitored, the variable to monitor, and it will create an appropriate MonitorStat for you. Or, you can use `Stats/RecordLayerValues` from the **Wizard** (see Section 5.5 [how-wizard], page 51). Note that the monitor stat is created in the schedule process that corresponds to the time-grain at which you want the information recorded. It is the log of this process which will show the values you are recording. For more info, see Section 12.6.2 [proc-stats-monitor], page 203.

How do I analyze the representations in my network (e.g., cluster plots, PCA, MDS)?

There are two basic strategies. Both start by recording data from the appropriate layer in the network (e.g., the hidden layer), using a **MonitorStat** as described above. The simplest, but less automated, technique is just create a log of the recorded data (e.g., if recording at the Trial process level, create a log of the trial process – any kind of log will do, but a **GridLog** might be the most appropriate), and then use the `Analyze` menu on the log to analyze the data (see Section 13.3.1 [log-views-logview], page 212). When using this technique, you need to remember to clear the log at the appropriate time so it contains just the data you want to analyze.

A more automated technique involves copying the monitored information into an environment using a **CopyToEnvStat** – the environment serves as a kind of data buffer for the network activations, and contains the necessary analysis

routines. Once the data is copied into the environment, you can just use the `Analyze` menu, or better yet, create a **DispDataEnvProc** that automatically displays the results of an analysis of a data environment. This latter technique is employed by the **Wizard** `Stats/Analyze Net Layer` function, which automates all of the above steps, and is highly recommended. Pieces of these steps are also found in related wizard functions such as `Auto Analyze Data Env` and `Save Values in Data Env`. See Section 5.5 [how-wizard], page 51.

Regardless of the technique, the set of available analyses, which include cluster plots, Principal Components Analysis (PCA), and Multidimensional Scaling (MDS) are described in the **Environment** `Analyze` menu (see Section 11.9 [env-analyze], page 179).

How can I get receptive field plots from my network?

The **UnitActRFStat** (see Section 12.6.5 [proc-stats-actrf], page 206) and **DispNetWeightsProc** (see Section 12.4.5 [proc-special-misc], page 199) processes both support the display of receptive fields. The former uses a reverse-correlation technique that works for any set of layers, regardless of connectivity, while the latter only plots directly connected sets of weights. These are supported by the **Wizard** `Act Based Receptive Field` and `Display Net Weights` functions (see Section 5.5 [how-wizard], page 51).

How do I record a reaction time measure (cycles) from my network?

Experience in a variety of cases has shown that human reaction times can be best modeled by recording the number of processing cycles it takes for a response/output layer unit to exceed some kind of activity threshold. This can be measured with the **ActThreshRTStat** Section 12.6.6 [proc-stats-rt], page 206 (see also the **Wizard** `Stop On Act Thresh` function, Section 5.5 [how-wizard], page 51). One can also use the **CyclesToSettle** statistic (see Section 12.6.8 [proc-stats-misc], page 207) to record however many cycles were used during a **Settle** process, but be careful because the settle process is typically stopped by a statistic that records the change in activation over time (in '`cs++`' this is **CsMaxDa**; in '`leabra++`' it is **LeabraMaxDa**), to ensure that the activation changes have gone below some threshold. This kind of change-based reaction time measure is typically not such a good measure of human reaction time.

How do I run my network in the background for training?

Often networks take a while to learn, and it is nice to be able to run them in the background for a day or two. To do this, get the network all setup to train, save it, and then use the following type of command:

```
unix > pdp++ -nogui -f startup.css <myproj> <extra args> &
```

where '`pdp++`' refers to the PDP++ executable you are using (bp++, cs++, etc), and `-nogui` tells it to not open up the graphical user interface, and `-f startup.css` is a script file that will automate the process of loading the project and running it. For more information on different startup arguments for PDP++, see Section 9.3 [proj-startup], page 124. Also, check out the comments at the top of '`startup.css`' (located in the '`css/include`' directory) to see what kinds of assumptions it makes about your project. Also see Section 9.6 [proj-scripts], page 128 for more useful scripts that you can use. See Section 9.4

[proj-signals], page 125 for information on how to control the process once it is running in the background.

How do I automatically save my networks after training?

This can be accomplished by creating a **SaveNetsProc** or **SaveWtsProc** in the `final_procs` group of an appropriate Batch or Train process. This process names the network with the name field of the network object plus the current epoch and, if present, batch counters. You can set the mod value of this process to save the networks at different intervals. An example project using this is '`demo/bp/gridsearch_xor.proj.gz`'.

There is a CSS script version of this process called '`save_networks.css`' in the '`css/include`' directory, as well as a modified version that saves networks that get below some criterion (which is presumably higher than the stopping criterion), called '`save_good_nets.css`'. These scripts are meant to be attached to **ScriptProcess** objects created in the `loop_procs` group of an appropriate Batch or Train process. They basically just issue a save command on the network, but they also give the network a name based on the current epoch and/or batch counters. See these scripts for more detailed documentation.

How do I change one of the processes in my schedule process hierarchy?

All objects can be changed to a different type using the Change Type menu option (ChangeMyType function in CSS). In the project view, just select the object you want to change, and hit the Change Type button, and it will change the type, and reset all the relevant pointers so your process hierarchy should remain intact.

How do I remove a higher-level process object without losing the lower-level ones?

If you remove a process object by using the *.processes/ Remove* option, it will automatically remove all the processes below the one removed. However, if you do *Object/Close* on the object itself, it will just remove itself and close-up the gap between the remaining processes (in terms of stat aggregation, etc). The *Rmv Obj(s)* button on the project view also does this kind of remove. Finally, you can use the *Structure/RemoveSuperProc* and *RemoveSubProc* menu options on a process object itself to remove its parent or sub process.

Why can't I change the environment or network pointer in my process?

The `environment` and `network` pointers are automatically inherited from the parent process in the hierarchy (this is true for stats and other process objects hanging off of a given process as well). This means that you can only change these at the top-level process in a given hierarchy. On the other hand, it means that when you want to change these pointers, you only have to change them in one place and they are automatically propagated to all processes (see Section 12.1 [proc-base], page 185).

Where do I create a given statistic process?

Each statistic process "knows" where to create itself in the standard schedule process hierarchy. This default location will be the one shown when you use the *NewStat* menu option in the *.processes* menu on the project. Use this to create your statistic. In general, most statistics should be created in a relatively low-level process, such as the **TrialProcess**, since most stats deal with data that

changes event-by-event. Remember that you create the stat at the lowest level so that it can compute the raw information, and these low-level values can be aggregated up the processing hierarchy if higher-level information is desired, see ⟨undefined⟩ [proc-stats], page ⟨undefined⟩.

Why can't I change the name of my statistic?

First, the name of the statistic object is not the same as the label that is associated with that object in the log (though they are related), so changing one will not change the other. The Stat object names are automatically set to reflect the aggregation and layer settings of the stat, see ⟨undefined⟩ [proc-stats], page ⟨undefined⟩. This is true as long as the name contains the type name of the stat (e.g., SE_Stat), so if you want to give a stat a custom name, make sure it doesn't include the type name.

To change the label associated with the stat as it *appears in the log*, you need to edit the **StatVal** object(s) that contains the value of the statistic. This object contains a `name` field, which is what determines the label used by the log. However, as it appears on most stats (e.g., SE_Stat), the `name` field is hidden, so you have to use the CSS script to set it. Typing:

```
css> .processes[1].loop_stats[0].se.Edit();
```

will pull up an editor for the epoch level SE_Stat StatVal (which is the member `se` of an SE_Stat) in the '`demo/bp/xor.proj.gz`' demo, and allow you to edit the name. Alternatively you can use *PDP++Root/Object/Browe* and click your way through the process hierarchy until you get to the variable you want to change, and then hit *Select* to edit it. You will have to do a `GetHeaders` on the log to see this new name. Other stats like ScriptStat have a list of StatVals, which, when edited, allow you to edit the names directly.

An easier alternative is to simply change the `display_name` of the **ViewSpec** in the relevant log.

How do I get the epoch counter into my testing process?

The epoch counter, which is on the **TrainProcess** is not directly available to a testing process, which typically is created at the epoch level. Thus, you need to add a **EpochCounterStat** to the final stats of the process where you want to see the epoch counter. This gets the counter off of the network itself. To get other counters from the training process hierarchy (i.e., the batch counter), use the **ProcCounterStat**. For more info, see Section 12.6.8 [proc-stats-misc], page 207.

How do I determine which event is closest to the output my network actually produced?

The **ClosestEventStat** does exactly this, see Section 12.6.3 [proc-stats-close-event], page 204.

How do I create a cross-validation setup?

Cross-validation is accomplished by periodically testing during training. Thus, you simply need to create a testing process hierarchy, (at the Epoch level), and link your testing epoch process into the `loop_procs` of the training process. This can be done with the **Wizard** `Processes/Cross Validation` function (see Section 5.5 [how-wizard], page 51). You should set the modulo factor (in the `mod` field of the process) of your testing epoch process to reflect the frequency with

which you want to perform testing. See 'demo/bp_misc/bp_xval.proj.gz' for a demonstration of a cross-validation training setup.

How do I set the stopping criteria for training based on testing error (e.g., for cross-validation)?

Add an aggregator stat of the testing epoch stat in the final_stats of your training epoch process (i.e., make an **SE_Stat** there, set its time_agg.from to point to your test epoch **SE_Stat**, use the LAST operator). You will want to create aggregates of this stat (which is itself an aggregator) using LAST, and set the stopping criterion in the aggregator of this stat in the loop_stats of the train process. This is all accomplished by the **Wizard** Processes/Cross Validation function. The project 'demo/bp_misc/bp_dig_xval.proj.gz' contains an example of this.

How do I graph both training and testing error (e.g., for cross-validation)?

If you follow the steps for the previous item, a graph log at the level of the training epoch process will show both the training and testing error. Again, see the demo project.

How do I make a hybrid network of two different types of

algorithms? Since all schedule process types assume that a given network has all of the appropriate types of objects (i.e., **BpUnit**'s, **BpCon**'s, etc for backprop), you have to create two separate networks, and then link them together with a **BridgeProcess**, see Section 12.4.4 [proc-special-bridge], page 198.

How do I setup distributed memory processing across events?

The EpochProcess supports distributed memory processing of events (using the MPI protocol), where each processor operates on a different set of events, sharing its results with the other processors to achieve processing of the entire epoch. This implies batch-mode weight updates. This is very efficient, and can produce nearly-linear speedups in the number of additional processors used. See Section 12.3.3.1 [proc-epoch-dmem], page 193 for details.

5.2 Questions about Networks

For information about how to build a network using the network viewer, see Section 10.7 [net-build], page 160 and the tutorial Section 4.3 [tut-config], page 33. Also, many questions can be answered by looking at the chapter on networks Chapter 10 [net], page 131.

How to I get certain units to use a different parameter than others?

For example, if you want certain units to use a different learning rate, or activation function, or gain, etc... This is done by making a new **UnitSpec** or **ConSpec** (depending on where the relevant parameter is), and telling the relevant units or connections to use this new spec. It is recommended that you create the spec as a child of an existing spec (see Section 8.4 [obj-spec], page 115), so that all the other parameters will automatically be inherited by the new spec (except for the one you change). The easiest way to apply a different spec is to select the relevant units or projections, and use the *Selections* menu on the **NetView**

(see Section 10.6 [net-view], page 149) to set the spec. The tutorial now has an example of how to do this (see Section 4.3.7 [tut-config-running], page 41).

What is the difference between Projections and Connections?

Projections specify the broad patterns of connectivity between layers. Connections are the actual unit-to-unit weights and other parameters which actually implement this connectivity. Thus, there is always a projection associated with a set of connections. See Section 10.3 [net-prjn], page 137 for more details.

How do I implement weight sharing/linking?

The **TesselPrjnSpec** (see Section 10.3.3.2 [net-prjn-tessel], page 139) and the **LinkPrjnSpec** (see Section 10.3.3.5 [net-prjn-misc], page 143) are two types of projection specifications (see Section 10.3 [net-prjn], page 137) that implement weight sharing. The **TesselPrjnSpec** generates repeated patterns of connectivity, and it can automatically link all of the repeated patterns together with the same set of weights. Thus, a set of units in a receiving layer can all have the same receptive field from a given sending layer, and all of the units can use the same set of weights to define their receptive field. The **LinkPrjnSpec** allows individual or small groups of weights to be specifically linked together, even if these connections are in different layers in the network. It does not generate any connectivity itself, it simply causes existing connections to share weights.

Can I temporarily lesion a layer in my network?

Sometimes, one wants to pre-train part of a network on one task, and then subject the rest of the network to some more complex task. This process is greatly facilitated by being able to create the entire network at the outset, and then temporarily "lesion" certain layers during pre-training. This can be accomplished by simply checking the `lesion` flag on the **Layer** object (see Section 10.2 [net-layer], page 135).

Are there functions for lesioning the weights or units in the network?

Yes, `LesionCons`, `AddNoiseToWeights`, `TransformWeights`, and `PruneCons` all perform various manipulations on the weights in a network, and could be used to simulation "lesions" of the network. `LesionUnits` lesions units. These functions, like most in the network, can be called at various levels of granularity from a single group of weights (or units) up to the entire network. See (see Section 10.1 [net-net], page 131) for details.

How can I use a specified (non-random) set of initial weight values?

There are several ways to do this. One is to write a CSS script to set the weight values by reading them in from a file or from values coded into the script itself. This script can be attached to a **ScriptPrjnSpec** so it is run automatically when the network is connected (see Section 10.3.3.5 [net-prjn-misc], page 143). It is also possible to use a **TesselPrjnSpec** (see Section 10.3.3.2 [net-prjn-tessel], page 139) or **RandomPrjnSpec** (see Section 10.3.3.3 [net-prjn-random], page 141) in conjunction with the `init_wts` flag to specify initial weight patterns, which are used instead of the random ones whenever the network is initialized (see Section 10.3.2 [net-prjn-spec], page 138). You could also construct a "donor" network that had the intial weights set as you wanted them (by hand or whatever), and then use the `CopyFrom` or `Copy_Weights` function

to initialize your training net from the donor net (see Section 10.1 [net-net], page 131). Similarly, you could use `WriteWeights` and `ReadWeights` to save and load weights from a file.

Is there a way to view the weights for a set of multiple units at the same time?

Yes – the function `GridViewWeights` on the network (in the *Actions* menu) will display the entire weight matrix between two layers of the network on a GridLog. Also, you can plot a matrix of events from an environment using the `EnvToGrid` function on an environment (*Generate* menu). This is useful for activity-based receptive fields computed via the **UnitActRFStat** (see Section 12.6.5 [proc-stats-actrf], page 206), which are stored in an **Environment**.

How do I setup distributed memory processing across multiple processors?

The Network object supports distributed memory processing of connections (using the MPI protocol), where each processor maintains a different set of connections and performs operations on only this subset, sharing its results with the other processors to achieve processing of the entire network. See Section 10.1 [net-dmem], page 134 for details on how to configure this. Given the relatively large amount of communication required, this is efficient only for relatively large networks (e.g., above 250 units per layer for 4 layers). In benchmarks on Pentium 4 Xeon cluster system connected with a fast Myrinet fiber optic switched network connection, networks of 500 units per layer for 4 layers achieved *better* than 2x speedup by splitting across 2 processors, presumably by making the split network fit within processor cache whereas the entire one did not. This did not scale that well for more than 2 processors, suggesting that cache is the biggest factor for this form of dmem processing. However, there is also the ability to distribute events across multiple processors, which achieves more reliable speedups (see Section 12.3.3.1 [proc-epoch-dmem], page 193).

5.3 Questions about Environments

How do I present events with different frequencies?

There is a type of environment that implements frequency-based presentation of events. See Section 11.10 [env-freq], page 181 for details. Note that you can do `Change Type` on an existing environment to change it to a frequency environment, retaining all the patterns, etc. However, you'll also need to convert the individual events to **FreqEvents** using change type on them. An alternative is to create a new FreqEnv, create the proper number of Freq Events, and then do `CopyTo` on the original environment to copy its stuff to the new one.

How do I present sequences of events in order?

This requires both a structured environment and a set of processes that can use that structure to present sequences. The environment must simply have the events in sub-groups where each sub-group defines a different sequence of events (see Section 11.3 [env-seq], page 169). The **SequenceEpoch** and the **SequenceProcess** work together to present sequenced events. The **SequenceEpoch** iterates through the different sequences (groups) of events (and determines if they are randomized or not at that level), and the **SequenceProcess** iterates

through the events in a given sequence (and determines the order of the events themselves). See Section 12.4.1 [proc-special-seq], page 195 for details.

How do I read patterns/events from a file?

The function `ReadText` on the **Environment** object will read in events from a text file, including files formatted for the old PDP software. This file has a simple format. See Section 11.7 [env-import], page 176 for details on the file format and how to use the function.

How do I read patterns/events from a file INCREMENTALLY during training?

The object **FromFileEnv** (see Section 11.11 [env-other], page 182) reads events one epoch or one event at a time from a file (either text or binary format) for presentation to the network. This should be useful for very large environments or very large patterns, or both. Reading one event at a time uses the "interactive" interface to the environment (GetNextEvent) meaning that the **InteractiveEpoch** epoch process (see Section 12.4.2 [proc-special-inter], page 197) must be used.

How do I present an arbirarily-selected event to the network?

First, view the Environment that contains the event, scroll the list of events to the desired one, and use the right-mouse-button to edit that event. This brings up an 'Edit' dialog, containing a 'PresentEvent' button at the bottom. Clicking this button will present the event using the selected TrialProcess (which must already be created, and determines which network is used, etc).

How do I have some events that set the output target, and others

that don't, or more generally, have events do different things within the environment? How an event is presented to the network, and the very configuration of the event itself, is determined by the **EventSpec** and its associated **PatternSpec**s. These can be configured by hitting *Edit Specs* on the **EnviroView**. To have different events do different things, you need two (or more) different event specs. Once you have configured the different event specs, you can click on events (after doing *Edit Events*) and select *View: Action/Set Event Spec* to set the event specs for the selected events.

How can I quickly update my Event specifications (pattern layout, size) from corresponding layers on the network.

Within the environment viewer, select **UpdateAllEventSpecs** in the *Actions* menu – this will automatically update the events to match the current network geometry. This function is also available in the Wizard.

How do I create an interactive environment where events depend on network responses?

This is implemented through a combination of an **InteractiveScriptEnv** environment (see Section 11.11 [env-other], page 182) and and an **InteractiveEpoch** epoch process (see Section 12.4.2 [proc-special-inter], page 197). See '`demo/leabra/nav.proj.gz`' for a working example.

How do I label the event pattern values so I know what they stand for?

The `value_names` field in the **PatternSpec** provide labels for the pattern values. These can be edited in the enviro view in the *Edit Specs* mode, by clicking on

a pattern and hitting the **Edit Names** button. To see them, you need to make sure the **Val Txt:** field in the viewer is set to **NAMES** or **BOTH**.

5.4 Questions about CSS

How do I know what functions or members are available in CSS?

Use the 'type' command, such as 'type UnitSpec', which will display a list of all CSS-accesable type information for that object, including descriptive comments. If you just type 'type' with no arguments, then a list of all defined types is presented.

How do I call the copy operator (=) for a hard-coded objects in CSS?

Hard coded types are a little different than types defined within CSS (e.g., a Unit is a hard-coded type), because they are used to refer to things like units, layers, etc, in the network, which are external to css. In particular, they are *always* pointers. There is no such thing as a 'Unit un' in css – its always really (implicitly) 'Unit* un'. So, dereferencing a pointer to a hard-coded object doesn't do anything, meaning that it can't tell the difference between assigning the pointer to point to a new object, and copying the object that the pointer currently points to. To explicitly invoke the object's copy operator, use the CopyFrom() member function (e.g., Unit* un = .units[0]; un.CopyFrom(.units[1]);). Note that if you obtain a pointer to a hard-coded object via a "path" in the object hierarchy, then it knows that this is not just a free css pointer, and it will apply CopyFrom when you do '='.

How do I deal with errors in Scripts that are auto-running at startup?

If you startup the program with the project name on the command line, then the project will load over and over again if there is an error with a script that is set to run upon loading the project (auto-run). The solution is to load the project from the Root menu, and to immediately move the mouse over the xterminal window, and type a Return into that window when the error occurs. Then, you can debug the problem with the script at the point where the error occurred.

5.5 The Wizard Object

The Wizard object is located in the .wizards menu on the project – one is automatically created whenever a project is created or opened. In the default color scheme, the wizard has a whitish-blue-green (azure) background. There are 6 main categories of actions that the wizard can perform, corresponding to the main types of simulation objects that can be created, which are arranged as button-menu's along the bottom of the Wizard dialog.

Typically, the wizard actions prompt you for all the information required – the one exception is the Network configuration, which is maintained in the values in the wizard dialog (`n_layers`, `layer_cfg`, `connectivity`). The `Defaults` menu has options for standard configurations of these network parameters.

So, if you want to make a network other than a simple three-layer architecture (the default), select `Defaults/Multi Layer Net` and specify how many of each type of layer (input, hidden, output) you want.

General usage: in general you can just proceed left-right, top-bottom in the menu buttons. Once you have the configuration of the network as desired (either from the Defaults functions or just by setting by hand) then **Network/Std Network** will make the network for you. The other options below this (for some versions) contain other optional kinds of network structures that can be constructed. For example, `bp++` has an option for making an `SRN Context` layer for simple recurrent networks, and `leabra++` has a number of options including unit-based inhibition (instead of the usual kWTA).

The **Environment/Std Env** will construct a standard (basic) environment to fit the network configuration. If you subsequently change the network configuration, the `Update Env Fm Net` will sync them back up. Other options include making event groups (`Sequence Events`, `Time Seq Events`). Note that the environment object, once constructed, has some useful generation functions in the `Generate` menu (see Section 11.8 [env-gen], page 178).

The **Processes/Std Procs** menu will make a standard process hierarchy for training your network, starting with a Batch process at the highest level. The easiest way to manipulate these processes once created is in the Project view (see Section 9.2 [proj-viewer], page 120). However, the `Processes` menu contains a number of options for commonly-used process types such as automatically saving networks (at the end of training), setting up an automatic testing process, cross-validation, and configuring the processes to deal with sequences of events within an epoch (event groups) (see Section 12.4.1 [proc-special-seq], page 195).

The **Stats** menu can be used to create statistics for monitoring the network as it trains. `Record Layer Values` creates a `MonitorStat` (see Section 12.6.2 [proc-stats-monitor], page 203) to record layer data in a log associated with a given process. Functions `Save Values In Data Env`, `Auto Analyze Data Env`, and `Analyze Net Layer` all support the analysis of recorded values (see Section 11.9 [env-analyze], page 179). The `Act Based Receptive Field` creates an **UnitActRFStat** for recording a receptive field for any layer in the network from any other set of layers (even if they are not connected) using a "reverse correlation" technique as described in Section 12.6.5 [proc-stats-actrf], page 206. It also automatically displays these weights at a higher level of processing. `Display Net Weights` makes a **DispNetWeightsProc** (see Section 12.4.5 [proc-special-misc], page 199). `Stop On Act Thresh` makes an **ActThreshRTStat** to stop settling when network activations exceed threshold, providing a reaction time measure (see Section 12.6.6 [proc-stats-rt], page 206). `Add Counters to Test` and `Get Stats From Proc` provide ways of sharing information across different process hierarchies (e.g., training and testing). `Add Time Counter` creates a **TimeCounterStat** that generates an ever-increasing time counter that cuts across processing hierarchy loops, and is initialized by a **TimeCounterResetProc** (see Section 12.6.7 [proc-stats-ctrs], page 207).

The **Logs/Std Logs** menu creates standard logs (trial, epoch and batch), while `Log Process` provides a way of creating a log for a given processing level.

6 Guide to the Graphical User Interface (GUI)

This chapter provides a general guide and reference for using the graphical user interface to the PDP++ software. This covers all of the generic aspects of the interface—details about specific parts of the interface like the network viewer are found in the the section of the manual that covers the object in question (e.g., Section 10.6 [net-view], page 149).

6.1 Window Concepts and Operation

In PDP++ the hierarchy of objects provides the basis for most of the gui (Graphical User Interface) interaction. To access a sub-object of a class, the best place to start is with the gui window for the parent class instance and work your way down to the sub-object. The higher levels of the hierarchy have windows which are mapped to the screen when the object is created. These windowing objects inherit from the base class **WinBase**. When PDP++ starts up, only one of these classes has been created and thus there is only one window on the screen. The initial object is an instance of the class PDPRoot, and is the top of the PDP hierarchy.

6.1.1 How to operate Windows

PDP++ relies on your window manager for positioning, and iconifying the graphical windows of the program. Please refer to your window manager's manual for more information on the mouse movements and button presses needed to accomplish these tasks. For all the window objects in PDP++ there are CSS commands which ask the window manager to position or iconify the windows associated with the object. It is up to the window manager to provide the correct behavior for these "hints". PDP++ WinBase Window's position and iconification status can be manipulated with the following commands:

`GetWinPos()`
: Stores the window's current position and size on the object. When the object is saved the position and size of its window will be saved as well so that the window has the correct geometry when the object is loaded at a later time.

`ScriptWinPos()`
: Generates css script code for positioning the window at its current location and prints the code to the output window or to a recording script.

`SetWinPos(float left, float bottom, float width, float height)`
: Asks the window manager to resize and move the window to the specified parameters. If no parameters are given, this functions uses the parameters stored on the object.

`Resize (float width, float height)`
: Asks the window manager to resize the window to the specified parameters. If no parameters are given, this functions uses the parameters stored on the object.

`Move (float left, float bottom)`
: Asks the window manager to move the window to the specified parameters. If no parameters are given, this functions uses the parameters stored on the object.

`Iconify()`
: Asks the window manager to iconify the window.

`DeIconify()`
: Asks the window manager to deiconify the window.

6.1.2 How to operate Menus

In all WinBase windows, there is a horizontal menubar along the top. In this menubar is a "Object" menu (see Section 6.2 [gui-object], page 55), and also "Subgroup" menus (see Section 6.4 [gui-subgroup], page 57) for access to the sub-objects in this class.

To access the menus press and hold button-1 (left button) on the mouse while the mouse pointer is over the menu name. A smaller vertical menu window will pop up under the mouse pointer. Moving the mouse vertically while the button is still pressed will highlight the different choices in a the menu. Some of the items in the menu may have three dots '...' after them. When highlighting these menu items a cascaded menu will popup to the right of the selected menu item. The cascaded submenu may be traversed by moving the mouse pointer horizontally into the submenu, and then moving the mouse vertically as before. Again, button-1 must be pressed and held down during this operation. To select an item, release the mouse button while the mouse pointer is over the highlighted selection. To cancel the menu (to select nothing) move the mouse out of the menu and release the mouse button. Menu items which spawn submenus may not be selected. A menu may be pinned (it will stay on the screen after the mouse is released) by releasing the mouse while the pointer is on the menu name or on a submenu's name. A selection can be made from a pinned window by clicking (pressing and then releasing a button) on the desired selection. This will also cause the pinned menu to become unpinned and it will disappear.

Note: This applies specifically to the motif mode in InterViews. Open-look or other modes may be slightly different.

6.1.3 Window Views

Some objects may have multiple windows. These multiple windows are called views, and provide alternative methods of interacting with the same object and its data. **In a view window then menubar at the top of the window is split into a left menubar and a right menubar.** The left menubar contains menus whose actions pertain the the base object of the view. The right menubar contains menus whose actions pertain to this particular view only. For instance, a network object may have multiple netviews. In each netview there will be the left and right menubars along the top of the window. In the left menubar one would find menus with function that are particular to the network, such as menus for adding new layers, or removing all connections. In the right menubar on would find menus with functions that are particular to the view, such as setting the colors or shape of the units in the view.

Each view has an associated list of Schedule processes (see Section 12.2 [proc-sched], page 186) which update the view in the course of their processing. By adding or removing these "updaters", one can control the grain at which changing data is displayed. For instance, in the netview the colors of the units change to reflect their current values whenever the view is updated. If the updater was a trial level process then the view would display new values for all the units after every trial.

WinView Functions

`AddUpdater(SchedProcess* updater)`
: Add the schedproc to the list of updaters for this view. (and vice-versa).

`RemoveUpdater(SchedProcess* updater)`
: Removes the schedproc to the list of updaters for this view. (and vice-versa).

`InitDisplay()`
: Initializes and graphically rebuilds the view's display.

`UpdateDisplay()`
: Refreshes the winview's display to reflect changes in the base object's values. This is the function that is called by the updater processes.

6.2 The "Object" Menu

The "Object" menu always appears as the left most menu in a WinBase's menubar. The "Object" menu for each object is used to perform file based actions on the object itself. Many of these actions involve the use of a file requester dialog. See Section 6.8 [gui-file-requester], page 69. These actions on the object can also be called through css (e.g., the Print action can be called from css as '`object.Print()`'). The "Object" menu has the following menu actions:

`Load`
: Load a text object dump of an object of the same class as this object on top of this object, replacing the values of fields of this object with the values of the saved object's fields. The saved object file is selected with the file requester.

`Save`
: Save a text object dump of this object in a file created with the file requester, or with the object's most recently used filename for saving.

`SaveAs`
: Save a text object dump of this object in a -new- file created with the file requester.

`Edit`
: The Edit menu action brings up an Edit Dialog on the object. See Section 6.5 [gui-edit], page 58.

`Close`
: The Close menu action will attempt to close/delete the object. If the object is referenced or pointed to by other objects, then it will not actually be deleted, only the windows which display it will be removed. The user must confirm the deletion if it is possible to safely delete the object. **NOTE THAT CLOSE IS NOT ICONIFY!, it really does delete the object, not just close the menu.**

`Copy From`
: Copies from another object of the same or related type — replace all of the current data in the object with those in another. In the menu, only the same

or subtypes of this object will be shown, but in the script, any type of related object can be passed to this function.

`Copy To` Copies the data in this object to another object. This can be useful if you want to copy from a more basic type of object (e.g., Environment) to a derived type (e.g., FreqEnv) – CopyFrom won't show the more basic type of object to copy from, but CopyTo will show the derived type.

`DuplicateMe`
Makes another copy of this object – creates a new object and then copies from this current object to that new object. Note this is DuplicateMe in the script code.

`ChangeMyType`
Changes the type of this object to be another related type (e.g., change to a FreqEnv from an Environment). Will usually do a good job of updating the various links to this object if changed. Not good for objects within a network, or generally for Stat objects that are aggregated. This is ChangeMyType in the script code.

`SelectForEdit`
Allows you to select a field (member) of this object to be edited in a **SelectEdit** object, which consolidates parameters and functions across multiple objects into a single edit dialog (see Section 9.7 [proj-seledit], page 129).

`SelectFunForEdit`
Allows you to select a function (method) of this object to be accessible from a **SelectEdit** object, which consolidates parameters and functions across multiple objects into a single edit dialog (see Section 9.7 [proj-seledit], page 129).

`Help` Will automatically pull up a help browser for information relevant to this object. Depends on the browser actually running on your system, as specified in the Settings on the root object.

`Print` The Print menu action will save a snapshot of the entire object's window to a file in Postscript format using a file requester. Note that this printout file uses structured graphics so it will scale well if resized, etc.

`Print Data`
The Print data menu action will save a snapshot of the window's data, not including the menubars and window decoration, to a file in Postscript format using a file requester.

`Update Menus`
If objects are created or deleted, sometimes the menus of their parent objects can become out of date. If this appears to be the case then use the "update menus" menu action on the parent object to fix the menus. The "update menus" menu action recursively traverses the menus of the object and its subobjects, adding and deleting menu items appropriately.

`Iconify` This will iconify the window (shrink down to an iconic representation, to get it out of your way).

6.3 The "Actions" Menu

Some objects will have an "Actions" menu following their "Object" menu. In the Action menu are functions which are apply specifically to the object and are not the common file based functions found on the Object menu. If you wish to perform a function directly on the object you are viewing in the window, the "Actions" menu is a good place to look. For more specific information concerning an object's "Action" menu please refer to the section of this manual which pertains to the object itself.

6.4 The SubGroup Menu(s)

A subgroup menu appears in the menubar of a WinBase object for each of the group members of the object. The name of the menu corresponds to the name of the group member (e.g., The Layer object has a group of units and a group of projections. It would have two subgroup menus, one labeled `.units` and a second labeled `.projections`. The "." before the name of the subgroup is used to indicate that the subgroups are sub-objects of the WinBase object. In the CSS script language one would access objects in these subgroups using the "." operator. (e.g., To access the first network in the project one would type '`.projects.networks[0]`'). In addition the subgroup menus appear in a non-italicized font to distinguish them from the "Object" and "Action" menus of the WinBase. The subgroup menu's have the following menu choices. Occasionally subgroup menus may add additional menu choices as well (e.g., The Processes submenu of the Project has the menu choice *Control Panel* which opens a control panel dialog for one of the processes in the subgroup). Some of the group operations require the use of a file requester. See Section 6.8 [gui-file-requester], page 69.

`Edit` The Edit action brings up a Group Edit Dialog for the group or an Edit Dialog for an individual object (Section 6.5 [gui-edit], page 58).

`New` The New action allows the use to create new objects in the group or in a subgroup of the group. The user can chose to create objects of the base object type, objects of a subclass of the base object type, or a subgroup object. A popup dialog appears which enables the user to select the number of objects to create, the type of objects, and where to place them. Sometimes the popup dialog may have additional fields and toggles which are particular to the item being created. If the auto_edit flag in the global settings (see Section 6.6 [gui-settings], page 63) is turned on, an edit dialog will be created for the newly created objects when button-1 (left button) is pressed on the OK button of the popup dialog. If button-2 (middle button) is pressed the edit dialog will be created only if the auto_edit variable is off. When button-3 (right button) is pressed an edit dialog will always be created.

`Open In` The "Open in" action allows the user to open a previously-saved object file and add the data into the group. See Section 8.1.3 [obj-basics-files], page 108.

`Load Over` The "Load Over" action allows the user to open a previously-saved object file and overwrite the objects in the group with the data in the file. See Section 8.1.3 [obj-basics-files], page 108.

`Save` The Save action allows the user to save the group or a group element as a PDP++ object file using the file requester or the object's most recently used name for saving. See Section 8.1.3 [obj-basics-files], page 108.

`Save As` The Save As action allows the user to save the group or a group element in a new PDP++ object file using the file requester. See Section 8.1.3 [obj-basics-files], page 108.

`Remove` The Remove action allows the user to remove the group or a group element. The user is prompted with a confirmation dialog to confirm the choice. However, If the chosen object is referenced by other objects then it will not be deleted.

`Duplicate`
The Duplicate action allows the user to add a duplicate of one of the objects in the group to the group or its subgroups.

`Move Within`
This allows the user to move objects to new positions within the group (e.g., for rearranging the order of layers in the network, which is important for the feedforward Bp algorithm).

`View Window`
View Window brings the window associated with selected object to the front, deiconifies it, or creates it if a window does not exist.

6.5 The Edit Dialog

The Edit Dialog allows the user to both visually inspect and modify the values of an object's fields. As an inspection tool, the user can use and edit dialog to check the values of an object's fields, and as an editing tool the user can use the edit dialog to change the values of some or all of those fields. For some objects, the edit dialog is the only representation available for inspecting, or accessing it members. Other objects may have extended edit dialogs or even multiple views which may also allow modification of the object's fields. An edit dialog also presents the user with easy access to the substructures of an object including editing of its subgroups and arrays. In addition the edit dialog may allow access to certain member functions on an object. Edit Dialogs are created by choosing "Edit" from an object's menu or by the `EditObj` command in css (see Section 7.4.6 [css-commands], page 92).

The basic layout of the edit dialog includes a list of object member names which are listed in a vertical column along the left hand side of the display. Clicking on a member name will popup a short description of the member. Certain members may not appear in the edit dialog. This is determined by the `show_iv` field of the global settings (see Section 6.6 [gui-settings], page 63). Typically the unshown members will not be of interest to the average user, or will contain values which are potentially dangerous to change.

The *Show* menu on the right-hand side of the menu bar at the top of the edit dialog will allow you to control how much stuff to view on a case-by-case basis. Typically, you'll only want to switch to viewing *Detail*, which is not viewed by default, but all levels can be controlled in this menu. If you suspect that something is not there which should be, try selecting *Show/Detail*.

Fields that are highlighted in bright yellow indicate that the field value is at variance with the default value for that field.

The values associated with the member names appear to the right. There are a number of different graphical representations of the content of these fields, which are described in the following section.

In addition, there can be special buttons and menu actions available on the dialog. These are described in subsequent sections.

6.5.1 Member Fields

The member fields in an edit dialog can be found to the right of the member name. These member fields can appear in many different forms. Sometimes there may even be more than one on a line.

Field Editor

The Field Editor is used for editing strings (e.g., an object's name), and number values. It appears as a box with text in it. If you click on the box an Ibeam cursor appears at the click location. When the mouse is in the box, the characters you type will be entered at the Ibeam location point as you would expect.

You must place mouse pointer in field to start editing! However, once you have started typing, the mouse pointer need only remain in the overall window.

Some Emacs style editing keys are also recognized, in addition to the standard keypad arrow and related keys:

Ctrl-f moves Ibeam forward (also right arrow key)

Ctrl-b moves Ibeam backward (also left arrow key)

Ctrl-a moves Ibeam to beginning of line (also Home)

Ctrl-e moves Ibeam to end of line (also End)

Ctrl-d deletes character to right of Ibeam (also Del)

Ctrl-u selects the entire line – subsequent typing replaces contents

Ctrl-n, TAB, or RETURN moves the Ibeam to the next field editor

Ctrl-p or Shift-TAB moves the Ibeam to the previous field editor

Alt-n moves to next object in list editor (also PageDn)

Alt-p moves to previous object in list editor (also PageUp)

You can also hit shift-mousebutton2 (middle button) to scroll the text with the hand cursor.

If the item to be edited is an integer then increment/decrement arrow buttons will appear to the right of the field editor. Pressing on these buttons will increase or decrease the value in the field editor. Pressing button-1 (left button) changes the value by 1, button-2 (middle button) changes the value by 10, and button-3 (right button) changes the value by 100. If the mouse button is pressed and held, the action will auto repeat.

Read Only Member Field

The read only member field is like the FieldEditor except the user cannot edit the field. It is used for display purposes only.

Boolean CheckBox

The checkbox is used for boolean values that are either on or off. Clicking on the checkbox changes its values. A checkmark or solid block indicates an ON, TRUE or 1 value, while no checkmark or an empty box indicates an OFF, FALSE or 0 value.

Enum Menus

The Enum menu is used for enumerated types. These types may have one of a number of symbolic values. The menu is shown with the member's current value. by clicking on the menu the other value choices appear in a popup menu. Selecting an alternate value from the menu sets the menu's value to the new value.

SubObject Menus

A SubObject menu is used when an instance of a class is part of the edited object and that instance is not edited inline (see Section 6.5.1 [gui-edit-fields], page 59). The menu has the name "Type: Actions" where Type is the class name of the subobject. The menu contains the following functions as well as object specific functions that are unique to the subobject class.

Load: Load an instance of the subobject class

Save: Save the subobject using a file requester

SaveAs: Save the subobject with its last saved name

Edit: Edit the subobject

Object Pointer Menus

Object pointer menus are used when the member is a pointer to another object. The current value of the pointer is shown in the menubar as the pathname of the object that is pointed to. If the pointer is not set, the name on the menubar will be set to NULL. Clicking on the menubar brings up a popup menu of other objects of similar type. Selecting an alternate object sets the menubar to the name of that object. The popup menu also allows the user to set the pointer to NULL and to Edit the object currently pointed to.

Object Type Menus

The object type menus are used when a member specifies a type class of object. This menu allows the user to select from classes which inherit from it base class. The current selection is displayed in the menubar.

SubGroup Menus

The subgroup menu allows the user to interact with a group member of the object. The name on the menubar is one of the two forms depending upon whether or not the group has subgroups. (see Section 8.2 [obj-group], page 109).

Clicking on the menubar gives the user the following choices:

New Create new object in the group

Load Load a group from a file using a file requester. See Section 8.1.3 [obj-basics-files], page 108.

Save Save the group using a file requester. See Section 8.1.3 [obj-basics-files], page 108.

SaveAs Save the group with its last saved name. See Section 8.1.3 [obj-basics-files], page 108.

Edit Bring up a Group Edit Dialog of the subgroup

RemoveAll
Remove all the elements of the group

Remove Remove an individual element of the group

Link Link an object from another group into this group

Move Rearrange the order of the elements in this group

Transfer Transfer objects from another group into this one

EditEl Edit and individual element of the group

Find Find and edit an object with a given name

Each of these selections will bring up a dialog for the user to specify the parameters to the operation.

SubArray Menus

The subarray menu allows the user to interact with a array member of the object. It appears as an "Edit" menubar. Clicking on the menubar gives the user the following choices:

Load Load an array from a file using a file requester. See Section 8.1.3 [obj-basics-files], page 108.

Save Save the array using a file requester. See Section 8.1.3 [obj-basics-files], page 108.

SaveAs Save the array with its last saved name. See Section 8.1.3 [obj-basics-files], page 108.

Edit Bring up an Array Edit Dialog for the array

Remove Remove a number of elements from the array at a given index

Permute Permute the order of the elements in the array

Sort Sort the order of the elements in the array

El Show the value of a member of the array at given index

Add Add elements onto the end the array

Insert Add a number of given values at a certain Array index

Find Find and Show the index of a member of the given value

ColorEdit Bring up a Color Array Editor for number arrays

Inline Fields

Inline Fields are used for small substructures within an object (e.g., an object's position, which is only three values: x,y,z). Instead of the expected SubObject menubar, the object is laid out horizontally across on line in the Edit Dialog. The names of each of the fields are often truncated so that the line will not take up too much space. This allows quicker access and visualization without having to traverse through multiple dialogs. The fields within an inline field behave as do the rest of the fields in the dialog.

6.5.2 Edit Dialog Buttons

At the bottom of the Edit Dialog is a row of buttons which have actions pertaining to the Edit Dialog's fields. Some of the buttons may be appropriately inactive due the state of the edit dialog. They will become active when their action is applicable. The following buttons will be available:

`Ok` The Ok button stores the information in the dialog's fields in the object's fields and closes the dialog window.

`Apply` The Apply button stores the information in the dialogs fields but does not close the window. If the apply button is highlighted the edit dialog may contain changes which have not been applied yet. The apply action can also be generated by pressing the Return key in the body of the editor, unless the editor has multiple field editors within it (see Section 6.5.1 [gui-edit-fields], page 59).

`Revert` The Revert button reloads the information from the object into the dialog's fields, clearing any changes the user might have made. If the revert button is highlighted the object's fields might have been changed "behind the scenes" by another part of the program in which case the information displayed in the edit dialog may be outdated. The revert action can also be generated by pressing the Escape key in the body of the editor.

`Cancel` The Cancel button dismisses the dialog without applying the user's changes.

`Other Member Buttons`
Some objects will add an additional row of member buttons to the Edit Dialog.(e.g., The Process object adds a row of Run,Step,Init buttons). These buttons perform additional actions on the object.

6.5.3 Edit Dialog Menus

At the top of the edit dialog is a menubar with at least two menus. The first menu entitled Object is similar to the Object menu on a WinBase window. If the edited object is not a WinBase, the "Object" menu my only provide the "Load", "Save, "Save As", and "Close" menu choices. (see Section 6.2 [gui-object], page 55). The second menu called "Actions" may contain member functions particular to the edited object's type which will be called on the edited object.

6.6 Settings Affecting GUI Behavior

There are two primary ways of setting parameters which affect the look of the GUI. There is an object called **taMisc** which contains miscellaneous parameters and settings that affect various aspects of the PDP++ system, but mostly the graphical interface. This object can be edited by using the *Settings* menu option on the **PDPRoot** object.

In addition to the taMisc settings, there are a number of XWindow resources or Xdefaults that can be set. Also, since PDP++ uses the InterViews graphics toolkit a number of command line arguments and InterViews Xresources can also be customized (see Section 9.3 [proj-startup], page 124).

6.6.1 Settings in taMisc

The following parameters can be set:

`int display_width`
: Width of the shell display in characters.

`int sep_tabs`
: Number of tabs to separate items by in listings.

`int max_menu`
: The maximum number of elements in a menu — if there are more than this number of instances (tokens) of a given type of object, then the menu for that type of object will say "<Over max, Select>", meaning that each item could not be listed separately in the menu, so you select this option to pull up a object chooser (see Section 6.9 [gui-obj-chooser], page 70), which allows you to choose the object from a longer list.

`int search_depth`
: The recursive depth at which css stops searching for an object's path.

`int color_scale_size`
: This determines how many colors are in a color scale for a color monitor (see Section 6.7 [gui-colors], page 67).

`int mono_scale_size`
: This determines how many colors are in a color scale for a monochrome monitor (see Section 6.7 [gui-colors], page 67).

`ShowMembs show`
: Type of members to show in css.

`ShowMembs show_iv`
: Type of members to show in edit dialogs The previous two variables can have one of four "ShowMembs" values:

  `ALL_MEMBS`
  : Shows all members.

  `NO_READ_ONLY`
  : Shows all but read_only members – stuff that can't be changed.

  `NO_HIDDEN`
  : Shows all but hidden members – stuff that is hidden because it is not typically relevant.

  `NO_HID_RO`
  : Shows all but hidden and r/o members.

  `NO_DETAIL`
  : Shows all but "detail" members – stuff that might be relevant, but is not often accessed.

  `NO_HID_DET, etc.`
  : Further combinations of RO, HID, and DET options.

  The default is `NO_HID_RO_DET`.

`TypeInfo type_info`
: The amount of information about a class type that is reported when the "type" command is used in CSS. Type_info has one of the following values:

  `MEMB_OFFSETS`
  : Shows the byte offset of members.

  `All_INFO` Shows all type info except memb_offsets

  `NO_OPTIONS`
  : Shows all info except type options

  `NO_LISTS` Shows all info but lists

  `NO_OPTIONS_LISTS`
  : Shows all info but options and lists

  The default is `NO_OPTIONS_LISTS`

`KeepTokens keep_tok`
: This sets the Default for keeping tokens (lists of members) for object types. This can have one of three "KeepTokens" values:

  `Tokens` Keep tokens as specified in the type.

  `NoTokens` Do not keep any tokens lists.

  `ForceTokens`
  : Force the keeping of all tokens lists.

  The default is `Tokens`.

`verbose_load`
: Indicates the amount of information reported in the shell window when objects are loaded

`iv_verbose_load`
: Indicate the amount of information reported in the Loading window when objects are loaded The previous two members may have one of the following values:

  `QUIET` No information is given

  `MESSAGES` General messages are reported for each object

  `TRACE` Each line of the file is printed as it is loaded

  `SOURCE` Intricate debugging details are reported

`bool auto_edit`
: If this value is on, an edit dialog will appear each time an object is created with the New dialog. (see Section 6.4 [The Subgroup Menu(New)], page 57).

`AutoRevert auto_revert`
: In some cases a dialog may be reverted to its old values by a PDP++ process. This variable controls the behavior of the edit dialog when this situation occurs. The behavior is determined by one of three values:

  `Auto_Apply`
  : Automatically apply changes before auto-reverting

  `Auto_Revert`
  : Automatically revert the edit dialog loosing changes

  `Confirm_Revert`
  : Popup a confirmation dialog to okay reverting

`String include_paths`
: Directory Paths used for finding files. This is an array of string values that represent paths to find CSS, defaults files, and Help files in (see Section 7.4.8 [css-settings], page 103, see Section 9.5.1 [proj-settings], page 126, see Section 9.5.2 [proj-objdef], page 126).

`String tmp_dir`
: The directory to use for temporary files

`String compress_cmd`
: Command to use for compressing files

`String uncompress_cmd`
: Command to use for uncompressing files

`String compress_sfx`
: Suffix to append to filenames when compressed

`String help_file_tmplt`
: Template for converting the type name of an object into a help file – %t is replaced with the type name. Include any leading paths to help files relative to the basic root paths listed on include_paths.

`String help_cmd`
: Comand for bringing up a help browser to read help file (typically html, netscape by default). %s is subsituted with the help file produced from `help_file_tmplt`.

6.6.2 XWindow Resources (Xdefaults)

The following XWindow resources can be set. These are typically placed in the user's '`.Xdefaults`' file in their home directory.

`PDP+++flat`
: The color of the buttons. It is typically some kind of greyish color. The default has a blueish tint: `#c0c4d3`

`PDP+++background`
: The color of various non-view background regions. The default is an aquamarine color: `#70c0d8`.

`PDP+++name*flat`
: If you change the valueof background, you should change this one to match. It makes the member names in the edit dialogs (see Section 6.5 [gui-edit], page 58) the same color as the background.

`PDP+++apply_button*flat`
: The color of the apply button (and other buttons) when they are the "suggested" choice. It is a dusty-red by default: `#c090b0`.

`PDP+++FieldEditor*background`
: The color of the edit fields, default is white.

`PDP+++font`
: The generic font to use for buttons, etc: `*-helvetica-medium-r-*-*-10*`.

`PDP+++name*font`
: The font to use for member names: `*-helvetica-medium-r-*-*-10*`.

`PDP+++title*font`
: The font to use for titles of edit dialogs: `*-helvetica-bold-r-*-*-10*`.

`PDP+++small_menu*font`
: The font to use for small menus (e.g., those in edit dialogs): `*-helvetica-medium-r-*-10*`.

`PDP+++small_submenu*font`
: This font is used for menu items that contain sub-menus (these are also indicated by three dots after the name) `*-helvetica-medium-r-*-*-10*`.

`PDP+++big_menu*font`
: This font is used for big menus like those on the permanent object windows, default is `*-helvetica-medium-r-*-12*`.

`PDP+++big_submenu*font`
: For sub menus on big menus: `*-helvetica-medium-r-*-*-12*`.

`PDP++*big_menubar*font`
: For big menus, the name of the menu itself: `*-helvetica-bold-r-*-*-14*`.

`PDP++*big_italic_menubar*font`
: For big menus, name of italicized menu items (i.e. those that apply to the object and not to its sub-objects): `*-helvetica-bold-o-*-*-14*`.

`PDP++*double_buffered:`
: Double buffering makes the display smoother, but can use lots of display memory. Default is "on".

`PDP++*FileChooser.width`
: The width of the file-chooser in pixels. Default is 100.

`PDP++*FileChooser.rows`
: The number of items to list in the file chooser, default is 20.

`PDP++*clickDelay`
: This is the number of milliseconds to count two clicks as a double-click. The default is 250.

For MS Windows Users:

To set XResources under MS Windows, you need to create an "application defaults" file that is formatted much like the XResources. There are two steps for creating this application defaults file:

1. Edit '`C:\WINDOWS\WIN.INI`', and add the following two lines:

```
[InterViews]
location = C:\PDP++
```

Where the location should be the actual location where you installed the software.

2. Create a sub-directory under '`C:\PDP++`' (again, use the actual location) called '`app-defaults`', and then create a file called '`InterViews`' in that directory. This file should contain resource values you want to set. For example, to change the amount of time to detect a double-click, you would enter:

```
*clickDelay: 400
```

To change the overall size of the PDP++ windows (scaling):

```
*mswin_scale: 1.25
```

6.7 Color Scale Specifications

Color scales are used in PDP++ to display the values of variables graphically in various types of displays. The choice of color scale depends on personal preferences as well as what type of display the user has available. There are a number of different types of color scales that come with the software, and the user can create their own custom colorscales.

A color scale is specified by creating a set of different color points. The actual scale is just the linear interpolation between each of these points, where the points are distributed evenly through the range of values covered by the scale. Thus, if there were three such

points in a color specification that goes from -1 to 1, the first point would represent the value -1, the middle one would represent 0, and the last one would represent 1. Values in between would be represented by intermediate colors between these points. The actual number of colors created in a given color scale is determined by the `color_scale_size` setting parameter for color displays, and `mono_scale_size` for monochrome displays (see Section 6.6 [gui-settings], page 63).

The **PDPRoot** object contains a group of color specifications called `.colorspecs`, which is where the default color scales and any new ones the user creates are located. The default element (see Section 8.2 [obj-group], page 109) of this group represents the default color scale to use when creating a new display that uses color scales. The defaults are as follows:

`C_ColdHot`
: interpolates from violet->blue->grey>->red->yellow

`C_BlueBlackRed`
: interpolates from blue->black-red

`C_BlueGreyRed`
: interpolates from blue->grey->red

`C_BlueWhiteRed`
: interpolates from blue->white->red

`C_BlueGreenRed`
: interpolates from blue->green->red

`C_Rainbow`
: interpolates from violet->blue->green->yellow->red

`C_ROYGBIV`
: interpolates from violet->indigo->blue->green->yellow->red

`C_DarkLight`
: interpolates from black->white

`C_LightDark`
: interpolates from white->black

`M_DarkLight`
: dithers from black->white

`M_LightDark`
: dithers from white->black

`M_LightDarkLight`
: dithers from white->black->white

`P_DarkLight`
: dithers from black->white with a white background for printing

`P_DarkLight_bright`
: dithers from black->white with a white background for printing, having a brighter overall tone than the basic one (a lighter zero value).

`P_LightDark`
: dithers from white->black with a white background for printing

`P_DarkLightDark`
: same as M_DarkLightDark with a white background for printing

`P_LightDarkLight`
: same as M_LightDarkLight with a white background for printing

The **ColorScaleSpec** is the object that specifies the color scale. It contains a group of RGBA objects, each of which is used to specify a point on the colorscale range based on the Red, Green, and Blue values, plus a "transparency" parameter Alpha. The ColorScaleSpec object has one primary function.

`GenRanges (ColorGroup* cl, int nper)`
: This function creates a range of colors in the ColorGroup by linearly interpolating nper colors for each RGBA set point value in the ColorScaleSpec.

The **RGBA** object has the following fields:

`String name`
: The name of the color

`float r` Amount of red in the color (0.0 - 1.0)

`float g` Amount of green in the color (0.0 - 1.0)

`float b` Amount of blue in the color (0.0 - 1.0)

`float a` Alpha intensity value (ratio of foreground to background) This is used primarily for monochrome displays.

If the name field of an RGBA object is set, then it will try to lookup the name to find the r,g, and b values for that color.

Thus, to create your own color scale specification, just create a new ColorScaleSpec object, and then create some number of RGBA objects in it. Then, edit your views (e.g., the network view, Section 10.6 [net-view], page 149), and set their `colorspec` to point to your new specification. In order to see changes you make to your color spec, you need to switch the `colorspec` pointer to a different one and then back to yours after making the changes.

6.8 File Requester

The File Requester Dialog is used for choosing filename for reading and writing to files. The directions for the file manipulation are listed at the top of the dialog. Typically the directions will ask the user to "Select a File for xxx" where xxx is one of "Opening", "Appending", "Writing", etc. Under the directions is the prompt "Enter Filename", and a FieldEditor where the user can type in the name of the file.

Below the FieldEditor is a vertical scrollbox of filenames in the current directory. If there are more names than will fit in the window, the scrollbar on the right edge of the scrollbox can be used to scroll through the full listing. A filename can be chosen by clicking on the name in the scrollbox. Subdirectories are listed with a slash ('/') following the name, and can be read in by double-clicking on their name. The "../" directory can be used to navigate up a directory level.

Below the filename scrollbox is a Field Editor for the filename filter. The String listed here is a unix csh filename completion string. Wildcards can be specified using the '*' character. See the csh man page for more details. Typically this field will be set by PDP++ to limit the range of filenames available to those which correspond to the type of file the dialog is to act upon. (e.g., If projects are being loaded, then the filter will be set to "*.proj.*" to limit the selection to files with ".proj" in the filename.) In most cases, compressed files can be loaded and saved as well.

Below the filter Field Editor are the action buttons. The leftmost button will perform the action specified in the directions at the top of the dialog. The rightmost button is the "Cancel" button and will exit the dialog without performing the action. Double-clicking on a filename, or pressing the "return" key, also causes the dialog's action to be taken.

6.9 Object Chooser

The Object Chooser Dialog is much like the file requester described previously, and is used primarily for choosing objects when there are too many such objects to fit within a menu. It can also be used to browse the entire hierarchy of objects. In the simple choosing function, just scroll and pick the object – double clicking to select, or using the Select button at the bottom. A name can also be typed into the field at the top of the browser.

By selecting *PDP++Root/Browse*, the object chooser acts more like a file browser. When you select an object, the default is to descend into the sub-objects on that object. Pressing .. will go back up a level (as in the file system). When you select an object, an edit dialog will show up for that object. This can be used for editing things that may not show up in a useful way in standard edit dialogs (e.g., the StatVal's in Stats, see Section 12.5 [proc-stat], page 199).

7 Guide to the Script Language (CSS)

CSS is the script language for the PDP++ software. It is a general-purpose language that allows one to do virtually anything that could be done by writing programs in C or C++ and compiling them. It provides full access to all the types and objects defined in PDP++, and allows you to call the "member functions" of these objects and set the values of their members variables.

There are two principal uses for the script language in running simulations. One is to automate the procedure of setting up and running a simulation. In this role, the script replaces commands that would otherwise be given through the graphical user interface (gui). The other is to extend the software without having to re-compile and re-link. Thus, one can write new kinds of procedures, statistics, etc. that are particular to a given simulation in the script language, and run them as if they were hard coded. This gives the PDP++ software more flexibility.

7.1 Introduction to CSS

CSS is a "C" language interpreter and script language (C Super-Script). The name derives from the fact that the language is C, written in C, so it is C to the C, or C^c, while at the same time being an excellent, even superlative scripting language. The actual syntax for CSS is somewhere in between C and C++, and the language is written entirely in C++. The major differences between CSS and C++ are listed in Section 7.3.1 [css-c++-diff], page 84, and the few discrepancies between CSS and C are noted in Section 7.3.2 [css-c-diff], page 84. Some of the convenient features of C++ that are incorporated into CSS are enumerated in Section 7.3.4 [css-c++-intro], page 86, for those unfamiliar with C++.

Being an interpreter, CSS presents the user with a prompt: `css>`. At this prompt, the user can enter any C expression, which will be evaluated immediately upon pressing Return, or enter a command. The commands operate much like a debugger (e.g., gdb, dbx), and allow control over the running, listing, and debugging of programs. In addition to the default "immediate mode" behavior of the system, it can behave like a compiler. For example, the command `define` will cause the system to enter a compiling mode (the prompt changes to `css\#`), where the C code that is subsequently entered is compiled into an intermediate machine code, but not run directly. Instead, it becomes a stored program that can be run many times.

Typically, one works with a program file (use the extension .css to indicate a css file) that is loaded and then run, instead of manually typing programs into the system using define mode. The command `load "filename.css"` will load a file and compile it, at which point it can be run. The command `run` will run the program from the beginning. `reload` will remove the existing compiled program and load (and re-compile) the previously loaded one from the disk file, which is handy when you are editing the program file and testing it out.

One can view the source code within CSS by using the `list` command, which takes (optional) line number and length arguments. Unlike C functions, which need to be called with the usual syntax of parentheses around the arguments, which are themselves comma

separated, and the whole thing is terminated by a semi-colon, the arguments to commands don't require the parentheses or the semi-colon, but do require thte commas.

One of the principle uses of CSS is as an interface for hard-coded C/C++ programs. There is a "TypeAccess" program that reads the header files for a given application and generates type information that can be used by CSS to automatically interface with the objects and types in the hard-coded world. CSS comes initially with some relevant hard-coded types from the standard C/C++ library, including a String class and the stream I/O classes, and various math and posix library functions.

7.2 Tutorial Example of Using CSS

This section provides a tutorial-style introduction to some of the features of the CSS language environment.

7.2.1 Running an Example Program in CSS

The following example will be used to illustrate the use of the CSS system:

```
// a function that decodes the number
String Decode_Number(float number) {
  String rval = "The number, " + number + " is: ";
  if(number < 0)
    rval += "negative";
  else if(number < .5)
    rval += "less than .5";
  else if(number < 1)
    rval += "less than 1";
  else
    rval += "greater than 1";
  return rval;
}

// decodes n_numbers randomly generated numbers
void Decode_Random(int n_numbers) {
  int i;
  for(i=0; i<n_numbers; i++) {
    float number = 4.0 * (drand48()-.5);
    String decode = Decode_Number(number);  // call our decoding function
    cout << i << "\t" << decode << "\n";    // c++ style output
  }
}
```

You can enter this code using a text editor (e.g., emacs or vi), or use the example code in '`css_example.css`' included with the PDP++ software in the '`css/include`' directory. Then, run the PDP++ software. At the prompt (which will vary depending on which executable you run — the example will use `css>`), type:

```
css> load "css_example.css"
```

and the prompt should return. If you made any typos, they might show up as a Syntax Error, and should be corrected in the text file, which can be re-loaded with:

```
css> reload
```

Loading the text file translates it into an internal "machine code" that actually implements CSS. This is like compiling the code in traditional C, but it does not write out an executable file. Instead, it keeps the machine code in memory, so that it can be run interactively by the user.

To ensure that CSS has loaded the text, you can list the program:

```
css> list

Listing of Program: css_example.css
1
2       // a function that decodes the number
3       String Decode_Number(float number) {
4         String rval = "The number, " + number + " is: ";
5         if(number < 0)
6           rval += "negative";
7         else if(number < .5)
8           rval += "less than .5";
9         else if(number < 1)
10          rval += "less than 1";
11        else
12          rval += "greater than 1";
13        return rval;
14      }
15
16      // decodes n_numbers randomly generated numbers
17      void Decode_Random(int n_numbers) {
18        int i;
19        for(i=0; i<n_numbers; i++) {
20          float number = 4.0 * (drand48()-.5);
21          String decode = Decode_Number(number);  // call decoder
css> list

Listing of Program: css_example.css
21          String decode = Decode_Number(number);  // call decoder
22          cout << i << "\t" << decode << "\n";    // c++ style output
23        }
24      }
26      list
```

Note that you have to type `list` in twice in order to see the whole program (by default, `list` only shows 20 lines of code). Also, notice that the list command itself shows up at the end of the program—this is because commands that are entered on the command line are actually compiled, run, and then deleted from the end of the program. Because `list`, when run, shows the current state of the code (i.e., before it has itself been deleted), it shows up at the bottom of the listing.

Now, let's try running the example:

```
css> run
css>
```

Nothing happens! This is because the code as written only defines functions, it does not actually call them. The program would have to have had some statements (i.e., function calls, etc) at the *top level* (i.e., not within the definition of a function) in order to do something when the `run` command is issued.

However, we can call the functions directly:

```
css> Decode_Random(5);
0       The number, -0.414141 is: negative
1       The number, 1.36194 is: greater than 1
2       The number, -0.586656 is: negative
3       The number, -0.213666 is: negative
4       The number, -0.725229 is: negative
css>
```

(of course, your output will vary depending on the random number generator). This illustrates the interactive nature of CSS — you can call any function with any argument, and it will execute it for you right then and there. This is especially useful for debugging functions individually. Thus, we can call the `Decode_Number` function directly with different numbers to make sure it handles the cases appropriately:

```
css> Decode_Number(.25);
css>
```

Notice, however, that there was no output. This is because the function simply returns a string, but does not print it out. There are several ways to print out results, but the easiest is probably to use the `print` command:

```
css> print Decode_Number(.25);
(String)  = The number, 0.25 is: less than .5
```

`print` gives you the type name, the variable name (which is blank in this case), and the value. To illustrate this, you can just declare a variable directly:

```
css> String foo = "a string";
css> print foo
(String) foo = a string
```

Compare this with the print *function* (not command) `printf`:

```
css> printf(Decode_Number(.25));
The number, 0.25 is: less than .5css>
```

which just gives you the value of the string (and does not automatically append a '\n' return at the end of the line). Finally, we can use C++ stream-based printing, which directs the return value from the function to print itself to the default current output `cout`:

```
css> cout << Decode_Number(.75) << "\n";
The number, 0.75 is: less than 1
css>
```

Note also that you can put any expression in the arguments to the function:

```
css> print Decode_Number(exp(cos(tan(.2) + .5) * PI)/ 20);
```

```
(String)  = The number, 0.549689 is: less than 1
```

In order to actually be able to run a script, we can add the following lines to the code by switching to `define` mode instead of the default interactive mode:

```
css> define
css# cout << Decode_Number(.75) << "\n";
css# Decode_Random(5);
css# exit
```

Note that we use `exit` to exit the `define` mode. You could also use the `EOF` character (*`ctrl-D`* on most systems) to exit this mode. To see that we have added to the program, list it from line 20 onwards:

```
css> list 20

Listing of Program: css_example.css
20          float number = 4.0 * (drand48()-.5);
21          String decode = Decode_Number(number);  // call decoder
22          cout << i << "\t" << decode << "\n";    // c++ style output
23        }
24      }
26      cout << Decode_Number(.75) << "\n";
27      Decode_Random(5);
28      exit
29      list 20
```

Now, when we `run` the program, we get:

```
css> run
The number, 0.75 is: less than 1
0       The number, 1.54571 is: greater than 1
1       The number, -1.93767 is: negative
2       The number, 0.336361 is: less than .5
3       The number, -1.36253 is: negative
4       The number, -0.465137 is: negative
css>
```

All of the C language rules about declaring or defining functions before they are called apply in CSS as well (in general, CSS obeys most of the same rules as C), so we could not have put those two extra lines of code in the example program before the functions themselves were defined. In general, this leads to a program layout consisting of various different functions, followed at the very end by one or two lines of code at the top-level which call the relevant function to start things off. If you are feeling traditional, you can call this function `main`, and it will look like a regular C/C++ program, except for the last line which calls the main function.

7.2.2 Debugging an Example Program in CSS

This section illustrates the use of the debugging facilities within CSS. We will use the example program from the previous section, with the addition of the two extra lines of code that were added in `define` mode. It is important to note that some important aspects of debugging like breakpoints can only be used when a program was started with the `run`

command. Thus, do not set breakpoints if you are going to be simply calling a function on the command line.

There are three primary debugging tools in CSS: **breakpoints**, **execution tracing**, and **single-stepping**. Because CSS is basically an interpreted system, all of the facilities for examining variables, the stack, source code, etc. which must be added into a real debugger come for "free" in CSS.

To illustrate, we will set a breakpoint in the `Decode_Number` function (assuming you have loaded this code already):

```
css> list Decode_Number
(String) Decode_Number((Real) number) {
4         String rval = "The number, " + number + " is: ";
5         if(number < 0)
6           rval += "negative";
7         else if(number < .5)
8           rval += "less than .5";
9         else if(number < 1)
10          rval += "less than 1";
11        else
12          rval += "greater than 1";
13        return rval;
14      }
css> setbp 5
css> showbp
Decode_Number   5         if(number < 0)
css>
```

Note that we first listed the function (referring to it by name), and then set a breakpoint with the `setbp` command at line number 5. We then confirmed this breakpoint with the `showbp` command.

Now, when we `run` the program, execution will stop at line 5, and we will be returned to the `css>` prompt:

```
css> run
2 css>
```

which has changed to `2 css>`, indicating that we are 2 levels deep into the execution of the program. To see where we are, `list` and `status` can be used:

```
2 css> list

Listing of Program: css_example.css
4         String rval = "The number, " + number + " is: ";
5         if(number < 0)
6           rval += "negative";
7         else if(number < .5)
8           rval += "less than .5";
9         else if(number < 1)
10          rval += "less than 1";
11        else
12          rval += "greater than 1";
```

```
13        return rval;
14      }
28      list
2 css> status

        Status of Program: css_example.css
curnt:  Decode_Number   src_ln: 5       pc:     9
debug:  0       step:   0       depth:  2       conts:  2
lines:  29      list:   28
State: shell:   1   run: 0  cont:   0  defn:   0  runlast: 0
run status:     Waiting
shell cmd:      None
2 css>
```

The `src_ln` of the status output tells us which source-code line we are at (and what function we are in). Here, we can use the interactive mode of css to determine the values of our variables:

```
2 css> print number
(Real) number = 0.75
2 css> print rval
(String) rval = The number, 0.75 is:
```

The values of all of the variables for the current "frame" (there is one frame for every call to a given function, or any expression appearing between the curly-brackets) can be viewed at once with the `frame` command:

```
2 css> frame

Elements of Spaces For Program: Decode_Number (frame = 1)

Elements of Space: Autos (3)
(String) _retv_this =                   (Real) number = 0.75
(String) rval = The number, 0.75 is:

Elements of Space: Stack (0)


Elements of Space: css_example.css.Static (3)
(String) Decode_Number((Real) number)   (void) Decode_Random((Int) n_numbers)
(String) foo = a string
```

Included in the frame information are the "automatic" variables (`Autos`), and the contents of the stack. To get information on previous frames in the execution sequence, use the command `trace`, which gives both a trace of processing and can give a dump of the entire stack up to this point if given a higher "trace level", which is an optional argument to the `trace` command. The default trace level of 0 just shows the frames, 1 also shows the stack, 2 also shows the auto variables, and 3 shows all variables. In addition, just the contents of the current stack can be viewed with the `stack` command (note that the saved stack is what is actually used by the program during execution).

In addition to viewing variables, it is possible to change their values:

```
2 css> number = 200;
2 css> print number
(Real) number = 200
```

Finally, to continue the program from where it was stopped by the breakpoint, use the `cont` command:

```
2 css> cont
The number, 0.75 is: greater than 1
5 css>
```

Since we changed the number after it was turned into a string for the `rval`, but before the `if` statements, we got the contradictory result printed above. Also, because the breakpoint is still in effect, the program has stopped due to the `Decode_Random` function calling the `Decode_Number` function. We can continue again, or we can disable the breakpoint first, and then continue.

```
5 css> cont
0       The number, 0.764017 is: less than 1
5 css> unsetbp 5
5 css> showbp
5 css> cont
1       The number, -1.76456 is: negative
2       The number, 1.59942 is: greater than 1
3       The number, -1.34582 is: negative
4       The number, -1.36371 is: negative
css>
```

Note that the breakpoint is referred to by the source-code line number.

Another way to debug is to get a trace of the running program. This can be done by setting the `debug` level, which can have a value from 0 (normal, quiet operation) through 4, with higher levels giving more detail than lower levels. For most users, levels 0 and 1 are the only ones needed, since the higher levels depend on understanding the guts of the CSS machine code. Debug level 1 shows the source-code line corresponding to the currently-executing code:

```
css> debug 1
css> run
31      run
3       String Decode_Number(float number) {
17      void Decode_Random(int n_numbers) {
26      cout << Decode_Number(.75) << "\n";
4         String rval = "The number, " + number + " is: ";
5         if(number < 0)
5         if(number < 0)
5         if(number < 0)
7         else if(number < .5)
7         else if(number < .5)
7         else if(number < .5)
9         else if(number < 1)
9         else if(number < 1)
9         else if(number < 1)
```

```
10          rval += "less than 1";
13        return rval;
26      cout << Decode_Number(.75) << "\n";
The number, 0.75 is: less than 1
27      Decode_Random(5);
18        int i;
19        for(i=0; i<n_numbers; i++) {
.
.
.
```

Since running proceeds from the top to the bottom, the definitions of the functions appear in the trace even though they do not really do anything. Also, some constructs like `if` and `for` result in multiple copies of the same source-code line being printed. This kind of trace is useful for seeing what branches of the code are being taken, etc.

The final kind of debugging is single-stepping, which is like having a breakpoint after every line of source code. Execution continues after each point by simply entering a blank line:

```
css> debug 0
31      debug 0
css> step 1
css> run
31      run
3       String Decode_Number(float number) {
1 css>
17      void Decode_Random(int n_numbers) {
1 css>
26      cout << Decode_Number(.75) << "\n";
4         String rval = "The number, " + number + " is: ";
2 css> print number
31      print number
(Real) number = 0.75
2 css> print rval
(String) rval = The number, 0.75 is:
5         if(number < 0)
3 css>
7         else if(number < .5)
4 css>
9         else if(number < 1)
5 css>
10          rval += "less than 1";
4 css>
13        return rval;
26      cout << Decode_Number(.75) << "\n";
The number, 0.75 is: less than 1
1 css>
27      Decode_Random(5);
18        int i;
2 css>
```

```
19        for(i=0; i<n_numbers; i++) {
19        for(i=0; i<n_numbers; i++) {
20          float number = 4.0 * (drand48()-.5);
4 css>
```

(note that we turned debugging off, since it is redundant with single-stepping). The `step` command takes an argument, which is the number of lines to step over (typically 1). Then, when we `run` the program again, it stops after every line. If you simply want to continue running, you can just hit return and it will continue to the next line.

If at any point during debugging you want to stop the execution of the program and return to the top-level (i.e., get rid of that number in front of the prompt), use the `restart` command.

```
4 css> restart
34      restart
css>
```

Be sure not to confuse `restart` with `reset`, as the latter will erase the current program from memory (you can just reload it with `reload`, so its not so bad if you do).

7.2.3 Accessing Hard-Coded Objects in CSS

In general, hard-coded objects (and their members and member functions) are treated just as they would be in C or C++. Objects that have been created (i.e., Networks, Units, etc) can be referred to by their *path* names, which start with the *root* object (PDPRoot). While you could type `root.projects[0]`, for example, to refer to the first project, it is easier to use the abbreviation of a preceding period to stand for `root`, resulting in: `.projects[0]`.

The following examples were performed on the XOR example project in Bp++. In order to examine the project in CSS, one could simply use the `print` command on the path of this object:

```
bp++> print .projects[0]
.projects[0] Proj (refn=1) {
  ta_Base*        owner           = .projects;
  String          name            = Proj;
  WinBase*        win_owner       = root;
  WinGeometry     win_pos         = {lft=4: bot=74: wd=535: ht=24: };
  WinGeometry     root_win_pos    = {lft=9: bot=79: wd=161: ht=23: };
  TypeDefault_MGroup   defaults          =  Size: 5 (TypeDefault);
  BaseSpec_MGroup   specs         =  Size: 3 (BaseSpec);
  Network_MGroup   networks       =  Size: 1 (Network);
  Environment_MGroup   environments       =  Size: 1 (Environment);
  Process_MGroup   processes      =  Size: 5 (SchedProcess);
  PDPLog_MGroup   logs            =  Size: 2 (TextLog);
  Script_MGroup   scripts         =  Size: 1 (Script);
}
```

The first network within this project would then be referred to as `.projects[0].networks[0]`:

```
bp++> print .projects[0].networks[0]
.projects[0].networks[0] XOR (refn=15) {
```

```
  ta_Base*        owner           = .projects[0].networks;
  String          name            = XOR;
  WinBase*        win_owner       = .projects[0];
  WinGeometry     win_pos         = {lft=4: bot=3: wd=536: ht=390: };
  WinView_MGroup   views          =  Size: 1 (NetView);
  Layer_MGroup    layers          =  Size: 3 (Layer);
  Project*        proj            = .projects[0];
  TDGeometry      pos             = {x=0: y=0: z=0: };
  TDGeometry      max_size        = {x=2: y=2: z=3: };
  int             epoch           = 0;
  Network::Layer_Layout   lay_layout      = THREE_D;
}
```

You can also use a shortcut by just typing `.networks[0]`, which finds the first member with the name `networks` in a search starting at the root object and scanning down the *first branch* of the tree of objects (i.e., looking in the first element of every group along the way).

Scoped types such as `Network::Layer_Layout` which appear in the above class are referred to just as they would be in C++:

```
bp++> .networks[0].lay_layout = Network::TWO_D;
```

As you can see, setting the values of hard-coded object variables simply amounts to typing in the appropriate C/C++ statement.

Type information (obtained via the TypeAccess system) about hard-coded objects can be obtained with the `type` command:

```
bp++> type Network
class Network : PDPWinMgr : WinMgr : WinBase : ta_NBase : ta_Base {
// The Network

  // sub-types
  enum Layer_Layout {   // Visual mode of layer position/view
    TWO_D            = 0;       // all z = 0, no skew
    THREE_D          = 1;       // z = layer index, default skew
  }

  // members
  ta_Base*        owner;          //   pointer to owner
  String          name;           //   name of the object
  .
  .
  TDGeometry      max_size;       //   max size in each dim
  int             epoch;          //   epoch counter
  Network::Layer_Layout   lay_layout;     // Visual mode of layer

  // functions
  void            UnSafeCopy(ta_Base* na);
  ta_Base*        GetOwner(TypeDef* tp);
  .
  .
  .
```

```
  void            InitWtState();        // Initialize the weights
  .
  .
  void            Compute_dWt();        // update weights for whole net
  void            Copy_Weights(const Network* src);
  void            Enforce_Layout(Network::Layer_Layout layout_type);
}
```

This shows the inheritance of this object, any sub-types that are defined within it, and all of its members and functions (including those it inherits from other classes).

In addition, there is Tab-completion for path names and types in the CSS prompt-level script interface. Thus, as you are typing a path, if you hit the Tab key, it will try to complete the path. If there are multiple completions, hitting Tab twice will display them.

In order to call member functions of hard-coded classes, simply give the path to the object, followed by the member function, with any arguments that it might require (or none).

```
bp++> .networks[0].InitWtState();
bp++>
```

It is possible to create pointers to hard-coded objects. Simply declare a pointer variable with the appropriate type, and assign it to the given object by referring to its path:

```
bp++> Unit* un;
bp++> un = .networks[0].layers[1].units[0];
bp++> print un
.projects[0].networks[0].layers[1].units[0] hid_1 (refn=6) {
  ta_Base*        owner           = .projects[0].networks[0].layers[1].units;
  String          name            = hid_1;
  UnitSpec_SPtr   spec            = {type=BpUnitSpec: spec=.specs[0]: };
  TDGeometry      pos             = {x=0: y=0: z=0: };
  Unit::ExtType   ext_flag        = NO_EXTERNAL;
  float           targ            = 0;
  float           ext             = 0;
  float           act             = 0;
  float           net             = 0;
  Con_Group       recv            =  Size: 0.1.2 (BpCon);
  Con_Group       send            =  Size: 0.1.1 (BpCon);
  BpCon           bias            = BpCon;
  float           err             = 0;
  float           dEdA            = 0;
  float           dEdNet          = 0;
}
```

There are two ways to create new hard-coded objects. The preferred way is to call one of the `New` functions on the group-like objects (`List` or `Group`, see Section 8.2 [obj-group], page 109), which will create the object and add it to the group, so that it can be referred to by its path as just described.

```
bp++> .layers[1].units.List();

Elements of List:  (2)
```

```
hid_1   hid_2
bp++> .layers[1].units.New(1);
bp++> .layers[1].units.List();

Elements of List:  (3)
hid_1   hid_2
bp++> .layers[1].units[2].name = "new_guy";
bp++> .layers[1].units.List();

Elements of List:  (3)
hid_1   hid_2   new_guy
```

Finally, it is possible to create new instances of hard-coded object types through the C++ `new` operator, which is especially useful in order to take advantage of some of the handy built-in types like arrays (see Section 8.3 [obj-array], page 114):

```
bp++> float_RArray* ar = new float_RArray;
bp++> print ar
[0];

bp++> ar.Add(.25);
bp++> ar.Add(.55);
bp++> ar.Add(.11);
bp++> print ar
[3] 0.25 0.55 0.11;

bp++> print ar.Mean();
(Real)  = 0.303333
bp++> print ar.Var();
(Real)  = 0.101067
bp++> ar.Sort();
bp++> print ar
[3] 0.11 0.25 0.55;
```

Remember to `delete` those objects which you have created in this fashion:

```
bp++> delete ar;
bp++> print ar
(float_RArray) ar = 0
```

(the `ar = 0` means that it is a null pointer). Be sure *not* to use the `delete` operator on those objects which were created with the group's `New` function, which should be `Removed` from the group, not deleted directly (see Section 8.2 [obj-group], page 109).

7.3 CSS For C/C++ Programmers

This section outlines the ways in which CSS differs from C and C++. These differences have been kept as small as possible, but nonetheless the fact that CSS is an interactive scripting language means that it will inevitably differ from compiled versions of the language in some respects.

Note that this manual does not contain an exhaustive description of the CSS language — only differences from standard C/C++. Thus, it is expected that the user who is unfamiliar with these languages will purchase one of the hundreds of books on these languages, and then consult this section to find out where CSS differs.

7.3.1 Differences Between CSS and C++

The primary difference between CSS and C++ is that CSS does not support the complex set of rules which determine which of a set of functions with the same name should be called given a particular set of arguments. Thus, CSS does not allow multiple functions with the same name. By avoiding the function-call resolution problems, CSS is much faster and smaller than it otherwise would be.

One consequence of the lack of name resolution is that only default (no argument) constructors can be defined. This also obviates the need for the special parent-constructor calling syntax.

Also, at the present time, CSS does not support the definition of `operator` member functions that redefine the operation of the various arithmetic operators. Further, it does not provide access control via the `private`, `public`, and `protected` keywords, or the `const` type control, though these are parsed (and ignored). Thus, it also does not deal with the `friend` constructs either. While these language features could be added, they do not make a great deal of sense for the interactive script-level programming that CSS is designed to handle.

CSS *does* support multiple inheritance, and the overloading of derived member functions. It does not support the inlining of functions, which is a compiler-level optimization anyway. The keyword `inline` will be parsed, but ignored.

Also, note that CSS gives one access to hard-coded classes and types (via TypeAccess), in addition to those defined in the script.

Templates are not supported for script-based classes, but are supported in hard-coded classes (via TypeAccess).

Exception handling is not supported, and probably will not be.

Since the primary use of CSS is for relatively simple pieces of code that glue together more complex hard-coded objects, and not implementing large programs, the present limitations of CSS will probably not affect most users.

7.3.2 Differences Between CSS and ANSI C

The design specification for CSS should be the same as ANSI C. However, at the present time, some features have not been implemented, including:

The full functionality of `#define` pre-processor macros is not supported. At this point, things are either defined or undefined and the `#ifdef` and `#ifndef` functions can be used to selectively include or not different parts of code.

The promotion of types in arithmetic expressions is not standard. In CSS, the result of an expression is determined by the left-most (first) value in the expression. Thus, `20 / 3.0` would result in an integer value of 6, whereas `20.0 / 3` gives a real value of 6.6667.

The guarantee that only as much of a logical expression will be evaluated as is necessary is not implemented in CSS. Thus, something like '`if((a == NULL) || (a->memb == "whatever"))`' will (unfortunately) not work as it would in C, since the second expression will be evaluated even when `a` is `NULL`.

Initialization of multiple variables of the same type in the same statement, is not supported, for example:

```
int var1 = 20, var2 = 10, var3 = 15;
```

and initialization of an array must occur after it has been declared (see example below).

There are a limited number of primitive types in CSS, with the others defined in C being equivalent to one of these basic CSS types (see Section 7.4.5 [css-types], page 91). This is because everything in CSS is actually represented by an object, so the storage-level differences between many different C types are irrelevant in the script language.

Pointers in CSS are restricted to point to an object which lies in an array somewhere, or is a single entity. The pointer is "smart", and it is impossible to increment a pointer that points to a single entity, which makes all pointer arithmatic safe:

```
css> int xxx;
css> int* xp = &xxx;
css> print xp
(Int)* xp --> (Int) xxx = 0
css> xp++;
Cannot modify a NULL or non-array pointer value
>1      xp++;
css> print *(xp +1);
Cannot modify a NULL or non-array pointer value
>1      print *(xp +1);
```

but it is possible to increment and perform arithmetic on a pointer when it points to an array:

```
css> int xar[10];
css> xp = xar;
css> xar = {0,1,2,3,4,5,6,7,8,9};
css> print xar
(Int) xar[10] {
0       1       2       3       4       5       6       7       8       9
}
css> print xp
(Int)* xp --> (Int) xar[10] {
0       1       2       3       4       5       6       7       8       9
}
css> print *(xp+2);
(Int)  = 2
css> print *(xp+3);
(Int)  = 3
css> xp++;
css> print *(xp+3);
(Int)  = 4
css> print *(xp+9);
```

```
Array bounds exceeded
>1      p *(xp+9);
(void) Void
css> print *(xp+8);
(Int)  = 9
```

The error that occurred when the `print *(xp+9)` command was issued (after `xp` already points to `xar[1]`), illustrates the kind of "smart pointers" that are built into CSS, which prevent crashes by preventing access to the wrong memory areas.

7.3.3 Extensions Available in CSS

There are several "extensions" of the C/C++ language available in CSS. The most obvious one is the ability to shorten the path to refer to a particular hard-coded object by skipping those elements that are the first of a given group. Thus, if referring to a unit in the first layer of a network, in the first project, one can say: `.units[x]` instead of `.projects[0].networks[0].layers[0].units[x]`.

Also, one can often avoid the use of a type specifier when initializing a new variable, because CSS can figure out the type of the variable from the type of the initializing expression:

```
var1 = "a string initializer";          // type is inferred from initializer
//        instead of
String var2 = "a string initializer";   // instead of explicitly declared
```

When referring to a hard-coded type, CSS will automatically use the actual type of the object if the object derives from the `ta_Base` class which is aware of its own type information.

CSS does not pay attention to the distinction between `ptr->mbr` and `obj.mbr`. It knows if the object in question is an object itself or a pointer to an object, and can figure out how to access the members appropriately.

All of the basic CSS types (see Section 7.4.5 [css-types], page 91) know how to convert themselves into the other types automatically, without even casting them. However, you can use an explicit cast if you want the code to compile properly in standard C/C++.

Also, all script variables for hard-coded objects are implicitly pointers, even if not declared as such. This is because script objects are not the same thing as hard-coded ones, and can only act at best as reference variables for them (i.e., essentially as pointers, but you can use the `obj.mbr` notation, see previous paragraph). Thus, while you can write CSS code that would also compile as C++ code by using `ptr->mbr` notation to refer to hard-coded objects, you can also cheat and use the simpler `obj.mbr` notation even when `obj` is a pointer.

7.3.4 Features of C++ for C Programmers

Since more users are likely to be familiar with C than C++, this section provides a very brief introduction to the essentials of C++.

The central feature of C++ is that it extends the `struct` construct of C into a full-fledged object oriented programming (OOP) language based around a `class`. Thus, a class

or object has data members, like a `struct` in C, but it also has functions associated with it. These *member functions* or *methods* perform various functions associated with the data members, and together the whole thing ends up encapsulating a set of related tasks or operations together in a single entity.

The object-oriented notion is very intuitive for neural-network entities like units:

```
class Unit {    // this defines an object of type Unit
public:         // public means that any other object can access the following

  // these are the data members, they are stored on the object
  float         net_input;      // this gets computed by the weights, etc.
  float         activation;     // this is computed by the function below
  float         target;         // this is set by the environment before hand
  float         error;          // this gets computed by the function below

  // these are the member functions, they can easily access the data members
  // associated with this object
  virtual void  Compute_Activation()
  { activation = 1.0 / (1.0 + exp(net_input)); }
  virtual void  Compute_Error()
  { error = target - activation; error *= error; }
};
```

The member functions encapsulate some of the functionality of the unit by allowing the unit to update itself by calling its various member functions. This is convenient because different types of units might have different *definitions* of these functions, but they all would have a `Compute_Activation()` function that would perform this same basic operation. Thus, if you have an object which is an instance or token of the type `Unit`, you can set its data members and call its member functions as follows:

```
Unit un;
un.net_input = 2.5;
un.target = 1.0;
un.Compute_Activation();
un.Compute_Error();
```

Thus, you use the same basic notation to access data members as member functions (i.e., the `obj.mbr` syntax). To illustrate why this encapsulation of functions with objects is useful, we can imagine defining a new object type that is just like a unit but has a different activation function.

```
class TanhUnit : public Unit {          // we're going to inherit from a Unit
public:
  void          Compute_Activation()    { activation = tanh(net_input); }
};
```

This notation means that the new type, `TanhUnit`, is just like a `Unit`, except that it has a different version of the `Compute_Activation()` function. This is an example of *inheritance*, which is central to C++ and OOP in general. Thus, if we have something which we know to be a unit of some type (either a `Unit` or a `TanhUnit`, or maybe something else derived from a `Unit`), we can still call the `Compute_Activation()` function and expect it to do the right thing:

```
Unit* un;
un->Compute_Activation();
```

This is an example of a *virtual member function*, because the actual function called depends on what actual type of unit you have. This is why the original definition of the `Compute_Activation()` function has the `virtual` keyword in front of it. Virtual functions are an essential part of C++, as they make it possible to have many different definitions or "flavors" of a given operation. Since these differences are all encapsulated within a standard set of virtual functions, other objects do not need to have special-case code to deal with these differences. Thus, a very general purpose routine can be written which calls all of the `Compute_Activation()` functions on all of the units in a network, and this code will work regardless of what actual type of units are in the network, as long as they derive from the *base type* of `Unit`.

While there are a number of other features of C++, the PDP++ software mainly makes use of the basics just described. There are a couple of basic object types that are used in the software for doing file input/output, and for representing character-string values.

The C++ way of doing file input/output is via the *stream* concept. There is an object that represents the file, called a stream object. There are different flavors of stream objects depending on whether you are doing input (`istream`), output (`ostream`) or both (`iostream`). To actually open and close a file on a disk, there is a version of the `iostream` called an `fstream` that has functions allowing you to open and close files. To send stuff to or read stuff from a file, you use something like the "pipe" or i/o redirection concept from the standard Unix shells. The following example illustrates these concepts:

```
fstream fstrm;     // fstrm is a file stream, which can do input or output

// we are opening the file by calling a member function on the
// fstream object. the enumerated type ios::out means 'output'
// also available are ios::in, ios::app, etc.

fstrm.open("file.name", ios::out);

// we "pipe" stuff to the fstrm with the << operator, which can deal
// with all different types of things (ints, floats, strings, etc.)

fstrm << "this is some text " << 10 << 3.1415 << "\n";

fstrm.close();     // again, the file is closed with a member fun
```

The mode in which the file is opened is specified by an *enumerated type* or an `enum`. This provides a way of giving a descriptive name to particular integer values, and it replaces the use of string-valued arguments like "r" and "w" that were used in the `open()` function of the standard C library. The base class of all the stream types is something called `ios`, and the enum for the different modes a file can be opened in are defined in that type, which is why they are *scoped* to the `ios` class by the `ios::` syntax. The definition of this enum in ios is as follows:

```
enum open_mode  { in=1, out=2, ate=4, app=010, trunc=020,
                        nocreate=040, noreplace=0100 };
```

which shows that each enum value defines a bit which can be combined with others to affect how the file is opened.

The following example illustrates how file input works with streams. One simply uses the >> operator instead of <<. Note that the fstream has to be opened in `ios::in` mode as well:

```
fstream fstrm;     // fstrm is a file stream, input or output

// we are opening the file by calling a member function on the
// fstream object. the enumerated type ios::out means 'output'
// also available are ios::in, ios::app, etc.

fstrm.open("file.name", ios::in);

// these variables will hold stuff that is sucked in from the stream
String words[4];
int number;
float pi;
// this assumes that the stream was written by the output example
// given previously
fstrm >> words[0] >> words[1] >> words[2] >> words[3] >> number >> pi;

fstrm.close();     // again, the file is closed with a member fun
```

7.4 CSS Reference Information

This section of the manual contains reference material on many aspects of CSS (excluding the basic C/C++ language itself, for which many excellent reference sources exist).

7.4.1 Name and Storage Spaces in CSS

Like C and C++, CSS has several different places that it can put variables and functions. Some of these amount to ways of organizing things that have a similar role in the same list, so that they can all be viewed together. These are essentially transparent to the user, and are described in the context of the various commands that print out lists of various types of functions and variables. The remaining distinctions have consequences for the programmer, and are described below.

The initial program space in CSS is different than that of C or C++, since it can actually contain executable code, whereas the compiled languages restrict the basic top-level space to consist of definitions only. Any definitions or variables declared in the top-level space are by default considered to be `static`, which means that they are visible only to other things within that program space. Other program spaces (such as those in a `Script` object or a `ScriptProcess`, `ScriptEnv`, etc.) do not have access to these variables or functions. However, the `extern` declaration will make a variable or function visible across any other program spaces.

Within a function, all variables declared as arguments and local variables are known as "autos", as they are automatically-allocated. These are stored locally on a kind of

stack within the same object that holds the code for the function, and a new set of them are allocated for each invocation of the function. The exception is for variables declared `static`, which are also stored on the function object, but the same one is used for all invocations of the function.

All new types that are declared (including enums and classes) are put in the global typespace, which is available to all program spaces.

7.4.2 Graphical Editing of CSS Classes

A class object defined within CSS can be edited using the `edit` command or the `EditObj` function (described below). Classes offer the ability to customize the edit dialog through the use of comment directives, which are the same as those used in the hard-coded C++ classes with the TypeAccess system. These allow class member functions to be associated with a button (using the #BUTTON directive) which, when pressed, calls the member function. By default functions are added to an "Actions" menu, but the #MENU_ON_menuname directive puts that function on a menu named "menuname". The full list of directives is given in Section 18.3 [prog-comdir], page 295.

7.4.3 CSS Startup options

The following startup arguments are interpreted by CSS:

`[-f|-file] <file>`
: Compile and execute the given file upon startup. The default is to then exit after execution, but this can be overridden by the `-i` flag.

`[-e|-exec] <code>`
: Compile and execute the given code upon startup. The code is passed as a single string argument, and should contain CSS code separated by semicolons. The default is to then exit after execution, but this can be overridden by the `-i` flag.

`[-i|-interactive]`
: If using `-f` or `-e`, CSS will go into interactive (prompt) mode after startup execution.

`-v[<number>]`
: Run CSS with the initial debug level set to given number (default 1)

`[-b|-bp] <line>`
: Set an initial breakpoint at the given line of code (only if using `-f`).

`[-gui]` Enable the graphical-user-interface to CSS class objects. This is not activated by default, since most uses of CSS don't involve the gui.

Any other arguments can be accessed by user script programs by the global variables `argv` (an array of strings) and `argc` (an int).

Note that it is possible to make a self-executing css program by putting the following on the very first line of the file:

```
#!/usr/local/pdp++/bin/SUN4/css -f
```

In addition to these arguments, CSS looks for a file named '`.cssinitrc`' in the user's home directory, which contains CSS code that is run when CSS is started. This is useful for setting various command `aliases`, and defining commonly-used `extern` functions. An example '`.cssinitrc`' file is located in '`config/std.cssinitrc`'.

7.4.4 The CSS Command Shell

The primary interface to CSS is via the command shell, which provides a prompt and allows the user to enter commands, etc. This shell has some useful features, including *editing*, *history*, and *completion*, all of which are provided by the GNU readline library.

The current line can be edited using a subset of the emacs editing commands, including Ctrl-f, Ctrl-b, Ctrl-a, Ctrl-e, etc.

A running history of everything that has been entered is kept, and can be accessed by using the Ctrl-p and Ctrl-n commands (previous and next, respectively). This makes it easy to repeat previous commands, especially when combined with the editing facility.

Completion occurs when the TAB key is pressed after some partially-entered expression, causing the shell to suggest a completion to the expression based on the part that has already been entered. If there is more than one possible completion, then the shell will beep, and pressing TAB a second time will produce a list of all of the possible completions. Completion is based on all of the keywords currently defined (including types, commands, functions, user-defined variables and functions, etc), except in the following two cases: If the expression starts with a ".", it is interpreted as a path, and the next segment of the path to either an object or a member function of an object is suggested by the shell. If the expression contains a scoping operator "::", then the completion will interpret whatever is in front of the scoping operator as a type name, and will suggest the possible members, subtypes, etc. of that type as completions.

7.4.5 Basic Types in CSS

There are six basic "primitive" types in CSS: `Int`, `Real`, `char`, `String`, `bool`, and `enum`. An `Int` is essentially an `int`, a `Real` is essentially a `double`, and a `String` is a C++, dynamically-allocating string object. A `char` is just an int object that will convert into its equivalen ASCII character instead of its numerical value when converted into a String. A `bool` has two values: `true` and `false`. An `enum` has defined enumeration values, and will convert a string representation of one of those values into the corresponding symbolic/numeric value.

All of the other flavors of `int` in C, including `short`, `long`, `unsigned`, etc, are all just defined to be the same as an `Int`. Similarly, a `float` and `double` are defined to be the same as a `Real`. The `String` object is the same as the one that comes with the GNU libg++ library (slightly modified), which provides all of the common string manipulation functions as member functions of the `String` object, and handles the allocation of memory for the string dynamically, etc.

In addition to the primitive types, there are reference, pointer, and array types. Also, there are both script-defined and TypeAccess-derived hard-coded `class` objects. Finally, there are types that point to hard-coded members of class objects, which correspond to all

of the basic C types, and, unlike the basic script types, are sensitive to the actual size of a `short` vs a `long`, etc.

Finally there is a special `SubShell` type, which allows one to have multiple compile spaces (cssProgSpace's) within css. Thus, one could `load` one program into the current shell, and use a SubShell to load another one without getting rid of the first. The two shells can communicate via `extern` objects, and they share the same type space. Typical usage is:

```
css> extern SubShell sub_shell;
css> chsh sub_shell
```

The first line defines the object (`extern` is recommended else it will be removed when the parent shell is recompiled), and the second switches to it. To exit from the sub shell, just use `exit` or *`ctrl-D`* (and have faith that when you answer 'y' you will be returned to the parent shell).

7.4.6 CSS Commands

The following are the commands available in CSS. Commands are a little bit "special" in that they are typically executed immediately, they can be called with arguments without using the parentheses required in normal C functions (though they can be used if desired), and they do not need to be terminated with a semicolon (note that this also means that commands cannot extend across more than one line). Also, they do not generate return values. In general, commands provide debugging and program management (loading, listing, etc) kinds of facilities.

`alias <cmd> <new_nm>`
: Gives a new name to an existing command. This is useful for defining shortcuts (e.g., `alias list ls`), but does not allow more complex functionality. For that, either define a new function, or use a pre-processor `#define` macro.

`chsh <script_path>`
: Switches the CSS interface to access the CSS script object pointed to by the given path. This is for hard-coded objects that have CSS script objects in them (of type `cssProgSpace`).

`clearall`
: Clears out everything from the current program space. This is like restarting the CSS shell, compared to `reset` which does not remove any variables defined at the top-level.

`commands`
: Shows a list of the currently available commands (including any aliases that have been defined, which will appear at the end of the list).

`constants`
: Shows a list of the pre-defined constants that have been defined in CSS. These are just like globally-defined `Int` and `Real` values, and thus they can be assigned to different values (though this is obviously not recommended).

`cont`
: Continues the execution of a program that was stopped either by a breakpoint or by single-stepping. To continue at a particular line in the code, use the `goto` command.

`debug <level>`
: Sets the debug level. Level 1 provides a trace of the source lines executed. Level 2 provides a more detailed, machine-level trace, and causes `list` to show the program at the machine level instead of at the usual source level. Levels greater than 2 provide increasing amounts of detail about the inner workings of CSS, which should not be relevant to most users.

`define`
: Toggles the mode where statements that are typed in become part of the current program to be executed later (define mode), as opposed the default (run mode) where statements are executed immediately after entering them.

`defines`
: Shows a list of all of the current `#define` pre-processor macros.

`edit <object> [<wait>]`
: If the GUI (graphical user interface) is active (i.e., by using `-gui` to start up CSS), `edit` will bring up a graphical edit dialog for the given object, which must be either a script-defined or hard-coded `class` object. The optional second argument, if `true`, will cause the system to wait for the user to close the edit dialog before continuing execution of the script.

`enums`
: Shows a list of all the current `enum` types. Note that most `enum` types are defined within a `class` scope, and can be found there by using the `type` command on the class type.

`exit`
: Exits from the program (CSS), or from another program space if `chsh` (or its GUI equivalent) was called.

`frame [<back>]`
: Shows the variables and their values associated with the current block or frame of processing. The optional argument gives the number of frames back from the current one to display. This is most relevant for debugging at a breakpoint, since otherwise there will only be a single, top-level frame to display.

`functions`
: Shows a list of all of the currently defined functions.

`globals`
: Shows a list of all of the currently defined global variables, including those in the script and hard-coded ones.

`goto <src_ln>`
: Continues execution at the given source line.

`help [<expr>]`
: Shows a short help message, including lists of commands and functions available. When passed argument (command, function, class, etc), provides help information for it.

`inherit <object_type>`
: Shows the inheritance path for the given object type.

`list [<start_ln> [<n_lns>]] [<function>]`
: Lists the program source (or machine code, if `debug` is 2 or greater), optionally starting at the given source line number, and continuing for either 20 lines (the initial default) or the number given by the second argument (which then

becomes the new default). Alternatively, a function name can be given, which will start the listing at the beginning of that function (even if the function is `extern`al and does not appear in a line-number based list). `list` with no arguments will resume where the last one left off.

`load <program_file>`
: Loads and compiles a new program from the given file.

`mallinfo`
: Generates a listing of the current `malloc` memory allocation statistics, including changes from the last time the command was called.

`print <expr>`
: Prints the results of the given expression (which can be any valid CSS expression), giving some type information and following with a new line (`\n`). This is useful for debugging, but not for printing values as part of an executing program.

`printr <object>`
: Prints an object and any of its sub-objects in a indented style output. This can be very long for objects near the top of the object hierarchy (i.e., the root object), so be careful!

`reload`
: Reloads the current program from the last file that was `load`ed. This is useful because you do not have to specify the program file when making a series of changes to a program.

`remove <var_name>`
: Removes given variable from whatever space it was defined in. This can be useful if a variable was defined accidentally or given the wrong name during interactive use.

`reset`
: Reset is like `clearall`, except that it does not remove any top-level variables that might have been defined. Neither of these commands will remove anything declared `extern`.

`restart`
: Resets the script to start at the beginning. This is useful if you want to stop execution of the program after a break point.

`run`
: Runs the script from the start (as opposed to `cont` which continues execution from the current location).

`setbp <src_ln>`
: Sets a breakpoint at the given source-code line. Execution of the program will break when it gets to that line, and you will be able to examine variables, etc.

`setout <ostream>`
: Sets the default output of CSS commands to the given stream. This can be used to redirect listings or program tracing output to a file.

`settings`
: Shows the current values of various system-level settings or parameters. These settings are all static members of the class `ta_Misc`, and can be set by using the scoped member name, for example: `ta_Misc::display_width = 90;`

`shell <"shell_cmd">`
: Executes the given Unix shell command (i.e., `shell "ls -lt"`).

`showbp` Shows a list of all currently defined breakpoints, and the source code line they point to.

`source <cmd_file>`
Loads a file which contains a series of commands or statements, which are executed exactly as if they were entered from the keyboard. Note that this is different than `load`ing a program, which merely compiles the program but does not execute it immediately thereafter. `source` uses run mode, while `load` uses define mode.

`stack` Displays the current contents of the stack. This can be useful for debugging.

`status` Displays a brief listing of various status parameters, such as current source line, depth, etc.

`step <step_n>`
Sets the single-step mode for program execution. The parameter is the number of lines to step through before pausing. A value of 0 turns off single stepping.

`tokens <obj_type>`
Lists the instances of the given object type which are known to have been created. Many object types do not register tokens, which will be indicated in the results of this command if applicable. It is possible to refer to the objects by their position in this list with the `Token` function, which can be a useful shortcut to using the object's path.

`trace [<level>]`
Displays a trace of the functions called up to the current one (i.e., as called from within a breakpoint). A trace level of 0 (the default) just gives function names, line numbers, and the source code for the function call, while level 1 adds stack information, level 2 adds stack and auto variable state information, and level 3 gives a complete dump of all available information.

`type <type_name>`
Gives type information about the given type. This includes full information about classes (both hard-coded and script-defined), including members, functions, scoped types (enums), etc.

`undo` This undoes the previous statement, when in `define` mode.

`unsetbp <src_ln>`
Removes a breakpoint associated with the given source-code line number.

7.4.7 CSS Functions

The following functions are built into CSS, and provide some of the basic functionality found in the standard C library. Note that the `String` and `stream` objects encapsulate many commonly-used C library functions, which have not in general been reproduced in CSS (with the exception of some of the file functions).

`Int access(String fname, int ac_type)`
This POSIX command determines if the given file name is accessible according to the ac_type argument, which should be some bitwise OR of the enums `R_OK W_OK X_OK F_OK`. Returns success and sets errno flag on failure.

`Real acos(Real x)`
: The arc-cosine (inverse cosine) – takes an X coordinate and returns the angle (in radians) such that cos(angle)=X.

`Real acosh(Real x)`
: The hyperbolic arc-cosine.

`Int alarm(int seconds)`
: Generate an alarm signal in the given number of seconds. Returns success and sets errno flag on failure.

`Real asin(Real x)`
: The arc-sine (inverse sine) – takes a Y coordinate and returns the angle (in radians) such that sin(angle)=Y.

`Real asinh(Real x)`
: The hyperbolic arc-sine.

`Real atan(Real x)`
: The arc-tangent (inverse tangent) – takes a Y/X slope and returns angle (in radians) such that tan(angle)=Y/X.

`Real atan2(Real y, Real x)`
: The arc-tangent (inverse tangent) – takes a Y/X slope and returns angle (in radians) such that tan(angle)=Y/X.

`Real atanh(Real x)`
: The hyperbolic arc-tangent.

`Real beta(Real z, Real w)`
: The Beta function.

`Real beta_i(Real a, Real b, Real x)`
: The incomplete Beta function.

`Real bico_ln(Int n, Int j)`
: The natural logarithm of the binomial coefficient "n choose j". The number of ways of choosing j items out of a set containing n elements:

$$\ln \binom{n}{j} = \ln \frac{n!}{k!\,(n-k)!}$$

`Real binom_cum(Int n, Int j, Int p)`
: The cumulative binomial probability of getting j *or more* in n trials of probability p.

`Real binom_den(Int n, Int j, Real p)`
: The binomial probability density function for j "successes" in n trials, each with probability p of success.

$$P(n,j,p) = \binom{n}{j} p^{j} (1-p)^{(n-j)}$$

`Real binom_dev(Int n, Real p)`
: The binomial random deviate: produces an integer number of successes for a binomial distribution with p probability over n trials.

`Int CancelEditObj(obj)`
: Cancels the edit dialog for the given object that would have been opened by `EditObj`.

`Real ceil(Real x)`
: Rounds up the value to the next-highest integral value.

`int chdir(String dir_name)`
: Change the current directory to given argument. Returns success and sets errno flag on failure.

`Real chisq_p(Real X, Real v)`
: Gives the chi-squared statistic P(X^2 | v).

`Real chisq_q(Real X, Real v)`
: Gives the complement of the chi-squared statistic Q(X^2 | v).

`Int chown(String fname, int user, int group)`
: Changes the ownership of the given file to the given user and group numbers. Returns success and sets errno flag on failure.

`Real clock()`
: Returns processor time used by current process in seconds (with fractions expressed in decimals).

`Real cos(Real x)`
: The cosine of angle x (given in radians). Use `cos(x / DEG)` if x is in degrees.

`Real cosh(Real x)`
: The hyperbolic cosine of angle x.

`String ctermid()`
: Returns the character-id of the current terminal.

`String cuserid()`
: Returns the character-id of the current user.

`String_Array& Dir([String& dir_nm])`
: Fills an array with the names of all the files in the given directory (defaults to "." if no directory name is passed). The user should copy the array if they want to keep it around, since the one returned is just a pointer to an internal array object.

`Real drand48()`
: Returns a uniformly-distributed random number between 0 and 1.

`Int EditObj(<object>, [Int wait])`
: This is the function version of the `edit` command. If the GUI (graphical user interface) is active (i.e., by using `-gui` to start up CSS), edit will bring up a graphical edit dialog for the given object, which must be either a script-defined or hard-coded `class` object. The optional second argument, if TRUE, will

cause the system to wait for the user to close the edit dialog before continuing execution of the script.

`Real erf(Real x)`
: The error function, which provides an approximation to the integral of the normal distribution.

`Real erf_c(Real x)`
: The complement of the error function.

`Real exp(Real x)`
: The natural exponential (e to the power x).

`css* Extern(String& name)`
: Returns the object with the given name on the 'extern' variable list. This provides a mechanism for passing arbitrary (i.e., class objects) data across different name spaces (i.e., across different instances of the css program space), since you can pass the name of the extern class object that contains data relevant to another script, and use this function to get that object from its name.

`Real fabs(Real x)`
: The absolute value of x.

`Real fact_ln(Int x)`
: The natural logarithm of the factorial of x (x!).

`void fclose(FILE fh)`
: Closes the file, which was opened by `fopen`. The FILE type is not actually a standard C FILE, but actually a `fstream` type, so stream operations can be performed on it.

`Real floor(Real x)`
: Rounds the value down to the next lowest integral value.

`Real fmod(Real x, Real y)`
: Returns the value of x modulo y (i.e., `x % y`) for floating-point values.

`FILE fopen(String& file_nm, String& mode)`
: Opens given file name in the given mode, where the modes are `"r"`, `"w"`, and `"a"` for read, write and append. The FILE type is not actually a standard C FILE, but actually a `fstream` type, so stream operations can be performed on it.

`void fprintf(FILE strm, v1 [,v2...])`
: Prints the given arguments (which must be comma separated) to the stream. Values to be printed can be of any type, and are actually printed with the `<<` operator of the stream classes. Unlike the standard C function, there is no provision for specifying formatting information. Instead, the formatting must be specified by changing the parameters of the stream object. The FILE type is not actually a standard C FILE, but actually a `fstream` type, so stream operations can be performed on it.

`Real Ftest_q(Real F, Real v1, Real v2)`

Gives the F probability distribution for P(F | (v1 < v2)). Useful for performing statistical significance tests. The _q suffix means that this is the complement distribution.

`Real gamma_cum(Int i, Real l, Real t)`

The cumulative gamma distribution for event i with parameters l=lambda and t=time, which is the same as `gamma_p(j, l * t)`.

`Real gamma_den(Int j, Real l, Real t)`

The gamma probability density function for j events, l=lambda, and t=time.

$$P(j,l,t) = \frac{l^j \, t^{(j-1)}}{j!} e^{-lt} \qquad (t > 0)$$

`Real gamma_dev(Int j)`

A random gamma deviate: how long it takes to wait until j events occur with a unit lambda (l=1).

`Real gamma_ln(Real z)`

The natural logarithm of the gamma function, which is a generalization of (n-1)! to real-valued arguments. Note that this is not the gamma probability distribution.

$$Gamma(z) = \int_0^x t^{z-1} e^{-t} \, dt$$

`Real gamma_p(Real a, Real x)`

The incomplete gamma function:

$$P(a,x) = \frac{1}{Gamma(a)} \int_0^x t^{a-1} e^{-t} \, dt \qquad (a > 0)$$

`Real gamma_q(Real a, Real x)`

The incomplete gamma function as the complement of `gamma_p`

$$P(a,x) = \frac{1}{Gamma(a)} \int_x^{inf} t^{a-1} e^{-t} \, dt \qquad (a > 0)$$

`Real gauss_cum(Real x)`

The cumulative of the Gaussian or normal distribution up to given x (sigma = 1, mean = 0).

`Real gauss_den(Real x)`
: The Gaussian or normal probability density function at x with sigma = 1 and mean = 0.

`Real gauss_inv(Real p)`
: Inverse of the cumulative for p: returns z value for given p.

`Real gauss_dev()`
: Returns a Gaussian random deviate with unit variance and 0 mean.

`String getcwd()`
: Returns the current working directory path.

`String getenv(String var)`
: Returns the environment variable definition for varable var.

`Int getegid()`
: Returns the current effective group id number for this process.

`Int geteuid()`
: Returns the current effective user id number for this process.

`Int getgid()`
: Returns the current group id number for this process.

`Int getuid()`
: Returns the current user id number for this process.

`String getlogin()`
: Returns the name the current user logged in as.

`Int getpgrp()`
: Returns the process group id for current process.

`Int getpid()`
: Returns the process id for current process.

`Int getppid()`
: Returns the parent process id for current process.

`Int gettimesec()`
: Returns current time of day in seconds.

`Int gettimmesec()`
: Returns current time of day in microseconds.

`Real hyperg(Int j, Int s, Int t, Int n)`
: The hypergeometric probability function for getting j number of the "target" items in an environment of size "n", where there are "t" targets and a sample (without replacement) of this environment of size "s" is taken.

`Int isatty()`
: Returns true if the current input terminal is a tty (as opposed to a file or a pipe or something else).

`Int link(String from, String to)`
: Creates a hard link from given file to other file. (see also symlink). Returns success and sets errno flag on failure.

`Real log(Real x)`
: The natural logarithm of x.

`Real log10(Real x)`
: The logarithm base 10 of x.

`Int lrand48()`
: Returns a uniformly-distributed random number on the range of the integers.

`MAX(<v1>, <v2>) or max(<v1>, <v2>)`
: Works like the commonly-used `#define` macro that gives the maximum of the two given arguments. The return type is that of the maximum-valued argument.

`MIN(<v1>, <v2>) or min(<vi>, <v2>)`
: Just like `MAX`, except it returns the minimum of the two given arguments.

`Int pause()`
: Pause (wait) until an alarm or other signal is received. Returns success and sets errno flag on failure.

`void perror(String prompt)`
: Prints out the current error message to stderr (cerr). The prompt argument is printed before the error message. Also, the global variable `errno` can be checked. Further, there is an include file in css/include called errno.css that defines an enumerated type for the defined values of errno.

`Real poisson_cum(Int j, Real l)`
: The cumulative Poisson distribution for getting 0 to j-1 events with an exected number of events of l (lambda).

`Real poisson_den(Int j, Real l)`
: The Poisson probability density function for j events given an expected number of events of l (lambda).

$$P(j, l) = \frac{l^j}{j!} e^{-l}$$

`Real poisson_dev(Real l)`
: A random Poisson deviate with a mean of l (lambda).

`Real pow(Real x, Real y)`
: Returns x to the y power. This can also be expressed in CSS as x ^ y.

`void PrintR(<object>)`
: This is the function version of the `printr` command. Prints an object and any of its sub-objects in a indented style output. This can be very long for objects near the top of the object hierarchy (i.e., the root object), so be careful!

`void printf(v1 [,v2...])`
: Prints the given arguments (which must be comma separated) to the standard output stream. Values to be printed can be of any type, and are actually printed with the `<<` operator of the stream classes. Unlike the standard C function, there is no provision for specifying formatting information. Instead,

the formatting must be specified by changing the parameters of the standard stream output object, `cout`. The FILE type is not actually a standard C FILE, but actually a `fstream` type, so stream operations can be performed on it.

`Int putenv(String env_val)`
: Put the environment value into the list of environment values (avail through getenv).

`Int random()`
: Returns a uniformly-distributed random number on the range of the integers. CSS actually uses the `lrand48` function to generate the number given the limitations of the standard `random` generator.

`String_Array& ReadLine(istream& strm)`
: Reads a line of data from the given stream, and returns a reference to an internal array (which is reused upon a subsequent call to ReadLine) of strings with elements containing the whitespace-delimited columns of the line. The size of the array gives the number of columns, etc. This allows one to easily implement much of the functionality of awk. See the file '`css_awk.css`' in '`css/include`' for an example.

`Int rename(String from, String to)`
: Renames given file. Returns success and sets errno flag on failure.

`Int rmdir(String dir_name)`
: Removes given directory. Returns success and sets errno flag on failure.

`Int setgid(Int id)`
: Sets group id for given process to that given. Note that only the super-user can in general do this. Returns success and sets errno flag on failure.

`Int setpgid(Int id)`
: Sets process group id for given process to that given. Note that only the super-user can in general do this. Returns success and sets errno flag on failure.

`Int setuid(Int id)`
: Sets user id for given process to that given. Note that only the super-user can in general do this. Returns success and sets errno flag on failure.

`Real sin(Real x)`
: The sine of angle x (given in radians). Use `sin(x / DEG)` if x is in degrees.

`Real sinh(Real x)`
: The hyperbolic sine of x.

`Real sqrt(Real x)`
: The square-root of x.

`void srand48(Int seed)`
: Provides a new random seed for the random number generator.

`Int sleep(Int seconds)`
: Causes the process to wait for given number of seconds. Returns success and sets errno flag on failure.

`Real students_cum(Real t, Real v)`
: Gives the cumulative Student's distribution for v degrees of freedom t test.

`Real students_den(Real t, Real v)`
: Gives the Student's distribution density function for v degrees of freedom t test.

`Int symlink(String from, String to)`
: Creates a symbolic link from given file to other file. (see also link). Returns success and sets errno flag on failure.

`void system(String& cmd)`
: Executes the given command in the Unix shell.

`Real tan(Real x)`
: The tangent of angle x (given in radians). Use `tan(x / DEG)` if x is in degrees.

`Real tanh(Real x)`
: The hyperbolic tangent of x.

`Int tcgetpgrp(Int file_no)`
: Gets the process group associated with the given file descriptor. Returns success and sets errno flag on failure.

`Int tcsetpgrp(Int file_no)`
: Sets the process group associated with the given file descriptor. Returns success and sets errno flag on failure.

`String ttyname(Int file_no)`
: Returns the terminal name associated with the given file descriptor.

`Token(<obj_type>, Int tok_no)`
: Returns the token of the given type of object at index `tok_no` in the list of tokens. Use the `tokens` command to obtain a listing of the tokens of a given type of object.

`TypeDef Type(String& typ_nm | <obj_type>)`
: Returns a type descriptor object (generated by TypeAccess), for the given type name or type object (the type object can be used directly in some situations, but not all).

`Int unlink(String fname)`
: Unlinks (removes) the given file name.

7.4.8 Parameters affecting CSS Behavior

All of the settings that control the behavior of CSS are contained in the global object called `cssSettings`. This is actually just a reference to the `taMisc` class, which is part of the TypeAccess system. The members of this class that can be set by the user are listed below (see also see Section 6.6 [gui-settings], page 63):

`int display_width`
: Width of the shell display in characters.

`int sep_tabs`
: Number of tabs to separate items by in listings.

`int search_depth`
: The recursive depth at which css stops searching for an object's path.

`TypeInfo type_info`
: The amount of information about a class type that is reported when the "type" command is used in CSS.

  Type_info has one of the following values:

  `MEMB_OFFSETS` shows the byte offset of members

  `All_INFO` shows all type info except memb_offsets

  `NO_OPTIONS` shows all info except type options

  `NO_LISTS` shows all info but lists

  `NO_OPTIONS_LISTS` shows all info but options and lists

  The default is `NO_OPTIONS_LISTS`

7.5 Common User Errors

The following are common user errors, which you can anticipate and avoid by reading about them in advance.

Forgetting the semicolon: CSS, being essentially like C or C++, requires most statements to end with a semicolon (;). This allows one to spread statements over multiple lines, since the semicolon and not the newline indicates the end of a statement. However, it is easy to forget it when typing stuff in interactively. The consequences of this are that the following command or statement will be treated as if it was part of the one where the semicolon was forgotten, usually resulting in a `Syntax Error`. Note that commands are exempt from the semicolon requirement, and, as a corollary, can not be extened across multiple lines.

Delayed impact of syntax error: This happens when the user types in something erroneous (i.e., something that will result in a Syntax Error), without following it with a semicolon (usually because it was supposed to be a command, which does not require a semicolon). However, because the entry was neither a command nor followed by a semicolon, it treats the following material as being on the same line, so that only after the second line has been entered (typically), is the first syntax error caught. The solution is to simply press enter a couple of times (or hit the semicolon and press enter), which will clear out the preceding line and let you continue on.

Trying to `print` or do something else with a `void`: If an expression cannot be evaluated (resulting in a `void` value), or a function is called which returns a type of `void` (i.e., nothing is returned), and the result of this expression is then printed or passed to some other function, the following error will result: `Incomplete argument list for: <function_name>, Should have: 1 Got: 0`. Since the void does not get passed to the function or command which is expecting an argument, the function/command (typically `print`) complains with the above error.

7.6 Compiling CSS files as C++ Hard Code

Because they use the standard C++ syntax, CSS script files can be compiled into "hard" code that runs (fast) as a stand-alone application. Of course, the files must not use any of the CSS shortcuts, and must otherwise be standard C++ code (i.e., no executable code outside of functions, using the correct `.` or `->` operator, etc).

There are three main steps that are needed to compile your CSS code. The first, which need only be done once, is the creation of the appropriate libraries that will be linked with the C++ compiled code to produce an executable. The second is formatting your file so that it can be both run by CSS and compiled by C++. The third is creating a Makefile which will allow C++ to compile your file. An example of this is provided in the directory '`demo/css`'.

There are two special libraries that are linked into your C++ executable, one in the '`src/ta`' directory, and one in the '`src/css`' directory. Both can be made using top-level Makefile commands, or by going into the directories separately. '`make LibMin`' makes a library which contains the minimal type-access stuff from the '`src/ta`' directory of the distribution, and it makes the '`libtypea_min`' library. '`make hard_css`' makes a library of special functions in '`src/css`' (e.g., the "special math" functions which have been added into CSS and are not part of the standard C library, and the Dir and ReadLine functions). This library is '`libhard_css`'. Both of these libraries will be visible to the C++ compiler using the makefiles as described below.

The CSS file needs to have a couple of conditially-included elements that resolve the basic differences between CSS and C++. Basically, this amounts to including a header file that estabilshes some defines and includes some commonly-used standard library headers, which are automatically present in CSS. This is the '`css/hard_of_css.h`' file. It is only included when compiling by making it conditional on the pre-processor define `__CSS__`, which is automatically defined by CSS. Also, '`hard_of_css.h`' defines a `main` function which calls a function called `s_main`, which is the actual main function that should be defined in your script.

The following example illustrates these elements, and can be used as a template for making your own CSS files compilable (see '`demo/css`' for a larger example):

```
#ifndef __CSS__
#include <css/hard_of_css.h>
#endif

void s_main(int ac, String* av) {
  // do stuff here..
}

// in css, call our main function as the thing we actually run..
#ifdef __CSS__
s_main(argc, argv);
#endif
```

In order to make the C++ compiling environment as similar to CSS as possible, a variant of the same Makefile can be used. This assumes that the makefiles for your CPU type are correct (i.e., those used in installing the PDP++/CSS source-code distribution (see Chapter 2

[inst], page 4, Section 2.2 [inst-prog], page 9)). The following steps will result in a Makefile that will enable you to compile your CSS code.

1) Copy the sample makefile in '`config/Makefile.hard_of_css`' into the directory where your CSS file is to be compiled, and name it '`Makefile.in`'.

2) Edit this file and ensure that the PDPDIR path is pointing to the installed pdp++ distribution.

3) Then, do a `make -f Makefile.in InitMakefile`, which will make a '`Makefile`' in the current directory that can be used to compile your file.

4) To compile, just type `make <filenm>`, where `<filenm>` is the CSS file without any extension (i.e., the name of the executable that will be produced. Some C++ compilers will complain if the file does not end in a "standard" C++ extension like .cc or .C, so you may have to rename it or create a symbolic link from your .css file (CSS does not care about using a non .css extension, as long as you specify the entire file name).

8 Object Basics and Basic Objects

PDP++ is written in the computer language C++. C++ extends the C language in many ways, but its primary addition is the concept of objects and object oriented programming (OOP). Many PDP concepts such as Units, Layers, and Networks easily fall into the object paradigm, while some other concepts such as Processes and Environments are initially difficult to grasp in this framework. The following chapters explain the particulars of the PDP concepts in the object-based paradigm in greater detail. This chapter establishes a number of conventions in PDP++ used for organizing and structuring its objects. It is really only relevant if you will be programming with PDP++, either in the script language or directly in C++ — the end user generally need not be concerned with this level of detail.

In addition to these object basics, this chapter also describes a number of basic types of objects that are used in many different places within the software. Examples of these include lists, groups, arrays, and specifications.

8.1 Object Basics

The following sections describe some basic ideas about objects and general things you can do with them.

8.1.1 What is an Object?

An object has both fields with data and functions (aka methods) which it can perform. A PDP Unit as an object might have the fields `activation` and `netinput`, and some functions like `ComputeActivation()` and `ClearNetInput()`. In C++ this might look like:

```
class Unit {
  float         activation;     // this is a member holding activation
  float         netinput;       // this is a member holding net input

  virtual void  ComputeActivation(); // this is a function
  void          ClearNetInput();        // this is a function
}
```

C++ also provides the mechanisms for object inheritance in which a new object can be defined as having the same fields and functions as its "parent" object with the addition of a few fields and/or functions of its own. In PDP++ the **BpUnit** object class inherits from the parent **Unit** class and adds fields such as `bias weight` and a BackProp version of `ComputeActivation`. In C++ this might look like:

```
class BpUnit : public Unit {
  float         biasweight;     // a new member in addition to others

  void          ComputeActivation(); // redefining this function
  void          ClearBiasWeight();  // adding a new function
}
```

By *overloading* the ComputeActivation function the BpUnit redefines the way in which the activation is computed. Since it doesn't overload the ClearNetInput function, the BPUnit clears its net input in the exact same way as the standard unit would. The new

function ClearBiasWeight exists only on the BpUnit and is not available on the base Unit class. Through this type of class inheritance, PDP++ provides a hierarchical class structure for its objects.

For a slightly more detailed treatment of some of the basic ideas in OOP and C++, see Section 7.3.4 [css-c++-intro], page 86.

8.1.2 Object Names

In PDP++, many objects have a `name` field. This gives the particular object an identifier that can be used to refer to it in the CSS script language (see Section 7.2.3 [css-tut-access], page 80), and it makes it easier to identify when you need to point to it one of the GUI operations.

All objects when created are given default names that consist of the their type followed by a number which increases for each subsequent object of that type which is created. Thus, the first **Layer** created will be called "Layer_0", and so on.

Note that the name given to an object is not necessarily the same name that object will have when saved to a file. It is simply a value of a particular member of the object.

8.1.3 Saving and Loading Objects

All PDP++ objects can also be saved to a file and loaded back in from that file. Note that when you save an object that has sub-objects in it (i.e. a Layer, which has Units in it), those sub-objects are also saved in the file, and loaded back in as well. Thus, to save all the elements of a project, one only has to save the project object, which will automatically save all of its sub-objects.

There are two ways in which objects can be loaded back in: in one case, you load the file *over* an existing object. This corresponds to the *Load Over* menu item that you will frequently see in PDP++. This will simply replace the values of the existing object with those saved in the file. This also applies to any sub-objects. Note if the existing object has *fewer* of some kind of sub-object than the saved file, new sub-objects will be created as needed. However, the same is not also true if the existing object has *more* of these sub-objects—they are not removed to make the existing object exactly the same as that which was saved in the file. If you want to make sure the object is exactly as it was saved, open a new one, do not open over an existing one.

To open a new object from a saved file, use the *Open in* menu item (instead of *Load Over*). This is equivalent to opening a new item into a parent group object. For example, the project has a group that contains its networks. If you open a saved network file in this group, as opposed to over an existing network in the group, a new network will automatically be created (along with all of its saved substructure).

Some types of objects have default "extensions" that are automatically added on to the file name to identify the type of file. For example, projects are saved with a '`.proj`' extension. Further, some objects automatically compress the file after saving, (and automatically uncompress it before loading) in order to save disk space. This is because the file format objects are saved in is actually text-based and human readable. Thus, it is much more efficient to save large files in a compressed form. These files automatically get the

compression suffix associated with the type of compression used (.gz for gzip, which is the default).

8.2 Groups

Groups are one of the major workhorse objects in the PDP++ software. Virtually every object is contained in a group object. Thus, groups provide a way of managing other objects, creating them, ordering them, and iterating through them.

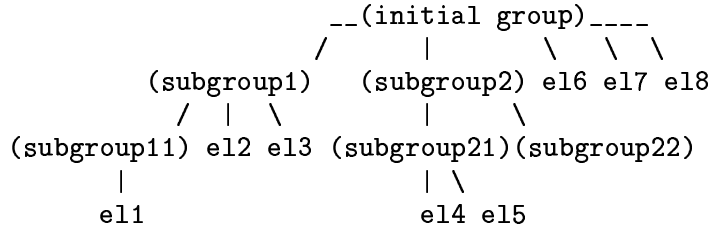
A particular group typically manages objects of the same type. Thus, a group of units always contains units of one form or another. Groups have a *default type* of object that they expect to contain, and this is what they will create if asked to create a new object.

Groups also have a notion of a *default element*, which can be useful. For example, the default element of the group of color specifications on the root object is the default color spec that will be used by all windows that need a color spec.

There are two basic types of groups. The simpler form of group is actually just a **List**. A **List** object manages a single list of elements, and does not allow for any sub-grouping of these elements. Lists are used to hold simple objects that probably don't need to have such subgroups.

A **Group** may also contain subgroups, which are similar in type to the group itself. The nested structure of subgroups within subgroups within groups can be organized into a conceptual "tree" of groups and elements. The initial group comprises the base of the tree with each of its subgroups representing a branch. The subgroup's subgroups continually branch off until in the end a group is reached without any further subgroups. The actual elements of the group are conceptually represented as "leaves", and may occur within any group at any level of the group tree.

`Ex:`



In this example there are six groups and eight "leaf" elements. The leaves are numbered in their Depth First Search retrieval order which is used by the `Leaf()` function described in the group functions section See Section 8.2.3 [obj-group-functions], page 111.

In some cases it useful for the same group element to occur in more than one group at time. The parent groups share the element, however, one group is assigned to be the owner (usually the initial group the element was created within). Usually a group will own all of its elements, but sometimes it may contain elements from other groups as well. Those elements which are owned by other groups are known as *linked* elements, and are added to a group by using the `Link()` function described in the group functions section. When

a group is deleted, it breaks its link to the linked elements, but does not delete them as it does the other elements it owns. Subgroups can also be linked.

```
Ex:
        (group1)                  (group2)
        /   |   \                /    |  \
(subgroup1) el1  \.............el2    el3 \
  /  \                                    \
el4   \.................................(subgroup2)
```

In this example group1 has a link to element el2 in group2. El2 is owned by group2 and is a link element of group1. In addition, subgroup1 has a link to subgroup2, which is owned by group2.

The following sections document the various operations and variables of the group types. These are probably of most interest to users who will be manipulating groups in the CSS language, or in programming in PDP++. However, some of the group functions are available in the *Actions* menu in the gui, and their function might not be fully clear from their name and arguments.

8.2.1 Iterating Through Group Elements

The elements in a group with nested sub-structure can be accessed as though this structure were "flattened out". This can be done more efficiently using an iterator scheme as opposed to repeatedly calling the `Leaf()` function. The iterator for going through leaves is an object that remembers information about where it is in the group structure. This iterator is of type **taLeafItr**, and a variable should be declared of this type before beginning a for loop to iterate. The `FirstEl` and `NextEl` functions use this iterator to traverse through the elements of the group.

There is a macro which makes group traversal relatively painless. The following example illustrates its use:

```
Unit* u;                        // a pointer to a unit object
taLeafItr i;                    // the iterator object
FOR_ITR_EL(Unit, u, layer->units., i) {
  u->DoSomething();
}
```

The FOR_ITR_EL macro iterates though the leaves of a group. It accepts four arguments. The first is the type of object expected at the leaf level of the group. The second is a pointer to an object of that type. The third is group followed by its access method. In the example, the group is "layer->units", and the access method is the "." . If the group was instead a pointer to group called "mygroup" then the third argument would be "mygroup->". The fourth argument is the iterator object.

To iterate through all the leaf groups (groups which contain leaf elements) the FOR_ITR_GP macro may be used. Its first argument is the group type (not the leaf type). The second is a pointer to group of that type. The third is pointer group and its access method. And again, the fourth argument is the iterator object.

8.2.2 Group Variables

The Group class has the following variables:

`String name`
: The (optional) name of the group

`int size`
: Number of elements in the group.

`TypeDef* el_typ`
: The "minimum" type of an element in the Group. When objects are created, transferred, or linked in the group, only objects of this type, or objects of a type which inherits from this type are considered for inclusion.

`int el_def`
: The index of the default element of the group. This element is returned when the `Default_El()` function is called on the group.

`int leaves`
: The number of leaves of the Group. This is the total sum of all the elements in this group plus all the elements of its subgroups, their subgroups and so on.

`Group gp`
: This is the group of subgroups.

`Geometry pos`
: The Group's position. In some cases, the group is represented graphically and the position provides a relative offset for the elements of the group

`Geometry geom`
: The Group's geometry. Although the group is accessed linearly, it can be represented graphically in more than one dimension. The geometry specifies the layout of the group's elements.

8.2.3 Group Functions

Groups have many functions for operating on the elements of the group.

These functions return an element of the group. The element is an instance of the group's element type.

`El (int i)`
: Returns the element of the group at index `i`. If the index is out of range, an out of rangeerror is reported, and `NULL` is returned.

`DefaultEl()`
: Returns the element indexed by the `default_el` variable of the group.

`Leaf (int n)`
: Performs a depth first search to find the nth leaf of the group tree.

`FirstEl()`
: Utilizes sets the internal leaf index to zero and return the first leaf.

`NextEl();`
: Increments the internal leaf index and returns the leaf of that index.

`Pop()` Returns the last element of the group, and removes it from the group as well.

`Peek()` This function is similar to the `Pop()` function, but it does not remove the element.

These functions return an integer indicating the index of the element in the group. In all these functions if no match is found, a value of "-1" is returned.

`FindEl (inst el)`
Returns the index of the element of the group which matches `el`.

`FindName (char* name)`
Returns the index of the element which has a name field which matches `name`.

`Find (TypeDef t)`
Returns the index of the first element of type `T`.

`FindLeaf (char* name)`
Returns the leaf index of the leaf element which has a name field which matches `name`.

`FindLeafEl (inst el)`
Returns the leaf index of the leaf element which matches `el`.

These functions are used to add elements to the group.

`AddEl (inst el)`
Adds element `el` to the group

`AddUniqueName (inst el)`
Adds an element `el` to the group and adds an instance index to its name if another element in the group has the same name.

`Push (inst el)`
Adds element `el` to the end of the group.

`PushUnique (inst el)`
Adds element `el` to the end of group only if it is not in the group already.

`PushUniqueName (inst el)`
Adds element `el` to the end of the group only if there are no other elements of the group with the same name.

`Insert (inst el, int i)`
Inserts element `el` at position `i` in the group. If the position `i` is out of range, it is added to the beginning or end of the group accordingly.

`Replace (int i ,inst el)`
Replaces the element at location `i` with the element `el`.

`ReplaceEl (inst rel, inst el)`
Replaces the element `rel` of the group with element `el`. If no match is found, the element `el` is not inserted.

`ReplaceName (char* name, inst el)`
Replaces the element with the name `name` with the element `el`. If no match is found, the element `el` is not inserted.

`Transfer (inst el)`
: Removes `el` from the group or owner it is currently in and adds it to this group.

`Link (inst el)`
: Adds a link to the object `el` to this group. There are also corresponding functions (InsertLink, ReplaceLink, etc) which perform as the do the Add functions except they link instead of add (see Section 8.2 [obj-group], page 109).

`New (int i, typedef t)`
: Creates `i` new objects of type `t` in the group.

These functions are used for removing elements from the group.

`Remove (int i)`
: Removes the element at position i. If the element is owned by the group it deletes the element. If the element is a linked element, it deletes the link.

`Move (int from, int to)`
: Removes an element from position `from` and inserts at position `to`.

`RemoveName (char* name)`
: Removes the element with a name matching `name`.

`RemoveLeafName (char* name)`
: Removes the leaf with a name matching `name`.

`RemoveAll()`
: Removes all the elements of the group.

These functions return a subgroup of the group.

`Gp (int i)`
: Returns the subgroup at index(i)

`LeafGp (int n)`
: Returns the nth subgroup which contains leaves.

`FirstGp()`
: Sets the group index to zero and returns the first subgroup with leaf elements.

`NextGp()` Increments the group index and return the corresponding subgroup with leaf elements.

8.2.4 Group Edit Dialog

The group edit dialog (GED) is in many ways like the the Edit dialog for other objects (see Section 6.5 [gui-edit], page 58). In the GED however, all the elements of the group are editable at one time. The elements are represented in a horizontal scrollbox. If there are more elements than can be represented in the dialog, dragging the scrollbar under the scrollbox will allow access to the other members. The members are ordered sequentially from left to right.

In some cases the elements of the groups may be of different types. When this occurs, all the members of all the element types are listed in the member names section of the dialog.

Elements which do not contain a certain member in the member names section will have a blank field where the edit field of that member would normally appear. The dialog buttons (Ok, Apply, Revert, Cancel) apply to all the elements of the group.

In addition to the standard editing control keys in the Edit Dialog, the following keys are are mapped to special functions.

Meta-f moves the Ibeam forward to the same field in the next group member

Meta-b moves the Ibeam backward to the same field in the previous group member

Many of the functions described in the previous section (see Section 8.2.3 [obj-group-functions], page 111) are available in the menus of the group edit dialog. Note that if you want to operate on the subgroups (e.g., to move them around), you can do EditSubGps which will pull up a dialog of these subgroups where the move and other actions apply to them.

8.3 Arrays

Arrays are the data objects for related sequences of simple data structures like integers, floating point numbers, or strings. Each of these types has its own corresponding array type (ex. floating point numbers (floats) are arranged in a float_Array). All array types however have the the same structure of variables and access functions.

The following sections document the various operations and variables of the array type. These are probably of most interest to users who will be manipulating arrays in the CSS language, or in programming in PDP++. However, some of the array functions are available in the *Actions* menu in the gui, and their function might not be fully clear from their name and arguments.

8.3.1 Array Variables

The array classes have two important variables:

`int size` Indicates the number of elements in the array. Although the array may have allocated additional space, this is the number of elements in use.

`item err` Indicates the value to return when the array is accessed out of range.(ex. If the array had a size of 5 and was asked for element 6 or element -1 then the err value would be returned).

8.3.2 Array Functions

`Alloc (int x)`
Allocates an array with space for x items.

`Reset()` Sets the size of the array to zero, but does not change or free the amount to allocated space.

`Remove(int x)`
Removes the array element at index `x`.

`Permute()`
: Permutes the elements of the array into a random order.

`Sort()`
: Sorts the array in ascending order.

`ShiftLeft (int x)`
: Shifts all the elements in the array `x` positions to the left.

`ShiftLeftPct (float f)`
: Shifts the array to the left by `f` percent.

`El (int x)`
: Returns the element at index `x`, or err if out of range. Indexing starts at zero, therefore an array with five elements would have valid indices for zero to four.

`FastEl (int x)`
: Fast element return. Returns the element at index `x` with no error checking. Caution: PDP++ may behave unexpectedly if this function is called with an index that is out of range.

`Pop()`
: Returns and removes the last element in the array.

`Peek()`
: Returns the last element in the array without removing it.

`Add (item i)`
: Adds `i` to the array.

`Push (item i)`
: Pushes (adds) `i` to the the end of the array

`Insert (item i, int num, int loc)`
: Inserts `num` copies of item `i` at location `loc` in the array.

`Find (item i, int loc)`
: Returns the index of the first element in the array matching item `i` starting at location `loc`.

`RemoveEl (item i)`
: Removes the first element matching item `i`. Returns `TRUE` if a match is found and `FALSE` otherwise.

8.3.3 Array Editing

Arrays are editing using an enhanced version of the standard Edit Dialog See Section 6.5 [The Edit Dialog], page 58. The Array Edit Dialog arranges all of the elements of the array horizontally and allows the user to scroll though the array elements using a horizontal scroll bar at the bottom of the dialog. In addition, number arrays (floats, ints, etc..) can be edited using a Color Array Edit Dialog which is similar in layout to the Array Edit Dialog, but adds a color palette for painting the values See Section 6.7 [gui-colors], page 67.

8.4 Specifications

One of the important design considerations for the PDP++ software was the idea that one should separate state variables from specifications and parameters (see Section 3.3

[over-spec], page 18). The attributes of an object can often be divided into two types– the first type of attributes represent the object's state. These change over time and are usually distinct within each instance of an object. The second group of attributes represent parameters of an object that tend to remain fixed over time, and often have values that are common among instances of the object class. This second group can be thought of as a **specification** for the object class.

For example: The car object class might have two attributes: color, and current-speed. The color attribute would be set when the car was built, and would (hopefully) not be changing very much. It would be classified as a specification parameter. The current-speed attribute is likely to be constantly changing as the car accelerates and decelerates. It is representative of the car's current state and would be classified in the first group, the car's state space. If you took a group of cars, chances are that some of them would share the same color, but they would probably be moving around at different speeds. Rather than have each car carry around an attribute for its color, the specification attributes are split off from the car and put into a special class called a car-specification or carspec. In this way cars with identical colors can share the same specification, while still having their own state attributes like current-speed. By changing the color attribute in the specification, all the cars sharing that specification would have their color changed. This allows easy access to common parameters of an object class in one location. Rather than individually setting the color parameter for each instance of a car, the attribute can be set in just once in the spec.

This idea is instantiated in a particular class of objects known as **Spec**s. Specs also have special "smart pointers" that objects use to refer to them. These spec pointers or **SPtr** objects ensure that a given object always has a corresponding spec, and that this spec is of an appropriate type.

While specs are basically just classes that have parameters and functions that control other object's behavior, there are a couple of special properties that specs have which make them more powerful.

Often when one wants to use two different specs of the same type, it is because one spec has one parameter different than the other. For example, one spec might specify a learning rate of .01, and the other a learning rate of .001. However, these specs might very well share several other parameters.

To simplify the ability of specs to share some parameters and not others, a special system of *spec inheritance* was developed. Basically, each spec has a group on it called `children`, in which "child" specs can be created. These child specs inherit all of their parameters from the parent spec, except those specifically marked as unique to the child. These fields appear with a check in the left-hand check-box when edited in the GUI. Thus, whenever you change a value in the parent, the children automatically get this changed value, except if they have a unique value for this field, in which case they keep their current value. For a tutorial demonstration of how this works, see (see Section 4.3.7 [tut-config-running], page 41).

There are a couple of things to know about the "smart" spec pointers. These pointers have both a type field and the actual pointer to the spec. When you change only the `type` field, it will automatically find a spec of that type, and set the pointer to that. If one does not yet exist, one will be created automatically and the pointer set to it. If however you change the pointer directly to point to a different spec, and this spec is of a different type

than that shown in the `type` field, then the type will prevail over the pointer you set. Thus you have to change both the `type` and `spec` fields if you change the latter to point to a different type.

The reason for this is that the spec pointer object does not know which field you actually changed, and for the nice automatic properties associated with changing the type field to work, the need to update both the type and the spec pointer is an unfortunate consequence.

8.5 Random Distributions

Random distributions in PDP++ are handled by an instance of the class **Random**. This class has a number of functions which return a random number from the distribution named by the function. Alternately the distribution can be specified in the object itself in which case the `Gen()` function returns a value from the specified distribution. Many of the distributions require parameters which are again either passed to the specific functions or set on the Random object itself.

Random class Variables:

`Type type` Indicates the type of random variable to generate when the `Gen()` function is called. It can have one of the following values:

- `UNIFORM` A uniform distribution with `var` = half-range
- `BINOMIAL` A binomial distribution with `var` = p, `par` = n
- `POISSON` A Poisson distribution with `var` = lambda
- `GAMMA` A gamma distribution with `var` and `par` = stages
- `GAUSSIAN` A normal gaussian distribution with `var`
- `NONE` Returns the `mean` value

`float mean`
The mean of a random distribution

`float var` The "variance" or rough equivalent (half-range)

`float par` An extra parameter used for some distributions

Random Class Functions

`float ZeroOne()`
Returns a uniform random number between zero and one

`float Range(float rng)`
Returns a uniform random number with a given range centered at 0

`float Uniform(float half_rng)`
Returns a uniform random number with given half-range centered at 0

`float Binom(int n, float p)`
Returns a random number from a binomial distribution with `n` trials each of probability `p`

`float Poisson(float l)`
: Returns a random number from a Poisson distribution with parameter `l`

`float Gamma(float var, int j)`
: Returns a random number from a Gamma distribution with variance `var` and `par` number of exponential stages

`float Gauss(float var)`
: Returns a gaussian (normal) random number with a variance `var`

`float Gen()`
: Returns a random number using the distributions type `type` and the `mean`, `var`, and `par` variables on the Random object itself.

9 Projects (and Defaults, Scripts)

Projects hold all the components of a PDP simulation. Projects are essentially just an object with groups for the networks, the environments which provide the patterns, the processes to control the simulation, and the logs of the statistical data. Typically a user will load and save the project as whole, since the project represents the conceptual experiment or simulation. Also, the first action a new user must perform is to create a new project to hold the various components that will be created thereafter.

This chapter describes operations and features of the project that affect the project as a whole. This includes the following:

Basic Project Management
: There are a number of different kinds of things one needs to do with a simulation as a whole, like saving and loading it, recording what parameters were used, etc. This section gives some tips on these tasks.

The Project Viewer
: The project window contains an interactive project viewer program that enables users to conveniently view and manipulate many aspects of the project. This is especially useful for configuring processes, because they link together all the different elements of a project to actually achieve various processing goals. In addition, specs can all be viewed together in the project viewer, making it easier to manage complex sets of specs.

Startup Arguments and Controlling PDP++ with Signals
: These sections describe how to affect how PDP++ runs (e.g., turing off the graphical user interface and running it in the background). If a process is running in the background, it can be controlled by sending it signals.

Customization Through Defaults and Settings
: There are several levels of defaults and settings that can be modified in the PDP++ environment to get things to work just the way you like them. In addition to XWindow defaults, there are some global parameters, and a specialized set of object-specific defaults that are actually stored on the project itself (in the `defaults` group).

Scripts These allow the user to add all kinds of different functionality to the software by creating objects that hold different CSS scripts. These script objects are saved with the project, and can be run from a simple pull-down menu.

Select Edits
: These are special edit dialogs that allow the user to edit selected fields on any other object, all within one edit dialog. Thus, you can select the key parameters you are playing with an put them all into one SelectEdit.

9.1 Basic Project Management

Every object in PDP++ can be saved to a file, and loaded back in later from that file (see Section 8.1.3 [obj-basics-files], page 108). Typically, a user will simply save and load

the project object, since it contains all the other objects of interest, which will be saved and loaded along with it. However, it is often a good idea to save parts of the project, particularly environments, since that makes it possible to load these parts into other projects.

A saved project file is actually human readable (once it is uncompressed), and you can pull it up in your favorite text editor (if its not too big for it), and see exactly what parameters were used, etc. This is much easier to do if you just save all of your specs, which contain most of the interesting parameters in the project. Thus, saved files serve as records of exactly how your simulation was configured.

It is recommended that you save your project frequently, in the unlikely event of a program crash. However, even if you have not saved the project and it does crash, a recover file is automatically created.

These recover files are named `PDP++Recover.#.proj.gz`, where the number # of the saved file will increment to be unique in the directory. You can typically load this file back in and resume where you left off. Always check the loaded project to ensure that it looks reasonable – sometimes the crash will cause the project file to be corrupted, though this is relatively rare. Any reliable crash should be reported to the developers, along with a project file and steps to reproduce the crash. See Chapter 1 [intro], page 2.

There is now an option to record an entry in a **SimLog** file every time you do `Save As` on your project file. This SimLog file records the user, date, current and previous project file names, together with comments that the user can enter as they save the project (e.g., what parameters were manipulated, etc). This is strongly recommended, as it is often quite difficult to remember what was going on with a project when one comes back to it, even after just a few weeks away! If you decide not to use it, you can click the `use_sim_log` flag off on the little dialog that comes up as you are saving.

9.2 The Project Viewer

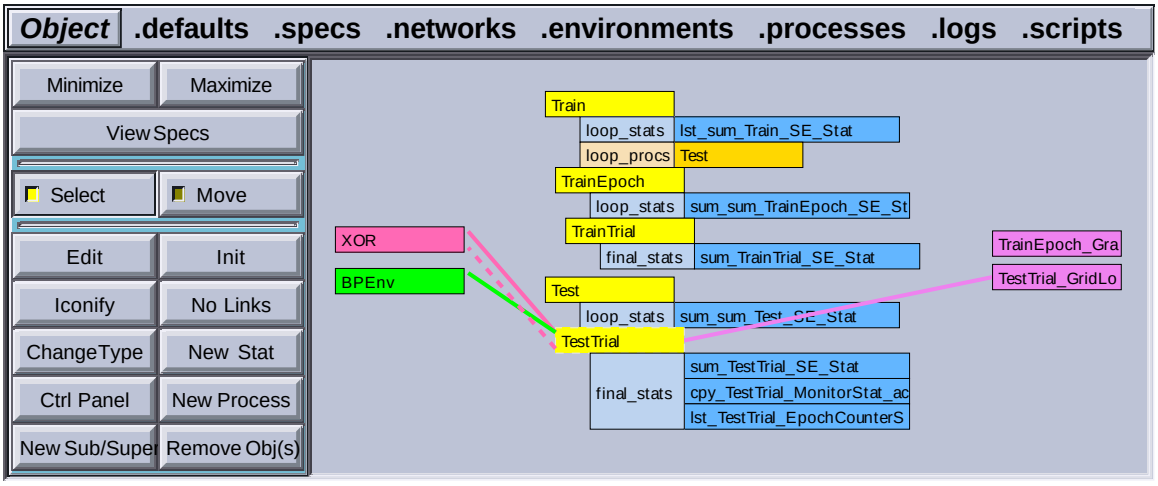


Figure: 1 The Project Viewer

In its default mode, the project viewer displays all the networks, environments, processes, and logs in your project using colored icons for each. By pressing the `View Specs` button,

it will switch to viewing the specs for this project, and then `View Project` will switch back. We begin by describing project mode, and then spec mode.

9.2.1 Project View Mode

In project mode, the following are the default colors:

Pink Networks

Green Environments

Yellow Schedule processes

Light Blue
Group of statistics within a schedule process

Slate Blue The statistics themselves – the lighter-colored stats are directly computing values while the darker ones are aggregators of others. If you click `Show Aggs` then the set of linked aggregators is displayed in aquamarine.

Red Statistics with a stopping criterion set

Wheat Group of processes within a schedule process

Gold The sub-processes within a schedule process

Brown Logs

Non-obvious actions you can perform on icons:

Single-click with left-mouse-button (LMB)
Will select or deselect item, and update the action buttons at the left of the display.

Double-click with LMB
On objects with windows (networks, environments, logs), will iconify or view the object. On processes, will iconify or de-iconify the object (an iconified process is collapsed so that you can't see all the sub-processes below it). Iconifying processes is important if you have many process hierarchies – they will not all fit in the display at once. Iconified processes are identifiable as being not colored.

Single-click with right-mouse-button (RMB)
Will bring up the edit dialog for that object.

Shift-LMB or middle-mouse-button
Extends the selection to include multiple items.

Note that because much of what happens in the project view concerns processes, you may need to read Chapter 12 [proc], page 184 to understand everything that goes on in this view.

Effects of action buttons on left side of display are (note that where multiple are listed, the same button does different things depending on what is selected – they are described in order from left-right, top-bottom).

Minimize Shrinks the display to the smallest vertical size. Note that at this size, you can re-expand the display later by hitting the top of the maximize button, which will be just visible at the bottom of the window.

Maximize Expands the display vertically to fit all of the items in the display.

View Specs
Switches the display to viewing specs (mode described below).

Select This is the standard mode – causes mouse clicks to select objects.

Move When in this mode, the mouse will move objects to different positions within their respective groups (does not work for schedule processes). For example, you can rearrange the order of statistics within a schedule process by moving them.

Edit Pulls up an Edit Dialog for selected object(s).

Init Initializes the display (this is rarely necessary, as the display is usually automatically updated when needed).

Iconify, DeIconify, Iconify All
Controls the iconification of windows or processes.

Show Links, No Links
Either shows or turns off showing the links between a selected item and other objects – the links show you how everything is connected up in the project. Links are drawn as solid lines for cases where a process uses the object for processing (e.g., The network used by the train process is shown as a pink solid line). Dashed lines indicate that the process updates the given object (e.g., the Trial process typically has a dashed line to the network, indicating that it updates it).

Change Type, Rmv Updater, Add Updater
If one object is selected, Change Type will allow you to change the type of that object. If an object that can be updated (e.g., a network) and a process object are selected, then it will allow you to either add or remove an updating link between these objects.

New Stat, Set Agg, Set Agg Link
If a schedule process is selected, New Stat will allow creating a new statistic in this object – you will be prompted for whether to make it in the loop or final stats group. Note that if you know where you want to put the new stat, and that stat group (loop_stats or final_stats) already appears in the view (because it has other stats in it), then you can just click right on the stat group and hit New Stat from there.

If a statistic is selected, Set Agg will allow you to set the kind of aggregation this statistic should use.

If two statistics are selected, Set Agg Link will set one statistic to aggregate from the other (order can be selected in a subsequent dialog).

Ctrl Panel If a schedule process is selected, this will bring up its control panel.

New Process, New Agg, Transfer Obj

If a schedule process is selected, this will allow you to create a new process in a process group (init_procs, loop_procs, final_procs) that is not currently shown in the display. Note that if the group you want is already shown in the display, then just click on it and do New Process from there. If a stat is selected, New Agg will create a new aggregator of that statistic. If a stat or non-schedule process together with a stat or process subgroup is selected, Transfer Obj will transfer the stat/process to the group. If a stat/process is selected together with a schedule process, Transfer Obj will bring up a dialog asking for which subgroup of that schedule process to transfer into.

New Scd Proc, New Process, New Stat, New Sub/Super, New Link, Transfer Obj

Transfer Obj on Project Viewer If nothing is selected, New Scd Proc will create a new schedule process.

If a stat group (loop_stats, final_stats) is selected, New Stat will create a new statistic in it.

If a process group (init_procs, loop_procs, final_procs) is selected, New Process will create a new process in it.

If one schedule process is selected, this New Sub/Super will prompt for creating a new schedule process above (super) or below (sub) the selected one in the process hierarchy.

If two schedule processes are selected, New Link will prompt for linking one process into a process group (init_procs, loop_procs, final_procs) of the other (direction is determined by a dialog, but default is that to to-be-linked process is first selected). Also see next case:

If a schedule process and a process group (init_procs, loop_procs, final_procs) on another schedule process are selected, then the schedule process will be linked into the process group with a New Link.

If a group and a non-schedule process object are selected, then Transfer Obj will transfer the object into the group.

New Proc Gp, Remove Obj(s), Remove Link

If nothing is selected, New Proc Gp will create a new process group for organizing schedule processes into groups.

Will remove object(s) that are selected, or if two objects are selected and they are linked, the link will be severed.

In summary, it should be clear that the project viewer enables you to establish linkages between different objects, and to perform detailed configuration of the process hierarchy.

9.2.2 Spec View Mode

Spec view mode shares much in common with project view mode. Specs are arranged in the order they appear in the .specs menu, left-to-right and then top-to-bottom. The default colors are:

Violet Unit Spec

Green Con Spec

Orange Projection Spec

Purple Layer Spec

Many of the the action buttons are similar to those in the project view mode, with the following special actions:

Edit

Set Spec Brings up the view of the default network in the .networks group, and applies the selected spec to whatever objects are selected in the network view. This is equivalent to performing Selections/Set XX Spec in the NetView (see Section 10.7 [net-build], page 160 for more details) where XX is the type of Spec that was selected. This provides a convenient way of applying a given spec to selected parts of the network.

Show Spec
This is like the inverse of Set Spec – it selects whatever objects in the network are currently using the selected spec.

New Child
On either a spec or the `children` group of a spec, will prompt for creating a new child spec of the selected item.

New Spec Allows one to create a new spec.

New Spec Gp
Allows one to create a new spec group for organizing specs into groups.

9.3 Startup Arguments to PDP++

PDP++ interprets a large number of startup arguments. It looks for any of the arguments that can be passed to CSS (see Section 7.4.3 [css-startup], page 90), which includes a script file that can be used to automate a set of actions to be taken in the simulator. This is particularly useful for running simulations in the background. The file '`css/include/startup.css`' provides an example of a startup file that loads in a saved project, sets some log files to record, and then runs the training or batch process. After the process is completed, the program will quit (unless the CSS argument -i was given).

The following arguments are PDP++ specific:

`-nogui` This specifies that the graphical user interface (GUI) should not be started (only the CSS interface will be present).

`-p <project_file>`
This specifies a project file to be loaded upon startup. Note that any argument within the first two args containing ".proj" will be interpreted as a startup project to run. Thus, one can type `bp++ xor.proj.gz`, and it will automatically load the project.

`-d <default_file>`
This specifies that the given default file should be used. If this parameter is not given, then a default file based on the name of the executable (e.g., bp.def

for bp++) will be used. One can also write a '`.pdpinitrc`' file containing CSS code that sets the default file with the following kind of statement: `root->default_file = "default_file"`. This can be based on the executable name by examining `argv[0]`, which contains the name of the executable being run.

In addition to the CSS arguments, PDP++ interprets all of the arguments for controlling the InterViews graphical user interface. These arguments allow one to select a different "look and feel" from the default one, which is an enhanced version of the Motif style:

`-openlook`
: The Sun OpenLook look.

`-motif` A standard Motif look. Note that the default is an enhanced SGI version of Motif, not this one.

`-monochrome`
: A Motif-like monochrome mode. It is the default if you have a monochrome monitor.

If you are logging in remotely or are somehow restricted to a non XWindows environment, you have to use the `-nogui` argument. This turns off the graphical user interface completely, and leaves you with the CSS interface.

Note that when you run simulations in the background, you will want to use `-nogui`.

Finally, there are some other InterViews arguments that might be useful:

`-nodbuf` This turns off double-buffering, which is on by default and results in smoother, flicker-free window updating, but also consumes much more display RAM from your XWindows server. Users of XTerminals in particular might want to use this option.

`-visual <dpy type>`
: By using this option and passing your default visual class (use `xdpyinfo` to obtain the visual class(es) supported by your x display) (e.g., PseudoColor for standard 8 bit (256 color) X displays), you can obtain a private color map for the PDP++ session. This can be useful if you are running out of colors in your shared colormap (e.g., because Netscape is such a color hog!).

9.4 Signals to Control a PDP++ Process

When a PDP++ process is running in the background (see previous section for instructions on how to do this), it is no longer possible to gain control of it through either the script or graphical interface. However, the process will respond to a number of signals which allow one to save the state of the process to a file. The saved file can then be pulled up in the interface and the state of the simulation examined.

Signals can be sent to a process using the unix `kill` command, with two arguments: the signal type and the process id. There are two signal types that are "user defined", `USR1` and `USR2`. We have defined `USR1` to save any networks in the process to file(s) named '`PDP++NetSave.#.net.gz`', and `USR2` to save any projects to file(s) named '`PDP++Project.#.proj.gz`'. The `ALRM` (alarm) signal will also save project files.

Any of the "lethal" signals like `SEGV` and `BUS` which cause the process to crash result in an attempt to save the current state of any open projects in `PDP++Recover.#.proj.gz` files. Thus, the user will not typically lose any work even when (or if) the software crashes. Note that the number # of the saved file will increment to be unique in the directory.

To kill a process without getting one of these recover files, use the strongest kill signal, `KILL` (signal number 9), which will bypass the saving of a recover file.

9.5 Customization Through Defaults and Settings

The different kinds of defaults and settings that can be used to customize the behavior of PDP++ are covered in this section. In addition to these defaults, there are some XWindow resources that can be set in the user's '`.Xdefaults`' file, which are covered in Section 6.6.2 [gui-settings-xdef], page 66.

9.5.1 Settings and the .pdpinitrc and .cssinitrc Files

Every time the PDP++ software starts up, it looks for a '`.cssinitrc`' and a '`.pdpinitrc`' file in the user's home directory. These files contain CSS script code that can set various default settings, set aliases for CSS commands, etc (see also Section 7.4.3 [css-startup], page 90). A list of the different settings that can be put into the '`.pdpinitrc`' file is given in Section 6.6 [gui-settings], page 63. These settings are actually "static" members located in the **taMisc** object, and are set in CSS as in the following example:

```
taMisc::display_width = 90;
```

Another thing that you might want to put in the '`.pdpinitrc`' is your default color specification:

```
.colorspecs.SetDefaultElName("C_ColdHot");
```

Finally, if you have put a set of PDP++ files (like the defaults files described in the next section, for example) in a directory somewhere, you can add this directory to the list of those automatically searched when loading CSS script files and defaults files. This is done with the `include_paths` member of the **taMisc** object that holds all of the other settings parameters. It is an array of strings, and you simply add a new string to it to add a new path:

```
taMisc::include_paths.AddUnique("/home/mach/me/pdp++/defaults");
```

Note that the environmental variable `PDPDIR` should be set to the directory in which PDP++ was installed in order to have the default include paths be correct. If `PDPDIR` is not set, the default is '`/usr/local/pdp++`'.

9.5.2 Project Object Defaults (TypeDefaults)

The `.defaults` group on the Project object contains a number of objects called **TypeDefault**s. These objects can represent the default values of fields within a whole range of possible objects that could be created in the project. These defaults are used to make sure that the appropriate types of objects are created for particular algorithms. This is done through the use of default files, which are saved groups of TypeDefault objects.

There are a number of different default files saved in the '`defaults`' directory, one of which is automatically loaded into the `.defaults` group of the project when it is created. Which one is used is based on the name of the PDP++ executable (i.e., bp.def for bp++). When PDP++ starts, it checks the name of the executable, and puts the appropriate defaults file name in the `default_file` field of the PDPRoot object. Changing this field will change the default file that will be used when a new project is created.

The user can create their own defaults files, to have things come up just the way they like them. It is recommended that you start with the basic defaults appropriate to a given algorithm that you will be using. Simply edit an existing or create a new **TypeDefault** object in the `.defaults` group of a project that has the right initial defaults in it. There is one **TypeDefault** object for each different type or class of object whose default values are being set. Derived classes (see Section 8.1.1 [obj-basics-obj], page 107) automatically inherit default values set on parent classes.

The TypeDefault object has the following members:

`TypeDef* default_type`
: This is where you pick the type of object you want to set the default values of. Note that many types are descended from **taNBase**, so look in its submenu. Be sure to press *Apply* after selecting a type, so that the object knows what type of `token` to make.

`TAPtr token`
: This field contains a token of the type selected in `default_type` (remember to hit *Apply* first before editing this!). Default values are set by simply editing this object and setting the values of the fields as you want them to be by default. Only those fields that have a check in the leftmost checkbox next to the field will have default values saved. Thus, make sure that box is checked when you make changes to the fields. Then, *Ok* the dialog, and be sure to hit *Apply* on the **TypeDefault** object itself. This causes the specific changes you specified to be registered in the `active_membs` group, which only records the values of those members that were checked.

`taBase_Group active_membs`
: The type default values are actually saved as name-value pairs (using the **NameValue** object), where the name is the name of the member, and the value is a string representation of its value. This is a group of such name-value pairs for the checked fields. While you can enter things directly in here, it is recommended that you use the `token` to do it instead.

When you have created a set of defaults, save them using the *Save As/All* menu item in the `.defaults` group. If you have the appropriate permissions, it is easier to put the defaults file in the global '`/usr/local/pdp++/defaults`' directory. However, it is cleaner to keep your defaults in your own directory. Since your home directory is searched automatically for default files, you could put the default files there. A better solution is to create a special directory to hold your defaults, and add this directory to your `include_path` in your '`.pdpinitrc`' file (see Section 9.5.1 [proj-settings], page 126).

9.6 Project Scripts

Scripts are an important component of a simulation, especially when things get a bit more complicated than a simple XOR example. They allow you to perform routine actions automatically, which can save a lot of mouse clicks, and makes some things possible that would be prohibitively time-consuming to do by hand (e.g., switching one's environment to use -1 to +1 values instead of 0 to 1 values). Since scripts can be created just by recording actions taken in the GUI, it is relatively easy to get started with them.

A PDP++ **Project** contains a special place to put different scripts that are associated with a given project. Thus, there is a **Script** object, which has a corresponding script file that contains CSS script code (see Chapter 7 [css], page 71). These **Script** objects provide a way of managing and running multiple different CSS script files, whereas the script window that is present at startup can only hold one script file at a time.

The script objects are found in the `.scripts` group on the project. Editing a **Script** object results in a little "control panel" with several buttons that manipulate the script in different ways. An existing script can also be run directly by selecting that script from the *Run* menu of `.scripts`.

The **Script** object has the following member fields:

`taFile script_file`
: This contains the name of the script file to associate with this object. The name field of the object will automatically reflect the name of the script file selected (minus the .css extension). Note that the script file can be edited by using the *Edit* menu option, which pulls up the editor given by the `EDITOR` environmental variable.

`String script_string`
: You can enter an entire script in this string field, and run it instead of reading the script from the `script_file`. If `script_string` is non-empty, it is used instead of the file, even if the file is open.

`bool auto_run`
: Check this option if you want the script to be run automatically when the project is loaded. This is can be used for example to automatically build and connect a large network which is saved without units or connections to make things faster and the file smaller. An example script which builds the network is '`css/include/build_network.css`'.

  If you startup the pdp++ program with the project name on the command line, then the project will load over and over again if there is an error with a script that is set to run upon loading the project using `auto_run`. The solution is to load the project from the Root menu, and to immediately move the mouse over the xterminal window, and type a Return into that window when the error occurs. Then, you can debug the problem with the script at the point where the error occurred.

`String_Array s_args`
: These are arguments that the script file can access. The argument values can be set in the script object, and used to modify the way the script behaves, etc.

Scripts that use these arguments should have the meaning of the arguments documented near the top of the script file.

The following button actions are available in the edit dialog of the **Script** object:

`Run()` This runs the script. A script must already be loaded and compiled.

`Record()` This causes future GUI actions to be recorded to this script. Thus, one can construct a script record of a series of gui actions, and play them back by compiling the resulting script, and running it. The script file can be edited and, for example, a `for` loop wrapped around a set of actions to perform them multiple times, etc.

`StopRecording() (StopRec)`
This stops the recording of GUI actions to this script. Only one script can record at a time, so if you press `Record` on a script, then all others will automatically stop recording.

`Interact()`
This allows you to interact with the script through the CSS shell window (the one you started the PDP++ program from). The prompt will change to the name of the script you are working with. All of the CSS commands can then be used to debug and run the script (see Chapter 7 [css], page 71). Note that to exit out of this css shell, you do a `quit` in CSS, which will bring you back to the initial PDP++ executable prompt.

`Clear()` This clears (empties out) the script file. This cannot be undone, so make sure you are ok losing all of the stuff in the script file! It is primarily used when recording scripts, when you want to start over from the beginning.

`Compile()`
This will re-compile the script file. Any time you make changes to the .css file associated with the **Script** object, these changes need to be re-interpreted by CSS into something that can actually be run (see Section 7.2.1 [css-tut-run], page 72).

There are a number of useful script files in the '`css/include`' directory of the PDP++ distribution. These files contain documentation as to what they do.

9.7 Select Edit Dialogs

The **SelectEdit** object consolidates multiple different fields and functions from different objects into one edit dialog. Typically, one selects the fields by editing the object in question, and using the **Object/SelectForEdit** menu option to select the field to be edited, or **Object/SelectFunForEdit** to select a function to be accessible from the select edit dialog.

The select edit dialog always has as its first line a `config` field, which contains configuration information for the select edit dialog itself. The `auto_edit` toggle specifies whether the select edit dialog is opened automatically whenever the project is opened. The `mbr_labels` contains extra user-specified labels to append before the field member names, to specify which object they came from. The `meth_labels` contains similar such labels for methods (functions).

The `SelectEdit` menu on the edit dialog contains functions for Removing and Moving fields and functions within the edit dialog. The `New Edit` function closes the current edit dialog and re-opens it. Use this if you have edited member or method labels, and want to see them reflected in the edit dialog.

10 Networks (Layers, Units, etc)

Every simulation is built around a network. Networks contain layers of units, with connections between the units. PDP++ provides objects at every level of structure from the **Network** down to the **Connection** from which neural network models can be constructed. While particular algorithms will define their own sub-classes of these basic object types the essential properties of these objects are relatively constant across different algorithms. This chapter describes these basic properties.

The user's primary interaction with the PDP++ software is through the **NetView**, described in Section 10.6 [net-view], page 149. This provides a graphical display of the network structure, and a means of creating and modifying the different elements of the network.

For a tutorial introduction on how to use the **NetView** to build and connect a network, see Section 4.3 [tut-config], page 33.

10.1 The Network Object

The network object is basically a container for all the things that go in it. Thus, it has a group of layers, which then contain units, projections, etc. In addition, each network also has an epoch counter and a variable controlling the default layout of the layers. The epoch counter reflects the number of *training* epochs the network has seen (not testing). Finally, the network is instantiated in the user interface by one or more **NetView**s, which provide a window with menu actions, tools, etc. to operate on the network.

Also see the following information about Networks:

`Layer_MGroup layers`
: This is the group of layers that have been created in the network. The layers then contain all of the remaining substructure of the network.

`int epoch`
: The epoch counter indicates how many epochs of training the network has been subjected to. This counter is incremented by the training processes which act upon it, see Section 12.3.3 [proc-levels-epoch], page 192.

`bool re_init`
: This flag is set by training schedule process to indicate whether the network should be reinitialized when the process is initialized Section 12.3.2 [proc-levels-train], page 191.

`LayerLayout lay_layout`
: This variable can be set to either `TWO_D` or `THREE_D`. It controls the default positioning of the layers when they are created, as well as the skew of the units when they are displayed. `TWO_D` networks are arranged in one big X-Y grid, while `THREE_D` networks simulate a three-dimensional space where successive layers are located at successively higher Z coordinates, and units within a layer are arranged in an X-Y plane.

`Usr1SaveFmt usr1_save_fmt`
: You can control how the network is saved when it receives the `USR1` signal (see Section 9.4 [proj-signals], page 125) with this – the full network or just the weights

`PThreadSpec pthread`
: This specifies parameters for parallel processing (SMP) of the network. Number of processors (threads) is specified by the **n_threads** parameter. This works by compiling a list of layers for each thread to process, based on the number of connections and units in the layers (separate list for each). The allocation algorithm tries to distribute the computation as evenly as possible. This is relatively fine-grained parallelism, and due to the overhead involved in managing the threads, it only speeds up relatively large networks (i.e., > 50 units per layer, >= 4 layers). Speedups can by upwards of 1.67 times as fast on dual pentium III systems. You can also restrict parallelism to only connection-level computations by turning off **par_units** — unit level speedup is typically quite quite small (but measurable..).

The network also has the following functions which can be activated by the corresponding menu selection in the **NetView** (**using the left hand menu**).

In the *Object* menu:

`ReadOldPDPNet(File input_file, bool skip_dots)`

`Copy_Weights()`
: Copies the weights from another similarly-configured network. By duplicating a network and then later copying weights back from it, one can restore weight values to known values while tinkering with various parameters affecting network learning.

`WriteWeights(File output_file)`
: Outputs bias and receiver weight values in a unit by unit format to a file (includes comments). NOTE: This is not the same format as the old pdp software.

`ReadWeights(File input_file)`
: Loads in the bias and receiver weight values from a file created with the WriteWeights function above. Reads in a "filename.net" type file format from the old pdp software. *skip_dots* indicates whether to skip over "." values in the "network:" weight matrix section or to create zero valued weights instead. This function will attempt to use and extend existing network structure if it exists and create new network structure if necessary. Currently (v1.2) it does not handle "Bias:" sections or "LINKED" weight constraints.

In the *Actions* menu:

`Build()` Create units in all the layers according to the size and shape of the layers. If units already exist, it makes sure they are of the right type, and arranged within the geometry of the layer. This is accessed through the *Build All* button of the **NetView** (see Section 10.6 [net-view], page 149).

`Connect()`
: Create connections on all the units according to the projections on the layers. The projections define a pattern of connectivity, and the connections actually flesh this pattern out by individually connecting unit to unit. Note that any existing connections are removed before connecting. This is accessed through the *Connect All* button of the **NetView** (see Section 10.6 [net-view], page 149).

`Check Types()`
: Checks to make sure all the objects and specs in the network are mutually compatible.

`Fix Prjn Indexes()`
: Fixes the other_idx indexes on the connection groups — CheckTypes might tell you you need to run this if your network was not properly connected.

`RemoveCons()`
: Remove all the connections on the units in the network. Like `RemoveUnits` this is useful for reducing the size of the network for saving.

`RemoveUnits()`
: Remove all the units from the network. This can be useful for saving large networks, since all of the relevant structural information for rebuilding the network is typically present in the layers and projections. Thus, one can save the "skeleton" and then simply press *Build All* and *Connect All* when the network is reloaded.

`InitState()`
: Initialize the state variables on the units. This means activation-like variables, but not the weights.

`InitWtState()`
: Initialize the weight state information on the connections in accordance with their **ConSpecs**. This also resets the epoch counter to zero.

`TransformWeights(PreProcessVals trans)`
: Applies given transformation to weights. Possible transformations include basic arithmetic operators (e.g., scaling via multiplication), and absolute value, thresholding, etc.

`AddNoiseToWeights(Random noise_spec)`
: Adds noise to weights using given noise specification. This can be useful for simulating damage to the network.

`PruneCons(PreProcessVals pre_proc, CountParam::Relation rel, float cmp_val)`
: Removes connections that (after a pre-processing transformation, e.g. absolute-value) meet the given relation in comparison to compare val (e.g., LESSTHANOREQUAL to some value).

`LesionCons(float p_lesion, bool permute)`
: Removes connections with probability p_lesion. If permute is true, then a fixed number of weights will be lesioned, where this number is equal to the probablility times the number of weights. Othewise, the actual number of weights lesioned will vary based on the probability.

`LesionUnits(float p_lesion, bool permute)`
: Removes units with probability p_lesion. If permute is true, then a fixed number of units will be lesioned, where this number is equal to the probablility times the number of units (on a layer-by-layer basis). Othewise, the actual number of units lesioned will vary based on the probability.

`TwoD_Or_ThreeD(LayerLayout layout_type)`
: Reposition the units and layers in either a 2D or 3D configuration.

`GridViewWeights(GridLog* grid_log, Layer* recv_lay, Layer* send_lay, int un_x, un_y, wt_x, wt_y)`
: Plots the entire set of weights into the recv_lay from the send_lay in the grid log specified (NULL = make a new one). The un_x, un_y parameters can specify a more limited range of receiving units (-1 means entire layer), and the wt_x, wt_y similarly specify a more limited range of sending units (weights).

These other functions of the network might be useful to those writing scripts or programming in PDP++:

`ConnectUnits(Unit* receiving_unit, Unit* sending_unit);`
: Creates a connection from the sending_unit to the receiving_unit. A custom projection between the units' layers is created if necessary

Finally there are a number of other functions that can be found in '`src/pdp/netstru.h`' which are useful for programming. In general the Network has a function corresponding to one that is performed on a lower-level object like a unit, and this function on the network simply calls the corresponding one on all of its layers, which then call the one on all of their units, etc.

10.1.0.1 Distributed Memory Computation in the Network

The Network supports parallel processing across connections, where different distributed memory (dmem) processes compute different subsets of connections, and then share their results. For example if 4 processors were working on a single network, each would have connections for approximately 1/4 of the units in the network. When the net input to the network is computed, each process computes this on its subset of connections, and then shares the results with all the other processes.

Given the relatively large amount of communication required for synchronizing net inputs and other variables at each cycle of network computation, this is efficient only for relatively large networks (e.g., above 250 units per layer for 4 layers). In benchmarks on Pentium 4 Xeon cluster system connected with a fast Myrinet fiber-optic switched network connection, networks of 500 units per layer for 4 layers achieved *better* than 2x speedup by splitting across 2 processors, presumably by making the split network fit within processor cache whereas the entire one did not. This did not scale that well for more than 2 processors, suggesting that cache is the biggest factor for this form of dmem processing.

In all dmem cases (see Section 12.3.3.1 [proc-epoch-dmem], page 193 for event-wise dmem) each processor maintains its own copy of the entire simulation project, and each performs largely the exact same set of functions to remain identical throughout the computation process. Processing only diverges at carefully controlled points, and the results of this divergent processing are then shared across all processors so they can re-synchronize with each other. Therfore, 99.99% of the code runs exactly the same under dmem as it does under a single-process, making the code extensions required to support this form of parallel processing minimal. This was not true for the pthread (parallel thread) model that was used in earlier releases – it is now gone.

The main parameter for controlling dmem processing is the `dmem_nprocs` field, which determines how many of the available processors are allocated to processing network connections. Other processors left over after the network allocation are allocated to processing event-wise distributed memory computation (see Section 12.3.3.1 [proc-epoch-dmem], page 193 for information on this). The other parameter is `dmem_sync_level`, which is set automatically by most algorithms based on the type of synchronization that they require (feedforward networks generally require layer-level synchronization, while recurrent, interactive networks require network-level synchronization).

The one area where dmem processing can cause complications is in networks that use shared/linked weights, such as those that can be created by the **TesselPrjnSpec** projection spec (see Section 10.3.3.2 [net-prjn-tessel], page 139). If shared weights end up on different processors, they cannot be shared! If these weights happen to be shared across unit groups, then you can set the `dmem_dist` parameter in the corresponding **Layer** to `DMEM_DIST_UNITGP`, which will distribute connections across unit groups such that the first unit in each unit group are all allocated to the first processor, the second to the second, etc. In this case, they can all share connections.

10.2 Layers and Unit Groups

Layers are basically just groups of units, but they also constitute the basic elements of connectivity as specified by the projections (see Section 10.3 [net-prjn], page 137). Thus, one can specify that a whole group of units should be connected in a particular fashion to another whole group of units, instead of going one-by-one through the units and telling each one individually how to connect to other units. Note that the model used in PDP++ is the *receiver* model – a layer contains the projections it *receives* from other layers, not the ones it sends to them.

Further, in certain kinds of learning algorithms, particularly those with a self-organizing character, the Layer plays a computational role in that it implements competition among units, for example (see Chapter 16 [so], page 256). In these algorithms, the **Layer** will have an accompanying **LayerSpec** that contains the parameters governing the computation that occurs in the layer. However, in standard backpropagation and constraint satisfaction algorithms (Chapter 14 [bp], page 226, Chapter 15 [cs], page 245), the LayerSpec is not used.

Layers have the following variables:

`int n_units`
: Number of units to create with Build command. Entering a value of 0 will cause the n_units to be computed based on the current geometry (i.e., n_units = x * y).

`Geometry geom`
: Specifies the layer's 3D geometry for its units. It is used for display purposes, and for computing a default `n_units`. Note that a `z` value of greater than 1 means that multiple sub-groups of units will be created in the layer, each of which having x * y units (by default).

`Geometry pos`
: Specifies the layer's 3D position relative to the network. Used for display purposes and potentially for algorithms that depend on physical distance information.

`Geometry gp_geom`
: This is the geometry of the sub-groups of units within a layer (only applicable if `geom.z > 1`). Groups will be arranged in the x,y geometry given by this parameter (i.e., `x * y` should be `>= geom.z`).

`Projection_Group projections`
: The group of projections which specify the connections that this layer receives from other layers.

`Unit_Group units`
: The group of units within the layer. The type of units which are created is determined by the `el_typ` on this group (see Section 8.2.2 [obj-group-variables], page 111). Note that sub-groups of units can be created, as given by the `geom.z` parameter, and as described in greater detail below.

`UnitSpec_SPtr unit_spec`
: The default unit specification for units created in this layer. This is applied whenever the Build function is performed.

`bool lesion`
: When set to `true`, this layer is inactivated from all processing, as if it was not there in the first place. This makes it possible to selectively pre-train certain layers within the network, for example.

`Unit:ExtType ext_flag`
: Indicates which kind of external input the layer last received. Whenever input is applied to the network from an environment, the affected layer's flag is set, so that when it comes time to clear the unit flags, or perform any other input-specific processing, only relevant layers need to be processed.

The following layer functions are available, either in the layer edit dialog or in the CSS script language:

`Build()`
: Create `n_units` in the layer and arrange them according to the `geometry`. If units already exist, they are enforced to be of the type of the unit group. Thus, the best way to change the type of units in the layer is to change the `el_typ` of the units group and then do a Build() (see also Section 10.1 [net-net], page 131).

`Connect()`
: Actually creates the connections into each of the units in the layer according to the projections on this layer. Note that any existing connections are removed before connecting.

`RemoveCons()`
: Removes any existing connections into units in the layer.

`DisConnect()`
: This removes all the projections (and connections) that this layer receives from other layers, and all the projections that other layers receive from this layer.

Thus, it is stronger than `RemoveCons` in that it removes the projections as well as the connections.

`Copy_Weights(Layer *)`
Copies the weights from the other layer including unit bias weights.

`SetLayerSpec(LayerSpec *)`
Sets the layer specification for this layer to be the given one.

`SetUnitSpec(UnitSpec *)`
Sets the unit specification of all units in the layer to be the given one, and makes that the default unit spec for any new units that would be created in the layer.

`SetConSpec(ConSpec *)`
Sets the connection specification of all projections in the layer to be the given one.

Note that the layer also has instances of other functions that are present on the network, such as `InitWtState` and `InitState`.

The **Unit_Group** provides another level of organization between the layer and the units within the layer. This can be useful for cases where different groups of units play specialized roles, but have otherwise the same connectivity to other layers (which is what makes them part of the same layer instead of being different layers). This level of organization has been used to implement independent sub-pools of activation competition within a larger layer, for example.

Unit groups are created as sub-groups within the `units` member of the layer. The `z` value of the layer's geometry specification indicates how many groups to create, and each of them has `n_units` units arranged in the `geom.x, geom.y` geometry as specified in the layer. However, once created, the unit groups can be individually re-sized, and they have their own `n_units`, `geom`, and `pos` variables. To have a unit group always use the settings on the layer, the `n_units` should be set to 0.

10.3 Projections

Projections represent a layer's connectivity with another layer. They serve as a "template" for individual connections between units in the two layers, thus simplifying the specification of general patterns of connectivity. The connectivity of the projection is specified in terms of the layer this projection's layer is receiving from. Thus if you had an input layer connected to a hidden layer, then the hidden layer would have a projection with its `from` field set to the input layer.

In addition to the `from` field, projections have a **ProjectionSpec** which determines the connectivity patterns to use when creating the actual connections between individual units. There are a number of different forms of connectivity that can be specified with the different **ProjectionSpecs**, from the simple full connectivity to different forms of random, one-to-one, and "tesselated" or repeated patterns of connectivity.

The projection object itself is primarily concerned with specifying *where* to receive connections from, and what kinds of connection objects to create. The **ProjectionSpec** is responsible for determining the *pattern* of connectivity.

10.3.1 The Projection Class

The projection object is primarily concerned with specifying *where* to receive connections from. Also, it determines what type of connections (and connection groups and connection specs) should be created:

`PrjnSource from_type`
: Type of the projection source. This can have one of the following values:
  - NEXT: Receive connections from the next layer in network.
  - PREV: Receive connections from the previous layer in network.
  - SELF: Receive connections from the same layer this projection is in.
  - CUSTOM: Receive connections from the layer specified in the projection.

`Layer* from`
: The layer this projection receives from. This is set automatically if `from_type` is not set to `CUSTOM`.

`ProjectionSpec_Sptr spec`
: Points to the **ProjectionSpec** which controls the pattern of connectivity for this projection.

`TypeDef con_type`
: The type of **Connection** to create when making connections.

`TypeDef con_gp_type`
: The type of connection group to create when making connections.

`ConSpec_SPtr con_spec`
: The connection specification to use for the connections.

The Projection class has a number of member functions, most of which have the same function as those defined on the Layer and the Network. Refer to Section 10.2 [net-layer], page 135 and Section 10.1 [net-net], page 131 for further details. The following are specific to projections:

`Copy_Weights(Projection* src)`
: Copies the weights values from an equal sized projection to the weight values of the connections on this Projection.

`ApplyConSpec()`
: Sets the conspec of the all the connections for this projection to the projection's `con_spec` without rebuilding them.

10.3.2 The Projection Specification

The **ProjectionSpec** class describes the patterns of connectivity between units in the two layers involved in a projection. The base ProjectionSpec class is a parent class for the more specific Projection Spec classes which are actually used (see Section 8.1 [obj-basics], page 107). Nonetheless, it provides the basic functions and variables common to all Projection Specs.

The projection spec actually implements many of the functions associated with the projection, so functionality can be modified just by changing the spec.

There are two variables that are common to all projection specs. One is the `self_con` flag. This indicates if self-connections from a unit to itself should be created in `SELF` projections. The other is `init_wts`, which indicates whether the connection weights should be initialized from this projection spec (see Tessel and Random, below) or via the ConSpec (which is the default, see Section 10.5 [net-con], page 146).

10.3.3 Specialized Types of Projection Specs

There are a number of different types of projection specification types which implement different kinds of connectivity patterns. They are described below.

10.3.3.1 Full Connectivity

The **FullPrjnSpec** is the most commonly used type of projection spec. It creates full connectivity between units. There are no other parameters controlling its behavior, and not much else to say.

10.3.3.2 Tesselated (Repeated) Patterns of Connectivity

The **TesselPrjnSpec** connects two layers using tesselations (repeated patterns) of connectivity. These patterns are defined in terms of a "receptive field" which is a two-dimensional pattern of connectivity into a given receiving unit. This pattern is defined in terms of two-dimensional X,Y offsets of the sending units relative to the position of a given receiving unit. Thus, there is an assumed correspondence between the layout of the receiving and sending units.

Which receiving units get this receptive field is determined by offset and skip parameters which allow one to have different receptive fields for the even and odd units, for example.

The center of each receiving unit's receptive field over the sending layer is either in direct correspondence with the coordinates of the receiving unit within its layer, or there can be a scaling of the coordinates based on the relative sizes of the two layers, so that the receiving units evenly cover the sending layer. Also, contiguous receivers can share the same effective receptive field center by using the `recv_group` parameters.

In addition, since the **TesselPrjnSpec** creates repeated versions of the same connectivity pattern, it is a natural place to implement weight sharing. There is a choice on the spec, `link_type`, that can establish shared weights among all units in the same `recv_group` (`GP_LINK`), or each unit has the the same weights (`UN_LINK`).

Finally, there are a number of functions which automatically generate receptive fields according to simple geometric shapes, or create receptive fields from patterns of selected units in the **NetView** (see Section 10.6 [net-view], page 149). Further, the weight values for these connections can be initialized to specified values (use the `init_wts` and set the `wt_val` values), and distance-based values can be computed automatically using functions. These are described below.

The spec has the following parameters:

`TwoDCoord recv_off`
: The offset (XY) in the layer for the start of the receiving units. Units before this offset do not get connections.

`TwoDCoord recv_n`
: The number of receiving units in each dimension (XY). The default values of -1 means use all the receiving units (after the offset).

`TwoDCoord recv_skip`
: The number of receiving units to skip in each dimension (XY). Thus, one could have one spec controlling the even units and another for the odd units in a given layer. NOTE: this is ignored when `GP_LINK` is used, for technical reasons.

`TwoDCoord recv_group`
: The number of receiving units to group together with the same receptive field center in each dimension (XY). Thus, one can have groups of units with identical receptive fields.

`bool wrap` Indicates whether or not to wrap coordinates at the edges of the sending layer (otherwise it clips at the layer edges).

`bool link_type`
: Indicates whether and how to link together receiving weights:

  `NO_LINK` Each unit has its own weights, as is normally the case.

  `GP_LINK` Shares weights among an entire `recv_group` of units, where the 1st unit in each group has the same weights, etc.

  `UN_LINK` The same weights are shared between all units (each unit has the same weights). Also see `link_src`.

`TwoDCoord link_src`
: The index of the receiving unit that should serve as the "source" unit for unit linked weights. If sending coordinates are not being wrapped, then the first unit in the receiving layer will likely not have the full complement of connections, since some of them will have been clipped, so this allows a unit that has the full complement of connections to be indicated as the source, which has to have all the possible connections.

`FloatTwoDCoord send_scale`
: Scales the coordinates of the receiving unit in order to determine the coordinates of the center of the receptive field in the sending layer. Thus, if in a given dimension (X or Y) there are only four units in the receiving layer and 8 units in the sending layer, one might want to use a scale of 2 in that dimension so the receivers will cover the whole sending layer.

`TwoDCoord send_border`
: A border (offset) that is added to the receiving unit's coordinates (after scaling, see above) in order to determine the coordinates of the center of the receptive field in the sending layer.

`TessEl_List send_offs`
: A list of offsets of the sending units. These offsets are relative to the center of the receiving unit's receptive field in the sending layer, computed as described above. Each offset is a member of the class TessEl which has the members:

  `TwoDCoord send_off`
  : The offset from the center of the receptive field.

  `float wt_val`
  : The value to assign to the weight of the connection. These weight values are given to the weight upon creation of the connection, but if the **ConSpec** performs its own initialization of the weights, they will be lost. Also, if learning is taking place, the only way to reinstate these values is to reconnect the network.

The functions that make particular receptive fields are as follows:

`MakeEllipse(int half_width, int half_height, int ctr_x, int ctr_y)`
: Constructs an elliptical receptive field from the given parameters. The ctr_x and y specify the center of the receptive field. For example, half_width = half_height = 2, ctr_x = ctr_y = 0, gives a circle from -2 to 2 in x and y.

`MakeRectangle(int width, int height, int ctr_x, int ctr_y)`
: Constructs a rectangular receptive field from the given parameters. ctr_x and _y are as in MakeEllipse (e.g., specifying a width, height of 5 and ctr_x, _y of 0 gives x and y coordinates from -2 to 2.

`MakeFromNetView(NetView* view)`
: Uses the currently selected units in the **NetView** to create a receptive field. First select a receiving unit, which establishes the center of the receptive field (this should be in the receiving layer). Then, in the sending layer, select the receptive field pattern. All units must be selected (use multiple-select) before this function is called.

`WeightsFromDist(float scale)`
: Initializes the `wt_val` value of each element of the receptive field according to its distance in offset coordinates, scaled by the given scale parameter. Note that `init_wts` must be set for these weights to be used in initializing the weights.

`WeightsFromGausDist(float scale, float sigma)`
: Like the above, but it uses a Gaussian function of the distance, with a standard deviation equal to the given sigma parameter.

10.3.3.3 Random Patterns of Connectivity

The **UniformRandomPrjnSpec** specifies a uniform random pattern of connectivity.

`p_con` specifies the overall probability of connectivity — a connection is made with a given sending unit with this probability. The `permute` flag indicates that a randomly-ordered list of sending units is created, and p_con * n_units are selected for connecting — creates exactly the same number of connections per receiving unit. `same_seed` specifies that this projection saves the random seed used for creating connections, so tht the pattern

is the same every time (useful for being able to read in weight files for randomly connected networks).

The **PolarRndPrjnSpec** creates randomized patterns of connectivity as a function of distance and angle between sending and receiving unit. Distance and angle are computed from the center of a receiving unit's receptive field in the sending layer, which, as with the **TesselPrjnSpec** described above, can be computed in different ways. Two different random functions control the distribution of connectivity in distance and angle.

The `dist_type` field controls how the distance is computed, as follows:

`XY_DIST` Just a simple distance function using the receiver's coordinates in the sending layer. This works fine when both layers are the same size.

`XY_DIST_CENTER`
The receiver's coordinates are transformed relative to the center of the sending layer. This makes the distance distribution symmetrical.

`XY_DIST_NORM`
The receiver's coordinates are normalized by the total size of the sending layer.

`XY_DIST_CENTER_NORM`
The receiver's coordinates are normalized by the total size of the sending layer, and are computed relative to the center of the layer. This will result in a reasonable distance measure even when the two layers are of different sizes.

The `rnd_dist` and `rnd_angle` are **Random** (see Section 8.5 [obj-random], page 117) classes that specify the distributions and associated parameters for connection distance and angle from the receiving unit. Distance is scaled as above, and angle is done on a 0-1 scale (i.e., the random number is multiplied by 2pi). `p_con` determines how many connections are made. A target value of p_con * n_units in the sending layer is used, and connections are attempted, rejecting attempts to reconnect to an existing connection, until `max_retries` such rejections have been made. Thus, for very tight distributions, the same units will be selected again and again, and it may be impossible for `p_con` different connections to be established. In this case, a warning message is issued.

The `wrap` flag determines if the units are treated as one big wrapping-around surface, or if it is clipped at the edges of the layer.

`same_seed` functions as on the UniformRandomPrjnSpec

10.3.3.4 Unit_Group Based Connectivity

There are a couple of classes that specifically pay attention to the sub-groups of units within a layer, if these have been created (the other types of projections just ignore this level of structure).

There is a sub-class of the full projection called the **GpFullPrjnSpec**, which does the same thing the full prjn spec, but creates separate connection groups based on the sending and receiving unit-group structure. Thus, the result is full connectivity, but this is broken down so separate unit sub-groups can be treated separately (e.g., if there were a weight-based competition between the units in a sub-group, or between sub-groups). The `n_con_groups`

parameter determines whether there is one con-group per `RECV_SEND_PAIR`, or just one per `SEND_ONLY`, which is one con-group per unit-group on the sending layer.

The **GpOneToOnePrjnSpec** connects unit groups in two layers in a one-to-one fashion, much as the OneToOnePrjnSpec connects units in a layer in a one-to-one fashion.

The **GpOneToManyPrjnSpec** connects one or more sending groups to all receiving groups. It can greate these connections in a number of separate connection groups, or all in one group, depending on the `n_con_groups` parameter (see GpFullPrjnSpec above). Note that the `recv_start` parameter is ignored, and only the `send_start`, which determines which sending group to start with, and the `n_conns`, which determines how many sending groups to use beyond the start, are relevant.

10.3.3.5 Miscellaneous other Projection Types

The **OneToOnePrjnSpec** simply connects units in a one-to-one fashion. This is typically for the entire set of units, but can be controlled by setting the `n_conns`, `recv_start` and `send_start` parameters, which specify the total number of connections to make and starting offsets.

The **SymmetricPrjnSpec** makes receiving connections to units in the sending layer that are already receiving connections from units in the receiving layer. Thus, it makes symmetric connectivity where another projection spec has defined the pattern from the other set of units. Note that the other projection spec must be associated with a layer that comes before the one this spec is on, otherwise it will not have any connections to copy from.

The **ScriptPrjnSpec** uses a CSS script to create the connections. It contains an `s_args` array of Strings which are passed to script as arguments (see Chapter 7 [css], page 71). Any arbitrary form of connectivity can be described by writing the appropriate script. Several useful functions on the unit are available for making connections, including `ConnectFrom`, which takes the sending unit and the projection as arguments, and returns the connection and the two connection groups associated with it.

The **CustomPrjnSpec** is used when the connectivity between units is hand assembled. Thus it does not specify a connectivity function, and therefore performs no actions when the Build command is called on a Network.

The **LinkPrjnSpec** does not create any connections itself. Instead, it turns existing connections into linked connections. The connections to be linked are specified by the layer name and unit index for both the sending and receiving units. The connection function then finds the connection that connects these two units, and links it in with the other ones. The first connection specified is the "owner" of the connection, and its weight values are the ones that are used. The `links` member is the list of connections to be linked together. This type of projection spec, since it does not create any projections itself, is typically assigned to a "dummy" self projection on one of the affected layers.

10.4 Units

Units are the basic computational elements of networks. They typically integrate information from a number of sources (inputs), and perform some relatively simple type of

processing on these inputs, and then output a single value which somehow summarizes its response to the inputs.

The basic **Unit** class in PDP++ contains real valued variables for representing a unit's activation and its net input from other units as well as its target pattern and/or external training input. In addition the unit class contains subgroups of sending and receiving connections between other units, and a 'bias' connection, which may or may not be present depending on the algorithm. A unit also has a position which represents its relative offset from its layer's position in the netview (see Section 10.6 [net-view], page 149).

As with many objects in PDP++, the **Unit** relies on a corresponding **UnitSpec** to provide most of the functions and parameters that control the unit's behavior. The unit itself contains the state variables. Thus, different units can have different parameters and functions simply by changing which **UnitSpec** they point to.

The following variables are found on the **Unit**:

`UnitSpec_SPtr spec`
: A pointer to the unit specifications for this unit (see below).

`Geometry pos`
: Specifies the unit's 3-D position relative to the layer. Used for display purposes and optionally for certain projection patterns

`ExtType ext_flag`
: This flag indicates which kind of external input unit last received. This may have one of four values:

  `NO_EXTERNAL`
  : Indicates that the unit received no input.

  `TARG`
  : Indicates that the unit received a target value, which is in the `targ` field.

  `EXT`
  : Indicates that the unit received external input, which is in the `ext` field.

  `TARG_EXT`
  : Indicates that the unit received both a target and external input.

  `COMP`
  : Indicates that the unit has a comparison value in its `targ` field. This is for computing an error statistic or other comparisons, but not for training the network.

  `COMP_TARG`
  : Both a comparison and a target (this is redundant, since all target values are included in comparisons anyway..)

  `COMP_EXT`
  : Both a comparsion and an external input.

  `COMP_TARG_EXT`
  : All three.

`float targ`
: The target value that the unit is being taught to achieve (i.e., for output units in a backpropagation network).

`float ext` The external input that the unit received. Depending on the algorithm, this can be added into the net input for the unit (soft clamping), or the unit' activation can be set to this value (hard clamping).

`float act` The unit's activation value, which is typically a function of its net input.

`float net` The unit's net input value, which is typically computed as a function of the sending unit's activations times their weights.

`Con_Group recv`
This group contains the unit's receiving connections. Each projection creates its own sub-group within this group (see Section 8.2 [obj-group], page 109), so `recv` just contains sub-groups which themselves contain the actual connections.

`Con_Group send`
This group contains sub-groups containing the unit's sending connections, one sub-group per projection (just like `recv`).

`Connection* bias`
A pointer to a **Connection** object which contains the bias weight for this Unit. Bias weights are treated as special connections which do not have a corresponding sending unit. This pointer may be NULL if the unit does not have a bias weight. The type of connection created here is specified by the `bias_con_type` member of the UnitSpec, and the ConSpec for this connection is in the `bias_spec` member of the UnitSpec.

The basic **UnitSpec** class defines the set of computational functions on the unit, and has parameters which control the unit's behavior. Specific algorithms add more parameters to this object.

`MinMaxRange act_range`
The legal range of activation values for the unit.

`TypeDef* bias_con_type`
The type of bias connection to create in the unit. The default value for this is set by different algorithms, and it can be NULL if no bias connections are to be created. The 'Build' operation should be performed if this connection type is changed manually.

`ConSpec_SPtr bias_spec`
This ConSpec controls the behavior of the bias on the unit in an algorithm-dependent fashion.

Note: the following information should be useful to those who wish to program in PDP++, but is not necessary for the average user to understand.

10.4.1 Implementational Details About Units

The base **UnitSpec** defines a standardized way of splitting up the computations that take place on a unit. This allows there to be a single function at the level of the **Network** which iterates through all of the layers and they iterate through all of their units and call these functions. This makes writing process code easier, and provides a conceptual skeleton

on which to implement different algorithms. Note that the unit has simple "stub" versions of these functions which simply call the corresponding one on the spec. This also simplifies programming.

`InitState(Unit* u)`
: Initializes the unit's state variables, including activations, net input, etc.

`InitWtDelta(Unit* u)`
: Initializes the stored connection weight changes (i.e., changes that have not yet been applied to the weights).

`InitWtState(Unit* u)`
: Initializes the connection weight values for all of the receiving connections of the unit.

`Compute_Net(Unit* u)`
: Iterates over the receiving connections of the unit and sets the unit's `net` field to the summed product of the sending unit's activation value times the weight.

`Send_Net(Unit* u)`
: Iterates over the *sending* connections of the unit and increments the `net` field of the unit's it sends to. This way of computing net input is useful when not all units send activation (i.e., if there is a threshold for sending activation or "firing"). A given algorithm will either use `Compute_Net` or `Send_Net`, but not both.

`Compute_Act(Unit* u)`
: Turns the net input value into an activation value, typically by applying a sigmoidal activation function, but this varies depending on the particular algorithm used. The version in the base **UnitSpec** just copies the net input into the activation (i.e., it is linear).

`Compute_dWt(Unit* u)`
: Iterates over the receiving connections on the unit and calls the `Compute_dWt` function on them, which should compute the amount that the weights should be changed based on the current state of the network, and a learning rule which translates this into weight changes. It should always add an increment the current weight change value, so that learning can occur either pattern-by-pattern ("online" mode) or over multiple patterns ("batch" mode).

`UpdateWeights(Unit* u)`
: Actually updates the weights by adding the changes computed by `Compute_dWt` to the weights, applying learning rates, etc. This function should always reset the weight change variable after it updates the weights, so that `Compute_dWt` can always increment weight changes. Note that this function is called by the **EpochProcess**, which decides whether to perform online or batch-mode learning (see Section 12.3.3 [proc-levels-epoch], page 192).

10.5 Connections

The Connection class contains the weights which represent the strengths of the relationship between units. Since there are typically many more connections than any other type

of object in a simulation, these objects are treated specially to speed up processing speed and reduce their memory consumption.

The main way in which connections are different than other objects, like Units, for example, is that they are usually operated on in a group. Thus, the **Con_Group** class becomes very important to determining how connections behave, which is not the case with a **Unit_Group**, for example.

The **Con_Group**, and not each connection, contains a pointer to the **ConSpec** which governs the behavior of all of the connections in the group. Also, **Con_Group**s can contain algorithm-specific parameters, and in general algorithms define their own type of **Con_Group**. When connections are created by projections, a new **Con_Group** is created to hold all of the connections for each projection.

The basic **Connection** class (which is often abbreviated **Con** when new types are defined based on it, e.g. **BpCon**) has only the weight value, `wt`. Other algorithms will add other variables as needed.

The basic **ConSpec** connection specification has parameters for determining how a connection's weights are to be initialized. The `rnd` field, which is of type **Random** (see Section 8.5 [obj-random], page 117), allows for weights to be initialized with random values. To get a specific weight value, use `UNIFORM` with a `mean` of the value you want and a `var` of zero. The `NONE` type of randomization will simply not do anything to the weight value. Note that this may be ignored if the **ProjectionSpec** which created the connections has its `init_wts` flag set (but most projection specs just use the ConSpec initialization anyway) (see Section 10.3 [net-prjn], page 137).

Each ConSpec also has a WeightLimit object which controls how the weights are constrained. This object has *min* and *max* fields and a controling field called *type* which can have the following values:

`NONE` no weight limitations (default).

`GT_MIN` constrain weights to be greater than min value.

`LT_MAX` constrain weights to be less than max value.

`MIN_MAX` constrain weights to be within min and max values.

In addition the ConSpec has a boolean field called *sym* which if true, symmetrizes (sets to the same value) the initial weights across two different connections that reciprocally connect the same two units.

The **ConSpec** is the place to look for parameters that determine how weights are updated, for example learning rate, etc. These are all defined in algorithm-specific versions of the ConSpec.

Note: the following information should be useful to those who wish to program in PDP++, but is not necessary for the average user to understand.

10.5.1 Implementational Details About Connections

The is a general policy regarding the organization of functions and parameters for connections, connection groups, and connection specifications:

Only those functions that relate to the computational processing done by the connections should be defined in the connection spec, while the "structural" or other "administrative" functions should be defined in the connection group object so that the spec can be invariant with respect to these kinds of differences. The idea is that the spec defines *functional* aspects while the object defines various implementational aspects of an object's function (i.e. how the connections are arranged, etc..).

Thus, the **Con_Group** type has a large number of functions that are useful for making, finding, and removing connections. See the header file '`src/pdp/netstru.h`' for a listing of these. The Con_Group has a special way of representing connectivity. The group itself contains **Connection** objects, which define the state variables associated with the connection itself. However, the pointer to the unit on the other side of the connection (the sending unit in the case of receiving connections, and the receiving unit in the case of sending connections), is kept in a separate list, which is in one-to-one correspondence with the connection objects. This arrangement allows for the same connection state variables to be shared across connections between different units. Indeed, the sending and receiving connection groups from the same projection share a single connection object between them. Some of the projection types (see Section 10.3.3.2 [net-prjn-tessel], page 139, Section 10.3.3.5 [net-prjn-misc], page 143) define additional forms of connection sharing.

Note that only one side of the connection is actually saved when a network is saved. This is the receiving side by default. Thus, after loading a project in from disk, the projection which manages the connections has to perform a `ReConnect_Load` function which builds the sending connections that correspond to the receiving connections that were just loaded in. A similar kind of operation must take place after copying a network.

For the functions that are defined in the **ConSpec**, a convention has been established regarding the division of labor between iterating through the connections in a group, and processing a given connection. Thus, there are two versions of each function defined in the con spec, one to apply to a single connection and another to apply to an entire Con_Group. The one which applies to a single connection has the name `C_XXX` where `XXX` is the name of the Con_Group version. Since some implementations of algorithms use bias weights, which are represented by a connection without a surrounding connection group, it is sometimes necessary to define a bias-weight version of a connection-specific function. Such a function will have the name `B_XXX`.

Note that the `C_` version of the function is *not* declared `virtual`, while the **Con_Group** is. This was done for reasons of speed, since the `C_` versions can be inlined within the iteration defined in the con-group version. However, it means that if you change one of the `C_` functions, *you must redefined the associated con-group version!*

Also note that the same object type, a **Con_Group** (and its associated **ConSpec**), is used for both sending and receiving connections. Thus, the **ConSpec** will have functions that apply to both cases.

As with the unit specs, a standard way of breaking up neural computations has been established by defining some basic functions on the con spec. Only the Con_Group version of these functions are listed below, but a C_ version is also defined. Also note that the C_ versions of these functions typically take a connection pointer, receiving unit, and sending unit arguments. The functions are as follows:

`InitWtState(Con_Group* cg, Unit* ru)`
: Initialize state variables (i.e. at beginning of training).

`InitWtDelta(Con_Group* cg, Unit* ru)`
: Initialize variables that change every delta-weight computation. This clears any existing weight change computations.

`float Compute_Net(Con_Group* cn, Unit* ru)`
: Computes the net input from the connections in this group, assuming that they are receiving connections.

`Send_Net(Con_Group* cg, Unit* su)`
: Adds the net input contribution from the given sending unit to all of the receiving units on the other side of these connections. This assumes that the con group is a sending connection group of unit `su`.

`float Compute_Dist(Con_Group* cg, Unit* ru)`
: Returns the distance (squared difference) between the unit activations and connection weights for all connections in group. This assumes it is a receiving group.

`Compute_dWt(Con_Group* cg, Unit* ru)`
: Computes the delta-weight change for all connections in the group. This is typically called on the receiving connections, and is defined by specific algorithms.

`UpdateWeights(Con_Group* cg, Unit* ru)`
: Updates the weights of the all the connections in the group. This again is defined by specific algorithms, but is called by generic functions up the network hierarchy.

10.6 Network Viewer

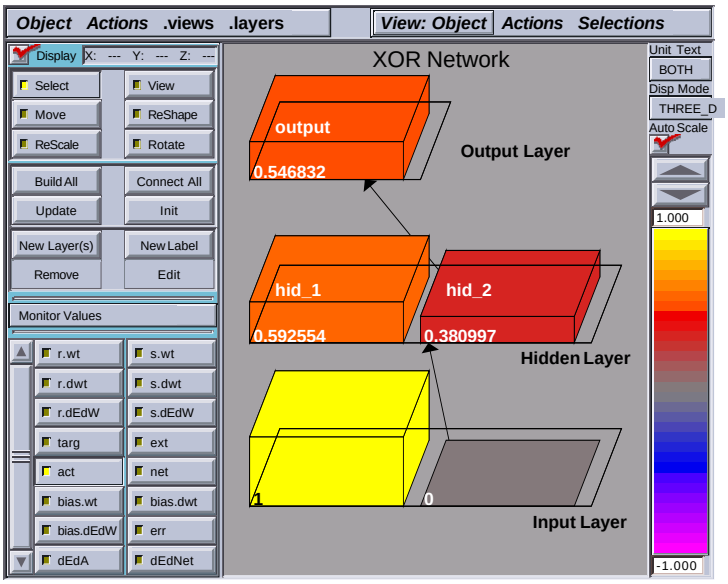


Figure: 2 The Network Viewer

The Network Viewer (NetView) provides an interactive pseudo-3D view of the network structure. A default view is associated with each network object, although networks may have as many additional NetViews as is necessary. The NetView can be spatially divided into four regions: Actions, Members, View, and Scale.

The NetView contains the following variables which control some of the display features of View and Scale Regions:

NetView Variables

`int skew` The skew controls the pseudo-3d depth dimension of the NetView's units. A skew of zero provides no perceived depth (2D). The default is 0.15.

`UnitShape shape`
Controls the shape of the units in the NetView and how their values are graphically displayed. This is also available as the *Disp Md* menu on the right-hand side of the NetView. The possible choices are:

`COLOR` A rectangular shape filled with a color from the colorscale corresponding to its value. If the unit's value is less than the minimum value of the colorscale then it will appear as stippled version of the minimum color. Similar stippling will occur it is greater than the maximum value of the colorscale. If the unit contains no value for the selected member it will be displayed as a stippled version of the View Region's background color.

`AREA`

`LINEAR` A rectangular shape filled with a background color corresponding to the midpoint value of the colorscale and a foreground rectangular spiral shape of with an area corresponding to the ratio of the unit's value to maximum range on the scalebar. Unit values above the scalebar's midpoint have spirals filled with the color of the maximum value of the scalebar, while units with values less than the scalebar's midpoint have spirals filled with the color of the minimum value of the scalebar. AREA spirals scale the size of their area with respect their value/scalebarmax ratio. LINEAR spirals scale their width and height linearly with respect to their value/scalebarmax ratio.

`FILL` FILL is similar to the AREA UnitShape but instead of a centered spiral area the FILL UnitShape begins covering an area at the bottom of the Unit and "fills" the unit left to right and bottom to top, covering a percentage area of the unit equal to the units value/scalebarmax ratio.

`DIR_FILL` This is similar to FILL except that the unit is divided with a horizontal line across the middle. Unit values greater than the midpoint

of the colorscale fill the upper half of the unit only, while values less than the midpoint of the colorscale only fill the lower half the unit.

`THREE_D` THREE_D Units utilize the pseudo-3D nature of the View Region to create a varying rectangular height field perspective of the unit's value. Unit values greater than the midpoint of the colorscale rise "above" the 3-D plane of the unit and unit values less than the midpoint of the colorscale sink below the 3-D plane of the unit. The units are colored in the same was as the COLOR UnitShape.

`ROUND` Round units have a circular or oval shape and are colored in the same way as the COLOR UnitShape.

`HGT_FIELD`
Like THREE_D, but each of the four corners of a unit are placed at the average height computed from the four adjacent units to that corner. This produces a smoothly-shaped field of height corresponding to average values of units. The units are colored in the same was as the COLOR UnitShape.

`HGT_PEAKS`
The same as HGT_FIELD, except that the center of the unit has a peak that is exactly the height corresponding to the value of the units.

`ColorScale* colorspec`
Controls the color spectrum used by the colorscale. (see Section 6.7 [gui-colors], page 67).

`float prjn_arrow_size`
Controls the size of the arrowheads at the end of the projections.

`float prjn_arrow_angle`
Controls the angle (sharpness) of the arrowheads at the end of the projections. Smaller values = sharper arrowheads.

`FontSpec layer_font`
The X11 specification for the font to use in drawing the layer names in the View Region

`FontSpec unit_font`
The X11 specification for the font to use in drawing the unit names and values in the View Region.

`Label_Group labels`
This group contains the arbitrary network labels visible in the View Region. Labels are usually created and place using the "New Label" button in the Action Region, however the can also be directly created in this group. Editing

these labels allows the user to change the text or font of the label, using a standard XWindows font specification string.

`NetViewGraph_Group idraw_graphics`
: This group contains the idraw graphics included in the network display – the menus *Load Graphic* and *Remove Graphic* manipulate these objects.

`UnitTextDisplay unit_text`
: Controls whether or not the unit's names and values are overlayed on top of the unit. This variable is also accessible directly in the NetView's Scale Region.

`SplitUnitLayout unit_layout`
: Controls how the units are split when multiple members are selected for display in the Member Region of the NetView.

`bool auto_scale`
: Controls whether or not the range of the scale in the Scale Region automatically adjust to range of values currently on display in the View Region. This variable is also accessible directly in the NetView's Scale Region as a toggle box.

`FloatGeometry spacing`
: Controls the spacing of the units and layers in the View Region. Useful values range from "0" to "1.0".

The NetView provides the following functions on its *Action* Menu:

`SetColorSpec(ColorScaleSpec* colors)`
: Set the color spectrum to use for color-coding values (NULL = use default).

`AutoPositionPrjnPoints()`
: Initialize the positions of all of the projection arrows according to the automatic positioning algorithm. If the positions are not quite ideal, the projections may be repositioned using the Move Mode (and this function removes any existing custom repositioning!)

`AutoPositionLayerNames()`
: Initialize the the size and position of the Layer names to fit in the lower left-hand corner of the layer (removes any custom positioning!)

`ZoomLayer(Layer* lay, float mag_factor_x, float mag_factor_y)`
: Set the magnification factors (zooming) for the display of a given layer.

`ResetLayerZoom()`
: Reset (remove) any magnification/zooming (rescaling) of the display of layers in the netview (made via the Zoom button or ZoomLayer menu).

`FreezeNetZoom()`
: Freeze (save) current network zooming/magnification (rescaling or repositioning) – it will no longer auto-resize to fit entire network in display.

`AutoNetZoom()`
: Clear any frozen/saved magnification/zooming (rescaling or repositioning) of the network display, enabling auto-zooming to fit entire network in display. This is the default.

`SetLayerFontSize(int point_size)`
Set the point size of the layer name font.

`SetUnitFontSize(int point_size)`
Set the point size of the unit name/value font .

`LoadGraphic(File name, float scale)`
Loads and Idraw image file of name `name` and scales it by `scale`. The image is displayed in background of the the View Region.

`RemoveGraphics()`
Removes all included graphics.

In addition the NetView provides a *Selection* menu with two sets of complementary functions. The **Set** version of these sets a given parameter or specification of the objects selected in the NetView. The **Show** version selects those objects in the NetView which have a given parameter or specification. These are very useful for constructing the network, and providing visual feedback about its state.

`Set/ShowUnitSpec(UnitSpec* spec)`
The unit specification, for units, groups of units, or layers.

`Set/ShowConSpec(ConSpec* spec)`
The Connection specification, for projections.

`Set/ShowPrjnSpec(PrjnSpec* spec)`
The Projection specification, for projections.

`Set/ShowPrjnConType(TypeDef* con_type)`
The type of Connection object to be created by the projection (`con_type` of Projection).

`Set/ShowPrjnConGpType(TypeDef* con_gp_type)`
The type of Con_Group to be created by the projections (`con_gp_type` of Projection).

`Set/ShowLayerSpec(LayerSpec* spec)`
The Layer specification, for layers.

`Set/ShowLayerUnitType(TypeDef* unit_type)`
The type of units to be created by the layer (`el_type` of the layer).

10.6.1 The Action Region

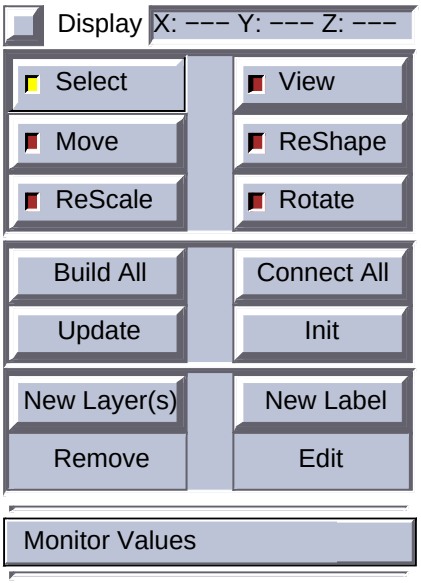


Figure: 3 The NetView Actions Region

The Actions region in the upper left contains a set of buttons which perform actions on the view. At the very top of the region is a toggle switch which provides convenient switching on and off of the display. When the display is switched off, the View and Scale regions will not be updated by the processes in the view's updater list, regardless of changes to the state of the network. (see Section 6.1.3 [gui-win-view], page 54).

To the right of the Display toggle is the Coordinate Box. The Coordinate Box displays the current location (X,Y,Z) of objects when they are moved and the size of objects when they are reshaped. At other times, the box remains empty.

Below the display toggle and Coordinate Box are Mode Buttons: Select, View, Move, ReShape, Zoom, and Rotate. These buttons are mutually exclusive, so only one may be pressed at a given time. Selecting one of these buttons enables a different mode of interaction with the View Region, and are described further in that section of the manual. (see Section 10.6.3 [net-view-view], page 157)

Below the Mode Buttons are the Major Action Buttons: *Build All*, *Connect All*, *Update*, and *Init*. Each of these buttons performs an action when the button is released. Sometimes one or more of these buttons will be hi-lighted, suggesting that the button's action should probably be performed due to the current state of the network.

The *Build All* button creates units in all of the networks layers in accordance with the `n_units` field of the Layer and the Layer's geometry. The *Connect All* button creates connections on all the units in accordance with projections and projection specs of the network's layers (see Section 10.1 [net-net], page 131).

The *Update* button refreshes the View to reflect the current state of the network. In most cases the network will have communicated its changes to the NetView, however if the NetView appears to be out of sync, the "Update" button will update it accordingly. The *Init* button resets the view so that the network is scaled to fit in the window. Zoom operations

on the network objects as well as Move and Zoom operations on the view region itself are undone when the "Init" button is pressed, *unless you run Actions/FreezeNetZoom* – this freezes the current zooming/move status of the netview until a subsequent *AutoNetZoom* is run. To reinitialize the projection points or the positions of the Layers' names select the corresponding function from the Actions menu of the NetView. (see Section 6.3 [gui-actions], page 57).

Below the Major Action Buttons are the *Minor Action Buttons*. The names of these buttons change to reflect the possible actions to be performed on the objects currently selected in the View Region. In most cases the lower two buttons will provide *Remove* and *Edit* actions for the selected objects. The upper two buttons typically provide actions used for creating new network objects and connecting them. For more information on how to use these buttons to build networks, see Section 10.7 [net-build], page 160.

At the bottom of the Action region is the *Monitor Values* menu. This menu provides a set of actions used for monitoring the values displayed in the NetView. To monitor values the user needs to create a **MonitorStat** statistic which handles the sampling and logging of the monitored values. To create the MonitorStat, the user first selects the network objects in the View Region to be monitored. Then after a member of the "Member Region" is selected, the user may select *New* from the *Monitor Values* menu. A dialog is then presented which asks the user for the process grain at which the values are to be monitored. The process grain is used to determine when the values are sampled from the network. In addition the process determines which Logs the sampled values are stored in. The dialog also asks the user for the network aggregation operator and whether or not aggregate statistics should be created in higher level processes. For simple monitoring, the default values of `COPY` and "NO" respectively will suffice. For more complicated monitoring the behavior of these variables is further explained in the statistics section of this manual (see Section 12.6.2 [proc-stats-monitor], page 203).

The *Monitor Values* menu also provides selections which allow the user to *Edit*, *Remove*, or change the objects or variables of previously created MonitorStats. To change the network objects which a MonitorStat is monitoring, simply select the objects in the "View Region" and choose *Set Objects* from the menu. To change the variable which is being monitored simply select the desired variable from the "Member Region" of the NetView and choose *Set Variable* from the menu.

10.6.2 The Member Region

Figure: 4 The NetView Member Region

Below the Action region, in the lower left corner of the NetView is the Member Region. This region contains a vertical scrollbox of buttons which are labeled with the names of the member fields of the units and their connections in the NetView. If there are more member field buttons than can fit in the Member Region, the vertical scrollbar on the left of the Member Region will allow the user to scroll through all the desired member fields.

By pressing one of the member buttons, the View Region will update to reflect the values of that member on the units it is associated with. In the case of connection member buttons, the user will need to pick a unit from which the connections are to be viewed. In this case, the View Mode action button in the Action Region will be automatically selected so that the user may pick a target unit. When The View Mode action button is selected the cursor will change to a pointing hand with which the user can pick units in the View Region. If the user chooses a unit member button instead of a connection member button, the display will revert to the default "Select" mode, and the "Select" Action button in the Action Region will be selected. In the "Select" mode, the pointer is usually an arrowhead.

By using the middle mouse button the user may select multiple member values to be displayed. When more than one member value is selected, the selected member buttons are numbered in the order of selection. The units in the "View Region" will split in to multiple sections each representing one of the selected member values. The units may split horizontally or vertically depending upon the setting of the `unit_layout` member of the NetView.

Selecting a member button with the right mouse button will provide a more detailed description of member variable.

10.6.3 The View Region

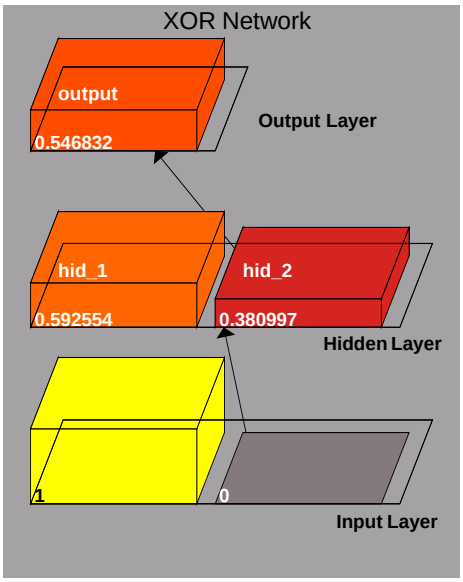


Figure: 5 The NetView View Region

The View Region is the core of the NetView and provides a multitude of configurable options to the user. The network's layers, units, and connections can be viewed, manipulated, scaled, and arranged as the user desires. In addition, arbitrary descriptive labels can be placed in the View Region to annotate the network. The user has full control over the colors used to view the network and can print the View Region itself using the PrintData() command on the NetView. Idraw image files may also be loaded into the View Region's background.

Non-obvious actions you can perform on objects in the NetView:

Single-click with left-mouse-button (LMB)
: Will select or deselect item, and update the action buttons at the left of the display.

Single-click with right-mouse-button (RMB)
: Will bring up the edit dialog for that object (for Layer names and labels).

Shift-LMB or middle-mouse-button
: Extends the selection to include multiple items.

Layers are displayed in the View Region as a rectangular boxes. The position of the layers is controlled by their `pos` member and the layer's size is controlled by the layer's `geom` member variable.

Within the Layer Box, **Units** are displayed as smaller rectangular boxes. The unit's `pos` variable controls its relative offset from the layer's position. If there are **Unit_Groups** within the layer, they are represented with their own box which surrounds the units they contain, and there are also two boxes associated with the layer in this case. One box represents the default size of each subgroup of units in the layer, and is the primary one that is selected

and manipulated in the interface. An additional box surrounds all of the unit groups, and identifies the extent of the layer as whole.

Projections are displayed as arrows starting at the sending layer's Layer Box and pointing at the receiving layer's LayerBox. In addition units and layers may have text near them which display their name or current value.

The actions of the mouse in the View Region are dependent upon the mode of the region selected by the Mode Buttons in the Action Region of the NetView.

In **Select Mode** the pointer is a small arrow. Clicking the left or right mouse button on an object selects the object under the pointer and unselects all other object. Clicking the middle mouse button selects also selects the object, but does not unselect the other selected objects. When an object is selected its border is drawn with a dashed line. Re-Selecting an already selected object Un-Selects the object. When a **Layer** or **Unit_Group** is selected, it will be displayed as grid instead of an outlined box. The grid provides the user with a depiction of how many units the layer or group contains, as well as the size and spacing of the units.

The names of the **Minor Action Buttons** in the Action Region will change is accordance with the objects which are selected. Since layers, unit groups, and units overlap, a special selection method is implemented for deciding which object is selected. On the first click of the mouse button, the layer is selected. On the second click, the layer is unselected and the unit group is selected. On the third click, an individual unit is selected. On the fourth click the individual unit is unselected.

In **View Mode** the pointer becomes a small pointing hand. This mode is used for selecting units from which to view sending or receiving weight values. View mode is orthogonal to select mode and is illustrated by a dotted line surrounding an object. Thus objects can be both selected and/or viewed at the same time. Since it is only useful to View individual units, clicking on a layer bypasses the Layer and Unit_Group selection of Select Mode and directly hi-lights a unit for viewing. Indeed, only units can be viewed. Clicking with the middle mouse button invokes the split unit mode and shows the sending or receiving values for multiple units.

In the other four modes, selection of objects occurs as in select mode. Each level of selection occurs on the up release of the mouse button, and these modes always operate on the currently selected object if there is one.

Therefore, to *Move* a layer, simply click on the layer and hold down the mouse button and move the layer.

To move a unit group, click on the layer and release, so that the unit group is selected. Then, the next down press will grab the unit group for moving – press and hold and move.

To move a unit, click twice, once to get through the layer, again to get through the unit group, at which point the unit will be selected and grabbed for moving upon the next down click.

In each of the following action modes, when the action is finished, the selected object is unselected. If no action was taken (i.e., the mouse button was clicked, but the mouse was not moved) the object remains selected. In this way, the user can select objects for editing and other Actions without being forced to re-choose the Select Mode button. In addition

the View Region itself my be manipulated by clicking the mouse in the background of the View Region and not selecting any objects.

In **Move Mode** layers, unit groups, and units are repositioned on a coordinate grid with spacing set at the size of an individual unit. As the layer or unit object is moved, the object will jump to the fixed grid position instead of moving smoothly with the mouse. As a layer or unit is moved, its position will be displayed in the Coordinate Box at the top of the Actions Region of the NetView. Using the left or middle mouse button moves the layer or unit objects in their x-y plane. Using the right mouse button allows movement in the x-z planes.

The head and tail of a projection arrow can be repositioned in Move mode as well. Although the position of a projection's head or tail is not constrained to the same grid like movement of the layer and units, it is constrained to lie within the layer it is connected to. Layer names and the arbitrary network labels may be moved without constraint. Moving the View Region itself is accomplished by clicking in the background area of the view. The pointer will change to a flat hand and the View Region will "slide" as the user moves the mouse. Pressing the middle mouse button constrains the movement to be horizontal, while pressing the right mouse button constrains the movement to be vertical.

Reshape Mode is used exclusively for changing the geometry of a layer or unit group. By clicking and dragging on a layer or unit group, the geometry (and number of units if there are no actual units created yet) of the displayed grid changes as the mouse moves. The geometry of the layer or unit group is displayed in the Coordinate Box at the top of the Action Region. If a layer or group already contains units, its area is constrained to be equal to or greater than the number of unit's in the layer or group. Thus a layer with 100 units could be sized to be 10x10 or 20x5, but not 4x3. Further reshaping a layer or group with units already in it will not change the number of units in the layer, while this will occur if there are no actual units.

Zoom Mode is used to change the magnification (zooming) of objects in the View Region, or the overall view itself. It is primarily used to rescale layers for maximum visibility when you have a really big network. You can reset the zooming of a layer with *Actions/ResetLayerZoom*, or impose a specific amount of zooming with *Actions/ZoomLayer*. If the mouse is pressed in the background of the View Region, the entire View Region can be scaled. Dragging the mouse upward zooms in on the region at which the mouse was first pressed. Dragging downward likewise zooms outward. If the middle mouse button is pressed only the horizontal dimension is scaled. If the right mouse button is pressed only the vertical dimension is scaled. The Init Button in the Action Region can be used to undo background scaling and recenter the Network, and *Actions/FreezeNetZoom* will lock in the current overall view zooming until a subsequent *Actions/AutoNetZoom*.

The Rotate mode is somewhat frivolous but can be used to further customize the objects in the View Region, or to manipulate imported Idraw graphics.

10.6.4 The Scale region

Figure: 6 The NetView Scale Region

The Scale Region is on the right edge of the NetView. At the top of the scale region is the Unit Text selector. This menu allows the user to directly access the NetView's `unit_text` member which controls whether or not unit names and/or values are displayed on the units.

Next is the display mode selector, which directly accesses the `shape` member of the NetView, controlling how the unit values are displayed.

Beneath this is the Auto Scale toggle. When the Auto Scale toggle is on, the values of the scalebar adjust to the positive and negative range of the the maximum of the value of the field selected in the Member Region. When the Auto Scale toggle is off, the up and down arrow buttons above and below the scalebar can be used to increase or decrease the range of the scale respectively. This can be used to "zoom" in on the differences between unit or connection values. In addition the user can click the mouse in the number region of the scalebar and type in the precise scale range by hand if so desired. The colors in the colorscale are determined by the `colorspec` field of the NetView.

10.7 Building Networks Using the Viewer

The network viewer (see Section 10.6 [net-view], page 149) can be used to build networks easily and rapidly. The *Minor Action Buttons* in the action region (see Section 10.6.1 [net-view-actions], page 154) in particular provide a series of actions that can be performed on objects selected in the view region (see Section 10.6.3 [net-view-view], page 157) that allow one to build the network. Also note that the tutorial covers some of the following material (see Section 4.3.2 [tut-config-networks], page 34).

The first step in building the network is creating the layers. The *New Layer(s)* button is highlighted when there are no layers in the network, and it is usable when nothing else

is selected. It will prompt for the number of layers to create. After they are created, they show up in the view region. Note that they might appear off-screen depending on how many were created, so you might have to hit the *Init* button to see them all.

The *ReShape* button is used to shape layers into their desired size. Note that the size of the layer is displayed in the coordinate area at the very top of the "actions" region (see Section 10.6.1 [net-view-actions], page 154). The default assumption of the software is to create as many units as will fit in the shape of the layer (i.e., `n_units = X * Y`). Whenever you use *ReShape* and there are no units in the layer currently, it will reset the `n_units` to fill the layer box completely. However, if you use *ReShape* and there *are* units in the layer, the value of `n_units` is not changed. Note, however, that it will be impossible to shape the layer to a size that is smaller than that which will contain all of the existing units. For more info on layer parameters, see Section 10.2 [net-layer], page 135.

There are two ways of specifying a number of units that is less than the size of the layer. One is to select the layer box, at which time, if it is empty, the *New Unit(s)* action button will become highlighted. This action will prompt you for the number of units to create (the default being the number that will fill the box completely). By entering a number that is less than this default value, you will simultaneously create the units in the layer, and set the `n_units` value to be that number. Thus, any subsequent `Build` actions on this layer will create that many units if they are not already there.

The units within the layer can be moved around. However, they must stay within the layer box. Thus, you must reshape the layer to a larger size if you wish to move the units beyond the box.

If the units within a layer are grouped into sub-groups, these constitute a distinct selection level, so that all the units in the group can be selected together.

Note that the *Build All* button is highlighted whenever there is a layer that does not have its full complement of `n_units` units in it. Pressing *Build All* will create units in all the layers which need them, and ensure that the units in the existing layers are of the type specified in the `units` group on the layer.

After creating layers and units within them, the next step is to specify the connectivity between layers. Alternatively, though less commonly (and more effortfully), connectivity can be specified on a unit-by-unit basis. This is covered later in this section. As was discussed above (see Section 10.3 [net-prjn], page 137), it is easier to specify connectivity in terms of projections between layers.

To create a new projection you select the *receiving layer first.* Then, "extend" select the sending layer(s) (use the middle mouse button or hold down the shift key while selecting, which adds the selected item to the list of those things selected, instead of making it the only thing selected). The *New Prjn(s)* button will be highlighted, and pressing it will create a projection into the first-selected receiving layer from the subsequently selected sending layer(s).

Note that you can also create *bi-directional projections* instead. Just use the *New BiPrjns* button instead of *New Prjn(s)*, and this will create a reciprocal projection into the sending layer from the receiving layer.

To *self-connect* a layer, simply select one layer, and press the *New Self Prjn* button.

Just as with units in layers, one can create connections in projections either all at once with the *Connect All* button, or individually by selecting the projection and using the *Fill Prjn(s)* button, which will be highlighted if the projection does not yet have any connections associated with it.

To connect units on an individual basis, one simply selects the receiving unit first and then the sending unit(s), and selects the *Connect Units* button (or the *BiCon Units* to bidirectionally-connect them). The connections made in this way will be associated with a **CustomPrjnSpec**, which basically just ensures that the connectivity pattern will not be reset when the *Connect All* button is pressed (i.e., it has no `Connect` function because the connections are made one-by-one).

The specifications associated with the objects in the network can be viewed and changed by using the *Selections* menu of the network viewer. This allows one to associate different parameters or types of processing with different components of the network (e.g., projections with different patterns of connectivity, units with different activation functions or parameters, connections with different learning rates, etc..)

Note that each layer has a default **UnitSpec** associated with it, which is applied to any new units created in the layer or when *Build* is performed. Thus, if all of the units in the layer will be using the same unit spec, it is a good idea to just select the layer itself and then use *Selections/Set Unit Spec* to set the spec. This will automatically apply this spec to all of the units in the layer.

Also note that connection specifications (**ConSpec**s) are associated with projections. Thus, projections will be selected when *Selections/Show Con Spec* is performed, and you should select projections when doing the *Selections/Set Con Spec*.

Finally, all objects can be *Remove*d and *Edit*ed by selecting them and pressing the appropriate action button. When editing a group of objects (i.e., after having multiply selected them and pressed *Edit*), these objects should remain the only ones selected until the dialog is either *Ok*ed or *Cancel*ed, since changing what is selected affects what the edit dialog thinks its editing, and you won't be able to *Apply* your editing changes if the selections change.

11 Environments, Events, and Patterns

Conceptually, a network always acts in an environment of some kind. For example, back propagation and pattern associator networks act in an environment of input-output patterns. An unsupervised learning network operates in an environment consisting of input patterns.

In PDP++, the object of type **Environment** contains all the information that specifies the kinds of stimuli or patterns that the network will be tested or trained on. The role of the environment is to represent all of the data, structured so that the data can be interpreted by the processes and presented to the network appropriately. Thus, it is like a database or a library. The environment does not specify things like the *order* with which events will be presented to the network–this is the job of the processes, which, to continue the metaphor, act like librarians in providing the interface between the data (the environment) and the consumer (the network).

There is a special relationship between the **EventSpec**, which specifies properties of events, and the **Event** objects themselves. Any changes made to the event spec (or its pattern specs), for example in the number of patterns per event or size of one of the patterns, are automatically propagated to all of the events that use the event spec. Thus, the **EventSpec** acts much like a dynamic template for events, which is a different role for a spec object. This design makes it very easy to modify environments once they've been created.

The **EnviroView** enables you to interactively configure the event specs for the events, much in the same way the net view allows one to configure networks. In the standard color scheme, environment objects are colored green (like the great outdoors!).

11.1 Environments

The **Environment** is used by the processes to present inputs to the **Network**, and to provide training values for the network as well. At appropriate intervals, the processes will ask the environment for an **Event**. The event is a snapshot or instance of the environment at a given time. It is up to the environment to generate the appropriate event given an index presented by the processes. Typically, these events are precomputed and the environment simply iterates over the a list of events. Optionally, an environment can perform more complicated event choosing operations and can even modify or generate events on the fly if so desired (supported by the **InteractiveScriptEnv**, Section 11.11 [env-other], page 182).

Sometimes the events may need to be further organized into subgroups of events in order to represent groups of events that belong in a particular sequence (see Section 11.3 [env-seq], page 169). Specialized processes like the **SequenceEpoch** and the **SequenceProcess** interact with the Environment by asking it for one of these subgroups of Events. The Sequence Epoch allows updating of weights and other operations to occur after the entire "batch" of events in the Event SubGroup has been presented (see Section 12.4.1 [proc-special-seq], page 195).

In summary, there are three different interfaces to the environment:

1. A flat list of events accessed by event index (supported by the standard **EpochProcess**). `InitEvents()` is called at the start, and a **ScriptEnv** can render a batch of events for each epoch.

2. Groups of events accessed first by group index, then by index of event within group (supported by **SequenceEpoch** and **SequenceProcess**, Section 12.4.1 [proc-special-seq], page 195). This allows for subsets of events to be grouped together as a sequence. Again, `InitEvents()` is called at the start of the entire epoch.

3. An 'interactive' model that doesn't depend on indicies at all, supported by **InteractiveEpoch** (see Section 12.4.2 [proc-special-inter], page 197). This allows for events to depend on what has just happened in the network. `InitEvents()` is called at the start of an epoch, and `GetNextEvent()` is called for each new event until it returns a NULL. In a **IneractiveScriptEnv**, `GetNextEvent()` calls the script, and returns the event in the `next_event` pointer (which needs to be set by the script) – see Section 11.11 [env-other], page 182 for details.

In the generic **Environment** class there are two groups:

`Event_Group events`
: This is where the events are stored.

`BaseSpec_Group event_specs`
: This is where the **EventSpec** objects are stored. Note that event specs, unlike other specs, reside in the environment, and not in the **Project** `.specs` group. This makes environments self-contained so that they can be loaded and saved independent of the project.

The following functions are defined on the **Environment** object. Note that a given process will either use the Group model or the Event model of the environment. The group model is for sequences, which are stored in subgroups, and the event model treats the environment just like a flat list of events. These functions are on the *Actions* menu:

`InitEvents()`
: Initializes events for an epoch. Used for algorithmically generated events which are generated at the start of each epoch. This does nothing in the base class, but in a `ScriptEnv` it calls the script, which can then generate events.

`UpdateAllEvents()`
: Updates all events in accordance with their corresponding event specs. This should happen automatically but things can get out of whack and this should clean everything up.

`UpdateAllEventSpecs()`
: Updates all events in accordance with their corresponding event specs. This should happen automatically but things can get out of whack and this should clean everything up.

`EventCount()`
: Returns the number of events in the Environment. This is used by the processes to determine how long an epoch should be.

`GroupCount()`
: Returns the number of event groups in the Environment, for sequence-based processing. This determines how many sequences go in an epoch.

`GetNextEvent()`
: Returns the next event for processing (or NULL to signal the end of the epoch). This is the interface for the interactive environment model (hook for generating new event based on current state).

`UnitNamesToNet(EventSpec* event_spec_with_names, Network* network)`
: Copies names from pattern spec `value_names` to corresponding units in the network. Uses default event spec and network if NULL. This is a convenient way to name units in the network – these names are otherwise lost when the network is rebuilt.

`AutoNameAllEvents(float act_thresh, int max_pat_nm, int max_val_nm)`
: Automatically name all events based on the pattern names and value (unit) names for those units with activations above act_thresh, e.g., `Inp:vl1_vl2,Out:vl1_vl2`. There are also versions of this functionality in Events and Event specs.

The following functions are defined in the *Object* menu and provide additional input/output functionality:

`ReadText(ostream& strm, EventSpec* espec, TextFmt fmt)`
: Reads in an Environment from a text file which is in the format from the old PDP software, or other formats based on the fmt parameter. This uses the given event spec which must correspond to the pattern file being read in. See Section 11.7 [env-import], page 176.

`WriteText(istream& strm, int pat_no, TextFmt fmt)`
: Writes an Environment to a text file in the format from the old PDP software, or other formats. See Section 11.7 [env-import], page 176.

`ReadBinary(ostream& strm, EventSpec* espec)`
: Reads in an Environment from a binary file, which must be just a continuous stream of floating point numbers that are applied in sequence to the patterns and events of the environment. See Section 11.7 [env-import], page 176.

`WriteBinary(istreamm& strm, int pat_no=-1, TextFmt fmt = NAME_FIRST)`
: Writes an Environment to a binary file as a continuous stream of floating point numbers. See Section 11.7 [env-import], page 176.

The following are function interfaces used by processes to access events:

`GetEvent(int event_number)`
: Returns a pointer to the Event corresponding to `event_number`. The event number should be between 0 and `EventCount()`.

`GetGroup(int group_number)`
: Returns a pointer to the Event group corresponding to `group_number`, which should be between 0 and `GroupCount()`.

11.2 Events, Patterns and their Specs

Events are the snapshots of the Environment presented to the Network. They represent a single coherent set of stimuli drawn from the environment. They are comprised of a set of patterns, which hold the information affecting the network on a layer by layer basis.

The structure of an event is determined by its corresponding **EventSpec**. Any changes made in the event spec are automatically made to all of the events that use that event spec. Thus, there must be one event spec for each different type of event created. Event specs reside in the environment itself, and not in the Project like other specs. The **Event** has a pointer to its spec, called `spec`. This works just like other spec pointers (see Section 8.4 [obj-spec], page 115).

Both the **Event** and **EventSpec** objects are essentially just containers for their constituent **Pattern** and **PatternSpec** objects, which are also kept in one-to-one correspondence. Thus, any new pattern specs added in the `pattern` group in an event spec will result in corresponding patterns in the `pattern` group on the event. Different varieties of events will add event-level parameters like frequency and time (see Section 11.10 [env-freq], page 181, Section 11.11 [env-other], page 182).

A **Pattern** is simply a list of real numbers, which are kept in a floating-point **Array** object called `value`. It can be "applied" to a specified layer in the network. Each element of the value array in the Pattern corresponds to a Unit in the Layer. In addition, one can assign a flag to each value, which will alter how this value is applied to the units in the network. The `flag` member of a Pattern holds the flags, which are only present if the `use_flags` field in the corresponding pattern spec is set to `USE_PATTERN_FLAGS` or `USE_PAT_THEN_GLOBAL_FLAGS`. All of the flags are independent of each other and be either on or off. The meaning of the flag values are given in the `PatFlags` enumerated type in the **PatternSpec** and listed below.

`TARG_FLAG`
: unit's TARG flag is set

`EXT_FLAG`
: unit's EXT flag is set

`COMP_FLAG`
: unit's COMP flag is set

`TARG_VALUE`
: pattern value goes to the unit targ field

`EXT_VALUE`
: pattern value goes to the unit ext field

`NO_UNIT_FLAG`
: no unit flags are set, but value is set as normal

`NO_UNIT_VALUE`
: don't set the unit's value, but flag as normal

`NO_APPLY`
: don't apply this value to unit (no flags either)

The options include the ability to control how the unit's external input will be flagged, and where the value will be copied on the Unit (i.e., to its `targ` or `ext` field). This gives the user complete control over how the pattern value is presented.

Note that the `value` field is an `float_RArray` type, which has a range associated with it (hence the RArray), and it also has a number of other useful functions that enable the distance between two arrays to be computed, or various statistics like the mean, variance, etc. to be computed on a given array. These functions may be useful in creating and analyzing patterns.

The **PatternSpec** has several fields which determine where its corresponding pattern is presented, and what meaning it has to the network:

`PatType type`
: The type of the pattern, which determines how the network will interpret its values. This can be one of the following values:

  `INACTIVE:`
  : The corresponding pattern is not presented to the network.

  `INPUT:`
  : The corresponding pattern is presented to the network as external input activation: the value is copied to the `ext` variable of the corresponding Unit, and the `EXT` flag is set on the `ext_flags` of the unit (and the layer the unit is in).

  `TARGET:`
  : The corresponding pattern is presented to the network as target activation: the value is copied to the `targ` variable of the corresponding Unit, and the `TARG` flag is set on the `ext_flags` of the unit (and the layer the unit is in).

  `COMPARE:`
  : The corresponding pattern is presented to the network as comparison pattern, which does not affect activation states or learning, but can be used to compare output values to expected ones via error statistics (e.g., the **SE_Stat**). The value is copied to the `targ` variable of the corresponding Unit, and the `COMP` flag is set on the `ext_flags` of the unit (and the layer the unit is in).

`PatLayer to_layer`
: The network layer to present the Pattern to. This can be one of the following values:

  `FIRST:`
  : Apply to the first layer in the network.

  `LAST:`
  : Apply to the last layer in the network.

  `LAY_NAME:`
  : Specify the Layer to apply to by name. The name should be set in the variable `layer_name`.

  `LAY_NUM:`
  : Specify the Layer to apply to by layer number. The number should be set in the variable `layer_num`.

`String layer_name`
: Contains the name of the layer to apply to (used if `to_layer` is set to `LAY_NAME`).

`Int layer_num`
: Contains the number of the layer to apply to (used if `to_layer` is set to `LAY_NUM`).

`TypeDef* pattern_type`
: Determines the type of **Pattern** object that is created in events that use this pattern spec. Changing this after events have already been created will *not* cause them to change pattern types. You have to remove the Events and start over if you want to change the types of patterns that are used.

`LayerFlags layer_flags`
: Determines how the layer's `ext_flags` will be set. The `DEFAULT` is to set them according to the pattern `type` as described above, but they can be set to any of the possible combinations of flags (`TARG, EXT, COMP`), or to `NO_LAYER_FLAGS` at all. This can be useful if you have an pattern which uses flags for different values (see `use_flags`) so the layer actually receives multiple different kinds of external input.

`PatUseFlags use_flags`
: Determines how the `flag` field of the Pattern and the `global_flags` of this PatternSpec will be used when applying the Pattern to the network. This can be one of the following values:

  `USE_NO_FLAGS:`
  : Both pattern flags and global flags are ignored

  `USE_PATTERN_FLAGS:`
  : Patterns flags are applied, global flags are ignored

  `USE_GLOBAL_FLAGS:`
  : Global flags are applied, individual pattern flags are ignored

  `USE_PAT_THEN_GLOBAL_FLAGS:`
  : If any flags are set on the pattern then use all the pattern's flag settings, otherwise apply the global flags for that pattern. Note that the choice of global versus pattern flags is not done individually by each flag on the pattern but rather as an all or nothing check on each pattern to determine if any of its flags are set.

`int n_vals`
: The number of values to create in the corresponding pattern. It should be equivalent to the number of units in the corresponding layer. If this value is set to 0, it will be filled in with the number of units on the corresponding layer of the default network in the current project, if the layer can be found.

`PosTDCoord geom`
: Determines the shape of the values as displayed in the **EnviroView**. It should correspond to the geom of the corresponding layer, and is initialized as such if possible.

`PosTDCoord pos`
: The position of the start of the event in the **EnviroView**. Any overlap of displayed patterns is detected automatically, and the `LinearLayout` action on the EventSpec can be called to arrange the events linearly across or down the display.

`float initial_val`
: The initial value that will be placed in newly created pattern `value` arrays.

`Random noise`
: This adds random noise of the given specification as the pattern is applied to the network.

`String_Array value_names`
: The elements in this array are in one-to-one correspondence with the values in the pattern, and can be used to label the patterns in the **EnviroView** (see Section 11.4 [env-view], page 169). This is very useful

`int_Array global_flags`
: This contains flags that can apply (depending on `use_flags`) to all of the patterns, determining how they are presented.

Note: the following information should be useful to those who wish to program in PDP++, but is not necessary for the average user to understand.

11.2.1 Implementational Details of Events and Patterns

Note that layer pointer in a pattern spec depends on which network the environment is being applied to. Thus, there is some checking that takes place when an event is being applied to the network to make sure that the layer pointers are up-to-date. Along these same lines, any time the events or patterns are edited, the layer pointers are reset, with the idea being that they might have changed where the patterns are being applied to.

The basic set of functions can be found in the '`src/pdp/enviro.h`' file. They are fairly straightforward.

11.3 Representing Sequences of Events

Sequences of events are represented in PDP++ by creating subgroups within the main `.events` group of an environment. Each sub group should contain a set of events that define one discrete sequence of related events. If the appropriate **SequenceEpoch** and **Sequence** processes are used, they will access the `GroupCount` and `GetGroup` functions on the environment (see Section 11.1 [env-env], page 163), which will allow the epoch-level process to iterate over the groups (sequences) in the environment, while the sequence process iterates over the events within each group. The sequence-based processes are discussed in Section 12.4.1 [proc-special-seq], page 195.

11.4 The EnviroView

The EnviroView provides a graphical representation of the Environment, allowing the user to view, create, and edit Events and Patterns and their corresponding specs. In addition

the EnviroView can be used to track the current Event presented to the network by setting its updaters (see Section 6.1.3 [gui-win-view], page 54).

There are two basic modes to the enviro view — initially, it is in Edit Events mode, where the Events and Patterns are displayed and can be edited. By hitting the *Edit Specs* button at the top of the left-hand set of buttons, the display changes to editing the EventSpecs and PatternSpecs.

11.5 Edit Events in the EnviroView

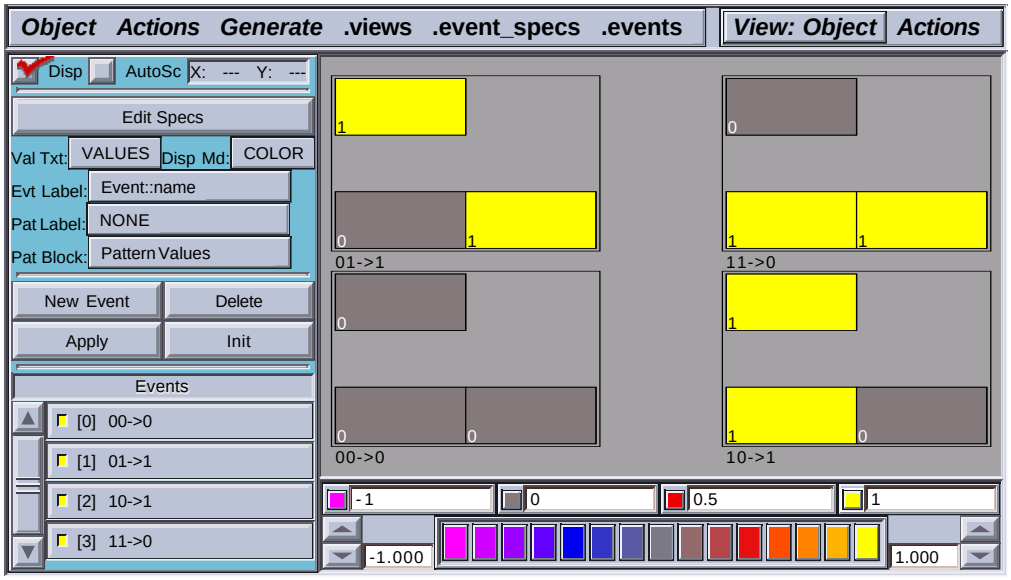


Figure: 7 The Environment Viewer – Events

Non-obvious actions you can perform on objects in the EnviroView

Single-click with left-mouse-button (LMB) on Event Button
Will select event for display in the view area to the right.

Single-click with right-mouse-button (RMB) on Event Button
Will bring up the edit dialog for the Event.

Shift-LMB or middle-mouse-button (MMB) on Event Button
Extends the selection to include multiple events, and can remove an event from the display if performed on an already-selected event.

Click on already selected Event Group Button
Will select all the events within the group, and deselect the group button.

LMB on a pattern element of displayed event
Will set the value of that pattern element according to highlighted drawing color (in color palette at bottom of display).

MMB on a pattern element of displayed event
Will set the value of that pattern element to 0.

RMB on a label on the displayed event
: Will allow you to edit the value (e.g., change the event name, or its frequency if that is what is displayed).

Turning off the Disp checkbox at top prevents display of events
: This can be useful for manipulating large numbers of events using the event buttons – just turn off the Disp and then you can select all the events for setting specs or changing types, etc.

All of the events in the Environment are listed sequentially from top to bottom in a scrollable vertical list on the left side of the EnviroView. The index number of the event is shown on the left-hand side of the label, and the first four characters of the sub-group name if the event is in a sub-group. Sub-groups have their own button that preceeds the events that are within them, and are indicated by the `>>` symbol on the left of the button. Selecting these group buttons twice (double clicking) will select all the events within them.

The following action buttons appear in the upper left of the EnviroView:

`New Evt/Gp`
: This button provides a convenient method for creating events or sub-groups in the Environment. Pressing the New Evt/Gp button creates a "New" dialog which prompts the user for the number and type of events or groups to create. Each new event is modeled after the default (usually the first) EventSpec in the Environment. If there are no EventSpecs, a default EventSpec consisting of an input pattern for the `FIRST` layer and a target output pattern for the `LAST` layer is created for the new events. If there are currently no events, the button will appear hi-lighted indicating that it is the suggested course of action. If there is one sub-group selected, then this button will create events within that selected subgroup.

`Delete`
: The Delete button removes the currently selected Events or groups from the Environment.

`Xfer/Dupe, Xfer Event, Dupe Event, Dupe Group`
: If event(s) are selected, Dupe Event will duplicate them. If sub-group(s) are selected, Dupe Group will duplicate them (and their associated sub-events). If a group and some events are selected, then Xfer Event will transfer the events into the selected group.

`Set Spec/Type, Change Type`
: If event(s) are selected, Set Spec/Type will bring up a choice of either setting the spec for these events, or changing their type. If sub-group(s) are selected, Change Type will change their type.

`Apply`
: The Apply button saves the changes made to the Patterns. It will appear hi-lighted if there are potentially changes which have not been saved.

`Init/Revert`
: The Init button refreshes the EnviroView to reflect the stored values of the currently displayed Events. *Any un-applied changes to the patterns will be lost.*

At the bottom of the EnviroView is a painting ColorBar. This colorbar differs from the colorbar in the scale region of the NetView in that the different shades of color are selectable

for use in "painting" values into the patterns of the Events. In addition, four custom value pads are provided for entering specific values. The color corresponding to the value of the pad appears next to the value and is selectable for use in painting just like to shades of the colorbar. These pads appear above the ColorBar and have default values of (-1.0, 0.0, 0.5, 1.0), which can be changed just by typing in a new value. The minimum and maximum of the range of the ColorBar may be independently specified or automatically scaled to the minimum and maximum value of the currently displayed events if the "Auto Range" toggle in the upper left corner of the EnviroView is selected.

When an Event is selected for viewing, it appears in the EnviroView as as set of name labels and colored blocks. The labels and blocks are positioned according to `pos` and `geom` values of the PatternSpecs for each Pattern in the Event (see Section 11.6 [env-view-specs], page 174 for how to configure this interactively). The color of the blocks corresponds to its value in the ColorBar at the bottom of the EnviroView. If a pad or shade value is selected in the ColorBar, the user may change the values of the Patterns by clicking or dragging on the colored blocks. Pressing the left mouse button changes the value of the block under the mouse pointer to be the selected value in the ColorBar. Pressing the middle mouse button changes the block's value to it previous value.

In the upper left area of the EnviroView are several menus which control certain features of each event which is displayed. The first menu called *Val Txt* determines what kind of textual information is rendered along with the pattern values (could be `VALUES` or `NAMES` (for the value_names) aor `BOTH`). The *Disp Md:* menu selects the way in which the values are rendered – `COLOR`, `AREA`, or `LINEAR`, just as in the NetView (see Section 10.6 [net-view], page 149).

The next two menus control what kinds of labels are shown along with each event and each pattern. *Evt Label* controls which propery of the Event is shown in the Event Label region (bottom) of the displayed events. Likewise, the *Pat Label* menu contols what pattern properties are shown in the formatted pattern label section of the displayed events (at the top of each pattern). If the displayed event and/pr pattern does not have the propery which is selected, a "n/a" will be displayed.

The *Pat Block* menu controls which property of the pattern is being displayed and edited when the user clicks on the pattern blocks described above. By default this menu is set to "Pattern Values" indicating that changes in the blocks colors reflect changes in the actual values of the patterns. More complicated patterns which behave in specialized ways can be created by changing the the "Pat Block" menu to one of the other selections and editing a pattern's flags (see Section 11.2 [env-event], page 166.) When editing pattern values a color value greater than zero sets the flag on, and a colorvalue less than or equal to zero sets the flag off. The user may also choose to edit the global PatternSpec flags from this menu. In this case the global flags on the displayed pattern's PatternSpec will be displayed and changed when the "Apply" button is pressed. Note that if multiple Events are being edited and more than one of those Events shares the same EventSpec then the last selected Event's values will be applied when the "Apply" button is pressed.

The EnviroView provides the following variables for customizing the display:

`EventLayout event_layout`
: Controls whether events are arranged `VERTICALLY` or `HORIZONTALLY` when multiple events are selected for viewing.

`bool auto_scale`
: Controls whether or not the ColorBar at the bottom off the EnviroView is automatically scaled to reflect the minimum and maximum pattern values of the Events currently displayed in the EnviroView.

`ValDispMode val_disp_mode`
: Controls how the values of the Patterns are displayed in the blocks. The `COLOR` setting fills the blocks with the colors from the colorbar, while the `AREA`, and `LINEAR` settings draw size varying spirals in the blocks. The spirals are similar to the spirals available for viewing in the NetView and a described in more detail in that section. (see Section 10.6 [net-view], page 149).

`ColorScale* colorspec`
: The colorscale to use for the ColorBar at the bottom of the EnviroView (see Section 6.7 [gui-colors], page 67).

`ValTextMode val_text`
: Can be `VALUES` or `LABELS` or `BOTH` (same as *Val Txt* menu) – controls whether or not the names stored in PatternSpecs' `value_names` array are displayed on top of the blocks (LABELS), and/or the numerical value of the pattern is displayed (VALUES).

`bool no_border`
: If selected, this eliminates the black border around the pattern values. This is useful for displaying picture-like stimuli.

`FontSpec view_font`
: The X11 specification for the font to use in drawing all of the basic text in the view.

`FontSpec value_font`
: The X11 specification for the font to use in drawing the pattern labels and values in the View Region (might want it smaller).

In addition, the EnviroView provides the following functions:

`SelectEvents(int start=0, int n_events=-1)`
: This will automatically select a range of events for display in the view – saves on clicking. start is the index of the event to start displaying, and n_events is the number to display (-1 = all).

`DeselectEvents(int start=0, int n_events=-1)`
: This is the de-selecting complement to SelectEvents.

`SetEventSpec(EventSpec* es)`
: Changes the EventSpec of all the currently displayed Events to `es`

`ChangeEventType(TypeDef* new_type)`
: Change event types for selected events

`ChangeEventGpType(TypeDef* new_type)`
: Change event group types for selected event groups

`DuplicateEvents()`
: Duplicate selected events

`DuplicateEventGps()`
: Duplicate selected event groups

11.6 Edit Specs in the EnviroView

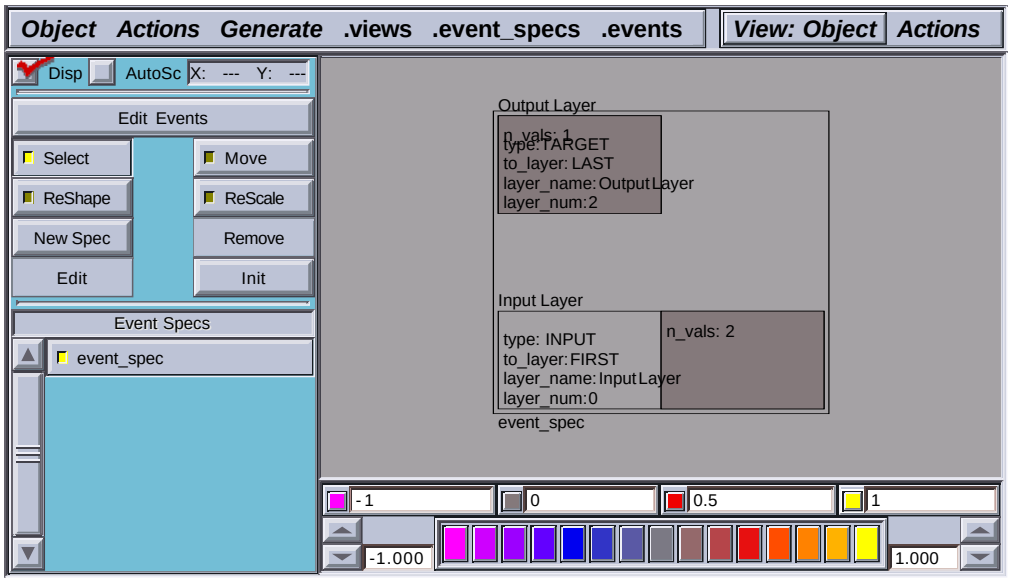


Figure: 8 The Environment Viewer – Specs

Non-obvious actions you can perform on objects in the EnviroView in this mode are:

Single-click with left-mouse-button (LMB) on EventSpec Button
: Will select event spec for display in the view area to the right.

Single-click with right-mouse-button (RMB) on EventSpec Button
: Will bring up the edit dialog for the EventSpec.

Shift-LMB or middle-mouse-button (MMB) on EventSpec Button
: Extends the selection to include multiple event specs, and can remove an event spec from the display if performed on an already-selected event spec.

RMB on text in the view area
: Will bring up an edit dialog for that text.

As in the "Edit Events" mode, the "Edit Specs" mode displays event specs in a scroll box along the bottom left-hand side of the window. It differs in having a variety of different action buttons above that, and in the way that the event specs and pattern specs are displayed.

When you click on an event spec, the event spec is displayed in "skeleton" format in the viewer area. By default, if there is only one event spec, it is automatically selected. The overall event spec is represented by a box, within which there are one or more pattern specs, represented by a grid of pattern elements. At the lower-left of the pattern spec is a display of four critical pieces of information about how that pattern is to be presented to the network (type, to_layer, layer_name, and layer_num, see see Section 11.2 [env-event], page 166 for details). The total number of values in a pattern is indicated by a grey square

with the "n_vals" label on it and a number indicating the number of values. This can be moved to change the total number of values within the constraints of the overall pattern geometry.

There are four mutually-exclusive tool selection buttons at the top of the action area, which operate much like their counterparts in the NetView:

`Select` Select mode makes clicking in the view area act to select items. In select mode, the entire event spec is selected upon the first click, and then the patternspec under the mouse is selected on the next click, and then any further sub-level of text is after that. Some of the other action buttons will update based on what is currently selected.

`Move` Move mode enables moving of PatternSpecs within the event. The patterns will be directly selected upon first mouse click, so just click and drag. You can also determine the total number of values in an event by moving the grey square that has the "n_vals" label on it, which typically is located in the upper-right corner of each pattern spec. You can also move the entire display by clicking on the background and dragging it around.

`ReShape` This allows you to reshape the geometry of the PatternSpecs. Simply click and drag to reshape the patternspec geometry.

`ReScale` This only works on the background, and serves to rescale the size of the entire display.

Below that are four buttons that change character depending on what is selected:

`New Spec, New Pattern, Set To Layer`
If nothing is selected, New Spec will allow you to create a new Event Spec (if one is selected, use Dupe Spec to get a new one). If an EventSpec is selected, New Pattern will allow you to create a new PatternSpec within that EventSpec. If a PatternSpec is selected, Set To Layer will prompt you for the name of a network Layer, and will configure the pattern to fit the geometry, etc of this layer (it will ensure that this pattern fits that layer and will be send to that layer when presented). **Set To Layer is the easiest and safest way to configure a pattern for a given network layer (see also Updt Fm Events)!**.

`Rmv Spec(s), Pat(s)`
Will remove (delete) selected EventSpecs or PatternSpecs.

`New Child, Edit Names`
New Child makes a child spec of selected EventSpec – a child spec inherits all the properties of the parent, except those checked as being unique. The only way to mark features as unique is to edit the pattern spec. Edit Names of a selected Pattern Spec will pull up an editor for the `value_names` that label values with names.

`Layout/Updt`
Pops up a dialog for selecting one of two actions: **Linear Layout** will arrange patterns in a line with a single space between them – useful for quickly arranging patterns that might be spread out. **Updt Fm Layers** will go through

each pattern and perform Set To Layer on that pattern, based on the current layer that each pattern is associated with. This is a quick way to update the environment to any changes you have made to your network.

`Copy Fm Spec`
: Allows you to copy parameters from another event spec to selected one.

`Dupe Spec` Duplicates currently selected event spec.

`Edit Spec(s), Pat(s)`
: Will bring up an edit dialog for selected objects (note: can also use RMB).

`Init` Rebuilds the display – should not generally be necessary but use if you suspect display is not up-to-date.

So, a typical session starting from scratch would be to do New Spec, then click on each pattern spec in turn and do Set To Layer and select the layer these patterns will go to. Then, the patterns may have to be moved around a bit, or new patterns created if there are more than two that receive environmental input (or one removed, etc). In general, everything should be managable within the interface.

11.7 Importing Environments from Text Files

To aid in the conversion of environments from the old PDP software to the format used in PDP++ ('`.pat`' files), and for generally importing training and testing data represented in plain text files, we have provided functions on the Environment that read and write text files. These functions are called `ReadText` and `WriteText`.

There is also a **FromFileEnv** that reads event patterns from files incrementally during processing (see Section 11.11 [env-other], page 182).

The format that these functions read and write is very simple, consisting of a sequence of numbers, with an (optional) event name at the beginning or end of the line. Note, you must specify using the fmt parameter whether there will be a name associated with the events or not. **Important: the name must be a contiguous string, without any whitespace – it can however be a number or have any other ASCII characters in it.** When reading in a file, ReadText simply reads in numbers sequentially for each pattern in each event, so the layout of the numbers is not critical. If the optional name is to be used, it must appear at the beginning of the line that starts a new event.

For example, in the old PDP software, the "xor.pat" file for the XOR example looks like this:

```
p00 0 0 0
p01 0 1 1
p10 1 0 1
p11 1 1 0
```

It is critical that the **EventSpec** and its constitutent **PatternSpecs** (see Section 11.2 [env-event], page 166) are configured in advance for the correct number of values in the pattern file. The event spec for the above example would contain two PatternSpecs. The PatternSpecs would look like:

```
PatternSpec[0] {
```

```
    type = INPUT;
    to_layer = FIRST;
    n_vals = 2;
};

PatternSpec[0] {
    type = TARGET;
    to_layer = LAST;
    n_vals = 1;
};
```

So that the first two values (n_vals = 2) will be read into the first (input) pattern, and the third value (n_vals = 1) will be read into the last (output) pattern.

The `ReadText` function also allows comments in the .pat files, as it skips over lines beginning with `#` or `//`. Further, `ReadText` allows input to be split on different lines, since it will read numbers until it gets the right number for each pattern.

There is a special comment you can use to control the creation and organization of subgroups of events. To start a new subgroup, put the comment `# startgroup` before the pattern lines for the events in your subgroup (note that the `# endgroup` comments from earlier versions are no longer neccessary, as they are redundant with the startgroup comments – they will be ignored). For example, if you wanted 2 groups of 3 events you might have a file that looked like this:

```
# startgroup
p01 0 0 0
p02 0 1 1
p03 0 1 0
# startgroup
p11 1 0 1
p12 1 1 0
p13 1 1 1
```

`WriteText` simply produces a file in the above format for all of the events in the environment on which it is called. This can be useful for exporting to other programs, or for converting patterns into a different type of environment, one which cannot be used with the CopyTo or CopyFrom commands. For example if events were created originally in a TimeEnv environment, but you now want to use them in a FreqEnv frequency environment, then you can use WriteText to save the events to a file, and then use ReadText to read them into a `FreqEnv` which will enable a frequency to be attached to them.

For Environments that are more complicated than a simple list of events, it is possible to use CSS to import text files of these events. Example code for reading events structured into subgroups is included in the distribution as '`css/include/read_event_gps.css`', and can be used as a starting point for reading various kinds of different formats. The key function which makes writing these kinds of functions in CSS easy is `ReadLine`, which reads one line of data from a file and puts it into an array of strings, which can then be manipulated, converted into numbers, etc. This is much like the '`awk`' utility.

The `read_event_gps.css` example assumes that it will be read into a `Script` object in a project, with three `s_args` values that control the parameters of the expected format.

Note that these parameters could instead be put in the top of the data file, and read in from there at the start.

The binary read/write functions (`ReadBinary`, `WriteBinary`) simply read and write a stream of floating point numbers in native binary format. These files are not necessarily portable, but are much more compact for large data sets.

11.8 Environment Generation Functions

The following functions are available on the **Environment** object (in the *Generate* menu) for algorithmically creating events. See '`demo/bp_misc/gen_rnd_prototypes.css`' for a CSS script that creates prototypes and random exemplars based on these prototypes using some of the following functions.

`ReplicateEvents(int n_replicas, bool make_groups)`
: This will create n_replicas of all of the existing events in the environment. If make_groups is set, then all events which are replicas of a given one will be placed into the same sub-group. Otherwise, the replicas appear right after the event they are replicas of. This can be used to replicate a set of prototype events, and noise can be added (AddNoise, FlipBits) to create random distortions of these prototypes.

`PermutedBinary(int pat_no, int n_on)`
: This produces random (permuted) binary patterns of 1's and 0's over the existing events in the environment. `pat_no` determines which pattern to use, and `n_on` specifies the number of 1's in each random pattern.

`PermutedBinary_MinDist(int pat_no, int n_on, float dist, bool max_correl)`
: This is like PermutedBinary, but it ensures that the hamming distance between any two patterns is at least `dist` (i.e., all patterns are separated by at least this minimum distance). If max_correl is set, this ensures that the maximum correlation between patterns is below the `dist` value.

`FlipBits(int pat_no, int n_off, int n_on)`
: This switches exactly `n_off` values of a pattern from 1 to 0, and `n_on` from 0 to 1. In other words, some values that were 1 will now be 0, and some values that were 0 will now be 1. This is a useful way of creating random distortions of prototypical patterns.

`FlipBits_MinMax(int pat_no, int n_off, int n_on, float min_dist, float max_dist, metric, norm, tol)`
: This does FlipBits, but if the new event that was just flipped is outside the distance limits set by min_ and max_dist, according to the metric supplied, then another pattern is tried. There is a timeout and an error message will be reported if it takes too many retries to fit these criteria.

`Clear(int pat_no)`
: Simply clears out the pattern values to a specified value (e.g., zero).

`AddNoise(int pat_no, Random rnd_spec)`
: This will add noise of the given specification to all events in pattern `pat_no`.

`TransformPats(int pat_no, PreProcessVals trans)`
: This will apply given transformation (simple math) operation to all events in pattern `pat_no`.

11.9 Environment Analysis Functions

The following functions are available on the **Environment** object (in the *Analyze* menu) for analyzing the patterns in the environment. Some of these functions can also be called from the **Log** objects in their own *Analyze* menu (see Section 13.3.1 [log-views-logview], page 212), and by the **DispDataEnvProc** process that automatically analyzes data collected in an environment from a statistic (see Section 12.4.5 [proc-special-misc], page 199).

`DistMatrix(ostream& strm, int pat_no, metric, norm, tol, format, precision)`
: This produces a distance matrix between all events for given pattern to a file specified by strm. The metric and associated parameters determine how distance is computed. Norm is whether to normalize entire distance, and tol is a unit-wise tolerance, below which dist = 0. The output format can be varied according to the format and precision parameters.

`DistMatrixGrid(GridLog* disp_log, metric, norm, tol)`
: This produces a distance matrix between all events for given pattern, and sends it to the given GridLog for immediate display (if NULL, a new GridLog is created).

`CmpDistMatrix(ostream& strm, int pat_no, Environment* cmp_env, int cmp_pat_no, metric, norm, tol, format)`
: This produces a distance matrix between all events across two different environments, for given pattern on each.

`CmpDistMatrixGrid(ostream& strm, int pat_no, Environment* cmp_env, int cmp_pat_no, metric, norm, tol, format)`
: This produces a distance matrix between all events across two different environments, for given pattern on each, and displays the results in a grid log.

`ClusterPlot(GraphLog* disp_log, int pat_no, metric, norm, tol)`
: This produces a cluster plot of the distance matrix between all events for given pattern, and sends it to the given GraphLog for immediate display. The clusterplot recursively groups the most similar items together, and uses the average of the individual cluster element distances to compute the distance to a cluster, instead of computing an average vector for a cluster and using that to compute distances. This makes the algorithm somewhat more computationally intensive, but also produces nicer results (the distance metric is cached so distances are only computed once, so it should handle relatively large numbers of events).

`CorrelMatrixGrid(GridLog* disp_log, int pat_no)`
: Generates a correlation matrix for all patterns in `pat_no` in the environment and plots the result in grid log (NULL = new log). The correlation matrix is defined by taking a vector of each pattern value (e.g., the first element in the pattern) across all events, and computing the correlation of this vector of values with those of all the other such value vectors. As such, it measures the degree

to which the variation of one pattern value (across events) correlates with the variation of another pattern value (across events).

`PCAEigenGrid(GridLog* disp_log, int pat_no, bool print_eigen_vals)`
Performs principal components analysis (PCA) of the correlations of patterns in `pat_no` across events, plotting all eigenvectors in the grid log (NULL = new log). The eigenvectors are the orthogonal components of the correlations ordered in terms of how much variability they account for. The first principal component (eigenvector) accounts for the greatest amount of variability in the correlation matrix, etc. Note that the eigenvectors are arranged from left-to-right, bottom-to-top, such that the first component is at the bottom-left.

`PCAPrjnPlot(GraphLog* disp_log, int pat_no, int x_axis_component, int y_axis_component, bool print_eigen_vals)`
Performs principal components analysis of the correlations of patterns in `pat_no` across events, and then plots projections of all the patterns onto two of the principal components in the graph log (NULL = new log). This provides a means of representing the similarities of the different patterns with each other in a two-dimensional space. The X-axis of the graph represents how similar the patterns are along the first selected component, and the Y-axis represents how similar they are along the second selected component.

`MDSPrjnPlot(GraphLog* disp_log, int pat_no, int x_axis_component, int y_axis_component, metric, norm, tol, bool print_eigen_vals)`
Performs multidimensional scaling (MDS) on the distance matrix (computed according to metric, norm, tol parameters) of patterns in pat_no across events in the graph log (NULL = new log). Multidimensional scaling takes a set of dissimilarities and returns a set of points such that the distances between the points are approximately equal to the dissimilarities. The results of this are typically very similar to PCA, even though the method is somewhat different, operating on the distance matrix instead of the correlation matrix. The classical (metric) form of MDS is used, computed according to the algorithm developed by Torgerson, W. S. (1958). *Theory and Methods of Scaling.* New York: Wiley, as extracted from the '`cmdscale.R`' function from the **R** statistical program.

`EventPrjnPlot(Event* x_axis_event, Event* y_axis_event, int pat_no, GraphLog* disp_log, metric, norm, tol)`
Projects all events according to their smiliarity to the two specified events using given distance metrics. This allows one to obtain a 2-d plot of the similarity of all events to two selected events.

`EnvToGrid(GridLog* disp_log, int pat_no, ev_x, y, pt_x, y)`
This dumps the entire set of events worth of a given pattern into a grid log for more efficient viewing and for more control over the display layout. The ev_x, ev_y parameters control the geometry of the events, and the pt_x, pt_y control the geometry of the pattern values within each event.

`PatFreqText(float act_thresh, bool proportion, ostream& strm)`
Reports frequency (proportion) of pattern values greater than act_thresh across events, to a text output (this is most useful if pattern values are named in

value_names). This can be useful for verifying that an environment has expected probabilities of different inputs.

`PatFreqGrid(GridLog* disp_log, float act_thresh, bool proportion)`
Reports frequency (proportion) of pattern values greater than act_thresh across events, to a grid log (if NULL, new log is made).

`PatAggText(Aggregate& agg, ostream& strm)`
Reports aggregate (SUM, AVG, etc) of pattern values across events, to a text output (this is most useful if pattern values are named in value_names). This can be useful for identifying the overall properties of events.

`PatAggGrid(GridLog* disp_log, float act_thresh, bool proportion)`
Reports aggregate (SUM, AVG, etc) of pattern values across events, to a grid log (NULL = make a new log). This can be useful for identifying the overall properties of events.

`EventFreqText(bool proportion, ostream& strm)`
Reports frequency (proportion) of event names in the environment. Also useful for validating that the environment contains what you think it should.

11.10 Frequency Environments and Events

In the basic model of the environment, all events are equally likely to be presented to the network. However, this is often not the case in the real world. Thus, PDP++ provides a set of environmental types that implement frequency-based presentation of events. These are the **FreqEvent**, the **FreqEvent_MGroup**, and the **FreqEnv** types. As discussed above (see Section 11.3 [env-seq], page 169, Section 11.1 [env-env], page 163), processes can ask the Environment for a single Event or a Group of Events. The FreqEvent is an individual event with its own frequency, and the FreqEvent_MGroup is a group of events with an associated frequency.

The frequency model supported by these types has a number between zero and 1 associated with each event or event group. This `frequency` value is essentially a probability with which the event should be presented.

A **FreqEnv** environment type must be used in order for the frequency variable associated with events or groups to be used. This type of environment uses the `InitEvents` function to create the list of events or groups that will be presented for the current epoch. There are two modes of selecting these events according to their frequency. Setting `sample_type` to `RANDOM` results in a random sample of events to be presented for each epoch, with the probability of an event's inclusion proportional to its frequency value. The `PERMUTED` option instead adds a fixed number of a given event which is equal to the frequency of that event times the `n_sample` parameter.

The **FreqEnv** class also has the following variables which affect this sampling process:

`int n_sample;`
The number of samples of the Events or EventGroups to make per epoch. In the `RANDOM` case, this is the number of times to "roll the dice" to determine if a given event gets into the epoch. An event or group with a frequency of 1 will appear exactly `n_sample` times in a given epoch. Events with lower frequency

will occur less often and with lower probability. In the `PERMUTED` case, this number is multiplied times the frequency to determine how many copies of an event or group appear in a given epoch.

`FreqLevel freq_level`
: Has the values `NO_FREQ`, `EVENT`, or `GROUP`. It controls the level at which the frequency sampling occurs. If `NO_FREQ` is selected, frequency is ignored and the environment is just like the basic one. The other two options should be selected depending on the type of process that is being used–use `GROUP` for sequence-based processes (see Section 11.3 [env-seq], page 169), and otherwise use `EVENT`.

Note that there is also a **FreqTimeEvent**, **FreqTimeEvent_MGroup**, and **FreqTimeEnv**, which are frequency versions of the time-based environments described below.

11.11 Other Environment Types

There are a couple of other types of environments that embelish the basic model described above. Two of these are particularly appropriate for specific algorithms, and are therefore described in the context of these algorithms. Thus, for information on the environment types that associate a particular time with each event (**TimeEnvironment**, **TimeEvent_MGroup**, and **TimeEvent**), see the recurrent backpropagation part of the manual: Section 14.2.5 [rbp-seq], page 240. However, note that the **CsSettle** process (see Section 15.4 [cs-proc], page 249) will use the `time` value for determining how long to settle, which is different from the way this variable is used in RBp. For information about the environment types that allow individual patterns within an event to be chosen probabilistically (**ProbEventSpec**, **ProbPatternSpec_Group**, **PropPattern**), see the constraint satisfaction part of the manual: Section 15.7 [cs-prob-env], page 252.

The **XYPattern** and **XYPatternSpec** provide a mechanism for applying patterns that move around on a larger input layer. The offset of a pattern is specified by the `offset` member of the pattern, which can be updated (e.g., by a script) to move the pattern around on subsequent pattern presentations. The pattern spec provides options to determine the way in which the pattern is applied to a network layer:

`bool wrap` This controls whether to wrap the pattern around the network layer if it should extend beyond the maximum coordinate for the layer in either the x or y dimension. The alternative is that it is "clipped" at the maximum, and that portion of the pattern is simply not presented.

`bool apply_background`
: The portions of the network layer which do not receive input from the pattern can optionally be set to a particular "background" level, by setting this flag.

`float background_value`
: This is the value to set the background units to, if applicable.

The **ScriptEnv** is an environment with a script associated with it. The script is called at the beginning of the epoch, when the **EpochProcess** calls the `InitEvents` function on the environment. The **ScriptEnv** defines this function to run its script, which can contain code

to build an entire epoch's worth of events dynamically (i.e., as the epoch is starting). This is useful for environments which have a probabalistic character which is more complicated than simple frequency sampling. An example of a **ScriptEnv** is provided in the demo of a simple recurrent network, which learns to perform like a finite state automaton. The training events are generated probabalistically at run-time using a **ScriptEnv** from a script version of the automaton. This demo can be found in the '`demo/bp_srn`' directory.

Note that the **ScriptEnv**, like other script-based objects, provides an array of script args called `s_args` that can be used to control the behavior of the script and customize it for particular cases. The meaning of these arguments depends of course on the script that is being used, but they should be documented near the top of the script code.

The **InteractiveScriptEnv** is a script environment that works with the interactive model of event construction (See Section 11.1 [env-env], page 163 and Section 12.4.2 [proc-special-inter], page 197). The script is called when the **InteractiveEpoch** calls `GetNextEvent` on the environment, at the start of each trial. At this point, the script should generate a new event (based on the network's output to the prior event, for example), and set the `next_event` pointer to point to this event. If the epoch is over, then next_event should be set to NULL. The script can examine the `event_ctr` variable on the environment to determine where it is within the epoch – this value is reset to 0 at the start of the epoch.

For a working example of this technology, see '`demo/leabra/nav.proj.gz`' and its associated '`nav_env.css`' script which is attached to an InteractiveScriptEnv.

The **FromFileEnv** environment reads events one epoch of `events_per_epc` events (`read_mode = ONE_EPOCH`) or one single event (`ONE_EVENT`) at a time from a file (either text or binary format) for presentation to the network. This should be useful for very large environments or very large patterns, or both. Note that events_per_epc events will be loaded into RAM at a time from the file, so this number should not be set too large relative to available memory, etc. Reading one event at a time (ONE_EVENT – only one event in RAM at any time) uses the "interactive" interface to the environment (GetNextEvent) meaning that the **InteractiveEpoch** epoch process (see Section 12.4.2 [proc-special-inter], page 197) must be used.

12 Processes and Statistics

Processes in PDP++ play the role of orchestrating and coordinating the interactions between different object types in order to carry out particular tasks. Processes are *objects* (see Section 8.1.1 [obj-basics-obj], page 107), which means that they can have their own variables and functions. This is fundamentally different from the idea of a process as something that simply acts on a data structure. There are many different kinds of process objects which have been designed to perform specific kinds of tasks.

The scheduling processes coordinate the overall scheduling of processing during training and testing of networks, and are the "backbone" of processing system. They are organized into a hierarchy of objects, each of which performs a specific level of processing:

```
TrainProcess (loop over epochs)
    EpochProcess (loop over trials)
        TrialProcess (present a single pattern)
        .
        .
    .
    .
```

The rest of the processes hang off of the schedule processes in different places, and perform specific tasks. The largest category of such processes are *statistics*, which compute and record data about the network, environment or other processes, and make this data available to the logs for displaying.

Statistics often need to be viewed at multiple levels of processing. Thus, one often wants to simultaneously view the trial-wise and the aggregated epoch-summed squared error. This is accomplished in PDP++ by having statistics which actually compute the squared-error for a given event in the trial process, but also copies of this squared-error statistic at subsequent (higher) levels of processing which perform the *aggregation* of the statistic over these higher levels.

Since the different time-grains of processing are represented by the schedule process hierarchy, other things like updating displays and logging data, which can also happen at different time-grains, are tied to the schedule process hierarchy. Thus, one can decide to log data at the **TrialProcess**, **EpochProcess**, and/or **TrainProcess** level, and similarly for updating displays like the network viewer.

The code to control a Process can either be hard-coded C++ code, or a pointer to a CSS script file. This means that one can change the behavior of a Process simply a writing a CSS script and setting the Process to use it (see Section 12.7 [proc-css], page 208).

The best way to interact and configure processes is throught the project viewer (see Section 9.2 [proj-viewer], page 120). Processes edit dialogs are yellow (`SchedProcess`) gold (regular `Process`) or slate blue (`Stat`) in the default color scheme.

The **EpochProcess** can also coordinate the processing of different events on different *distributed memory processors* to achieve much faster processing on parallel hardware (see Section 12.3.3.1 [proc-epoch-dmem], page 193).

12.1 The Basic Features of all Processes

The base Process class provides the following variables for controlling its behavior:

`Enum type` This can be either `C_CODE` or `SCRIPT`, which determines if the Process is going to execute C code or CSS script code. A script file must be attached to the process to run in `SCRIPT` mode.

`Modulo mod`
This object controls how often a process is run. It is applicable for processes and statistics that are placed in either the `loop_procs` or `loop_stats` groups of a scheduling process, and the modulo function is based on the counter variable of that parent process. For `final_stats` and `final_procs`, the counter variable is that of the parent of the parent schedule process. If the `flag` variable of the `mod` object is not set then the process is never run. Otherwise the process is run if the parent's counter minus the `off` variable modulo the `m` variable is equal to zero (i.e., it is run every `m` times, with a phase determined by the offset `off`).

`Network* network`
This is a pointer to the Network object on which the Process is acting. This pointer is copied automatically from the parent process, so it should be changed only at the highest level of the processing hierarchy, which will cause it to change in all the lower-level processes.

`Environment* environment`
This is a pointer the Environment in which the Process acts. Like the network pointer, it is automatically copied from higher-level processes, and should be set at the highest level.

The basic Process class provides the following functions for controlling its behavior. Some of these are available on the *Control Panel* buttons that appear at the bottom of the edit dialog and the control panel, and others are in the *Actions* menu.

`NewInit()`
Initializes the process using new random seed. This seed is saved, and can be recalled using the `ReInit` function, but the previously saved seed is then lost.

`ReInit()` Initializes the process using the previously-saved random seed.

`Run()` This function checks which type of code the Process is supposed to use (C code or script) and executes the appropriate code.

`Step()` Executes one step the process. For schedule processes, this is controlled by the `step` field, which specifies at what sub-level of the heirarchy to step (e.g., setting step.proc to the trial process will mean that Step executes one step of the trial process – it processes one event per Step). Schedule processes also have **Step Up** and **Step Dn** buttons that are useful for controlling the stepping level within the process hierarchy.

`Stop()` Stops the process when running.

`Step Up()` Moves the stepping level up one step in the heirarchy (e.g., from cycle up to settle). Removes updating of the network by the previous stepping level (e.g., cycle no longer updates). This only applies to Sched Processes.

`Step Dn()` Moves the stepping level down one step in the heirarchy (e.g., from settle to cycle). Adds updating of the network by the new stepping level (e.g., cycle now updates the network). This only applies to Sched Processes.

`ControlPanel()`
Brings up a small control panel dialog for running the process.

`LoadScript (char* filename)`
Sets the script file to be used by the Process to `filename` and compiles the script for execution. This function clears any previous script file being used by the process. It automatically sets the `type` variable to `SCRIPT`.

12.2 The Schedule Process (SchedProcess)

Instead of putting all the control necessary to iterate over the several levels of processing needed to train a network (training, epochs, trials, settles, cycles, etc.) into one process object, we have divided up the processing hierarchy into a set of nested scheduling processes, each of which handles one level of processing. This results in a lot of flexibility, since one can then change only the code controlling the trial, for example, or easily extend it using a CSS script.

The Schedule Process (or **SchedProcess**) is an extension of the basic Process type. It provides more variables and functions with which one can control the execution of the Process. It has fields for the parent and child processes in its hierarchy, support for counters that control the iteration of the process over time, places to link in **View** objects and **Log** objects to be updated, and groups to hold the various sub-processes and statistics that can be associated with a given level of processing.

In order to support all of its extended functionality, the schedule process has a somewhat complicated execution structure. However, understanding how a schedule process runs should make it easier to figure out how to get them to do what you want them to.

The central function a schedule process performs is one of looping, where the process repeated performs some function. In most cases, this function simply involves telling the process below it in the hierarchy to run. Thus, an epoch process repeatedly loops over the trial process, for example. The functions in a schedule process center around the main loop of processing,

The main loop is written so as to be re-entrant. Thus, something can cause the process to pop out of the loop (i.e., the user pressing *Stop*), and when it runs again, it will fall back down to the point where it was last running and pick up again where it left off.

The places where the things that hang off of a schedule process, like statistics, logs and displays, can all be seen in the main schedule process loop code, which is reproduced here:

```
void SchedProcess::C_Code() {
  bool stop_crit = false;        // a stopping criterion was reached
  bool stop_force = false;       // either the Stop or Step reached

  if(re_init) {                 // if its time to re-initialize, then do it
    Init();                      // this sets the counters to zero, etc.
    InitProcs();                 // this runs any initialization processes
```

```
  }

  do {
    Loop();                      // user defined code goes here
    if(!bailing) {
      UpdateCounters();          // increment the counters (before logging)
      LoopProcs();               // check/run loop procs (use mod of counter)
      LoopStats();               // update in-loop statistics
      if(log_loop)               // can log inside loop or after it...
        UpdateLogs();            // generate log output and update logs
      UpdateState();             // update process state vars (current event..)

      stop_crit = Crit();        // check if stopping criterion was reached
      if(!stop_crit) {          // if at critera, going to quit anyway, so don't
        stop_force = StopCheck(); // check for stopping (Stop or Step)
      }
    }
  }
  while(!bailing && !stop_crit && !stop_force);
  // loop until we reach criterion (e.g. ctr > max) or are forcibly stopped

  if(stop_crit) {              // we stopped because we reached criterion
    Final();                    // user defined code at end of loop
    FinalProcs();               // call the final procs
    FinalStats();               // run final_stats at end of loop
    if(!log_loop)
      UpdateLogs();             // if not logging in loop, logging at end
    UpdateDisplays();           // update displays after the loop
    SetReInit(true);            // made it through the loop, so Init next time
    FinalStepCheck();           // always stop at end if i'm the step process
  }
  else {                        // we we're forcibly stopped for some reason
    bailing = true;             // now we're bailing out of all further procs
  }
}
```

The fall-through character of processing is made possible by storing all of the counter state variables on the process object itself, so it is preserved even when we pop out of the loop, and by only initializing once we make it through the loop (by setting the `re_init` flag).

As you can see, there are two places where statistics get updated, inside the loop (`LoopStats()`) and after the loop (`FinalStats()`). While it is more natural to think of computing statistics at the end of the loop (e.g., at the end of the trial or the end of the epoch), the need to aggregate statistic values over time (e.g., compute the sum of the squared-errors over an entire epoch) necessitates inside-the-loop statistics.

Finally, it should be noted that all schedule processes are written to allow any number of intervening processes to be added in to the hierarchy at any point. Thus, new and unanticipated levels of processing can be introduced by the user without breaking the assumptions

of the existing process objects. Basically, any time there is a dependency of one level of processing on another (e.g., the trial process looks to its parent epoch process to determine if it should be testing or training the network), the dependent process searches through the entire hierarchy for the type of process it depends on.

12.2.1 Variables and Functions used in a SchedProcess

The SchedProcess provides the following variables and functions (in addition to the ones already provided by the basic Process):

Variables

`bool can_stop`
: This variable is a flag that controls whether the SchedProcess can be interrupted during execution. This also determines if GUI events are being processed. Things can be speeded up a bit if this flag is turned off for low-level processes like the `CycleProcess`.

`TypeDef sub_proc_type`
: This is type of process the `sub_proc` should be. If the `sub_proc` type does not inherit from this type then a new `sub_proc` is created of the correct type.

`SchedProcess* sub_proc`
: This is a pointer to the child Process (if it exists) of this SchedProcess. This is the sub process that the process iterates over.

`StepParams step`
: It is possible to single-step (or multiple step) through processing, and this controls how the *Step* function behaves. The `step` variable contains a pointer to the sub-process under the current one that represents the time-grain at which stepping should occur. The number of times this step process is iterated per each *Step* (i.e., each time the user hits the *Step* button) is determined by the `n` parameter.

`Stat_Group loop_stats`
: This is a group that contains the Statistic processes that will be executed within the loop of the schedule process (i.e., called by `LoopStats()` in the loop code shown above). Thus, for a epoch process, these stats will be computed after every trial, since the epoch process loops over trials. These are typically aggregation stats, which are adding up values computed in the trial process. For example the epoch sum of squares error would be aggregated in the epoch loop stats.

`Stat_Group final_stats`
: This is a group that contains the Statistic processes that will be executed at the end of the loop for this process. This is typically where statistics go which are computed for the first time (i.e., not those that are simply aggregating values computed lower down in the hierarchy).

`Process_Group init_procs`
: This contains miscellaneous processes that get executed when the process is initialized. Note that these are run only when the process is actually running, *not* when the *ReInit* or *NewInit* buttons are hit. Thus, if you hit one of these buttons, and then do a *Run*, the first thing that will happen when the process is run is that the init_procs will be executed.

`Process_Group loop_procs`
: These are miscellaneous processes that get executed inside the loop of the process. For example, it is possible to link in a testing epoch process into the `loop_procs` of a training **TrainProcess**, with a mod value set to 10, for example, which will result in the network being tested after every 10 epochs of training.

`Process_Group final_procs`
: These are miscellaneous processes that get executed after the loop of the process, just before the final stats are computed.

`bool log_loop`
: Either the process sends its data at the end of its processing loop, which is the "natural" (and default) way to do things, since it corresponds with the name of the process (the end of the epoch process means once every epoch, while the loop of the epoch process is actually the end of every trial!), or it sends its data inside the loop, which can be useful to see the aggregation of the `loop_stats` statistics over time. This flag, if checked, means that it logs inside the loop.

`bool log_counter`
: This flag determines if the counter associated with this process (e.g., the epoch counter in the **TrainProcess**) is logged along with all the other data that is logged.

Many of the core functions on the schedule process object were documented in the main loop code shown previously. In addition to the functions on the process object, the following functions are available in a schedule process (most can be found in the *Actions* menu of the edit dialog or control panel).

`InitMyLogs()`
: Clear all logs that this process updates.

`InitAllLogs()`
: Clear all logs that exist in the Project that this SchedProcess is in.

`InitNetwork()`
: Initialize the weights in the network associated with this process (calls `InitWtState()` on the network).

`InitAll();`
: Initialize the process, network weights, and logs.

`RemoveFromLogs();`
: Remove this SchedProcess from all the logs in the `logs` group and clear out the `logs` group.

`RemoveFromDisplays();`
: Remove this SchedProcess from all the displays in the `displays` group and clear out the `displays` group.

`CheckAllTypes();`
: This goes through all the objects in the network and makes sure that they are all of the minimum type necessary for all of the processes statistics being computed by this processing hierarchy. This is done automatically whenever the training process is initialized, but it can be done manually just to make sure. This check is useful, especially if you are experiencing unexplained crashing, because many process objects assume that the objects in the network are of the appropriate type.

The following are a set of functions that are particularly useful for configuring the processing hierarchy, and appear in the *Structure* menu of a SchedProc.

`MoveToSubGp(const char* gp_name)`
: This moves the current process and all of its sub-processes to a new sub-group within the `.processes` group of the project. Run this on the top-level process in a process hierarchy. This is useful for organizing the menu when several processing hierarchies are present in the same project.

`ChangeNameSuffix(const char* new_name_sufx)`
: This changes the portion of the sched process names after the underbar (_) to the new specified name. This is the preferred way to give a whole hierarchy of sched procs the same semantically meaningful tag (e.g., a suffix of "Trn" for training processes, and "Tst" for testing processes.) Run this on the top level process, as it works on all sub-processes.

`AddSuperProc(TypeDef* type)`
: This will add a new schedule process above this one, of the given type, while preserving as much of the existing structure as possible. Thus, any aggregated stats will be aggregated through the new super proc, and it is inserted so that any previous super proc is now the super proc to the new super proc, etc.

`AddSubProc(TypeDef* type)`
: Like AddSuperProc, but adds a new process below this one.

`RemoveSuperProc(TypeDef* type)`
: This is effectively the inverse of AddSuperProc – removes parent process and closes up any existing aggregation links, etc.

`RemoveSubProc(TypeDef* type)`
: This is effectively the inverse of AddSubProc – removes sub process and closes up any existing aggregation links, etc.

12.2.2 Statistics and Logging in a SchedProcess

The relationship between statistics and logging in schedule processes is fairly straightforward, but there are some subtleties. Basically, a schedule process sends two kinds of data to the log. The first is a record of the current state of all the counters in the process

hierarchy above (and if `log_counter` is checked, from) the logging process itself. This tags the log data with the point in time when it was computed. The other component is a series of columns of data that are generated by each of the statistic processes in either the loop or final statistics groups.

This information is sent out to any logging processes that are being updated by the schedule process in question. These logging processes then transform the data into the graphical form characteristic of the display they are using (e.g. a graph or a grid of color squares), and/or dump it to a log file, etc (see Chapter 13 [log], page 210).

Thus, the way to get a log to record some information is to have a statistic process which collects the information and sends it along to the log when the schedule process tells it to. For this reason, when you want to monitor unit state variables over time, for example, you have to create a statistic process which gets these state variables, and sends them to the log (see Section 12.6.2 [proc-stats-monitor], page 203).

12.3 Schedule Processes for Different Time-Grains

Each schedule process handles processing at a different time grain. The following time grains are supported by a standard set of schedule processes. Note that a processing hierarchy typically has to start with at least an epoch process since that is what gets the events from the environment, and the lower-level processes look to the epoch process for the current event.

12.3.1 Iterating over Networks: BatchProcess

The **BatchProcess** iterates over the training of networks. This is useful in determining the average learning time for a given problem with different random initial weights, for example. The batch process has a `batch` counter object, which records the number of networks that have been trained so far. The appropriate sub-process type for a batch process is a **TrainProcess**, though it is possible to have multiple batch processes before the train process, in order to have multiple loops of network training (presumably with some parameter manipulation in between).

There is a built-in batch process type that makes it easy to perform simple searches of parameter space called the **GridSearchBatch**. This process increments a single parameter value in step with the batch counter, and applies this value to any parameter of the user's choosing. The parameter to be modified is specified by giving a CSS-style path to that parameter from the common project object (see Section 7.2.3 [css-tut-access], page 80 for details). An example project which uses this grid search batch is '`demo/bp/gridsearch_xor.proj.gz`', which can be consulted to see how it works in practice. It also records the current value of the parameter to the log file.

12.3.2 Iterating over Epochs: TrainProcess

The **TrainProcess** iterates over epochs of training a network. It typically has an **EpochProcess** as its `sub_proc`. When this process is initialized (e.g. by `ReInit` or `NewInit`), it also initializes the weights of the network. It has an `epoch` counter which is tied to the `epoch` counter on the network object, which this process increments after every epoch of

training. Note that if the epoch process under this training process is in `TEST` mode, then neither epoch counter is incremented.

There is an alternative kind of process which also iterates over epochs, called the **NEpochProcess**, which differs from the **TrainProcess** in that it does not initialize the network when it is initialized. Also, it keeps its own `epoch` counter separate from that of the network. Thus, while it will increment the network's counter during training (but not turing testing), it *will* increment its epoch counter even during testing. Thus, it is useful for cases where you need to run multiple epochs of testing (e.g., to get multiple samples from settling in a stochastic network).

12.3.3 Iterating over Trials: EpochProcess

The **EpochProcess** loops over the set of Events in the Environment (see Chapter 11 [env], page 163). Each presentation of an event is known as a *trial*, and this process typically has a **TrialProcess** as its `sub_proc`, although the situation is different when the environment contains sequences of events (see Section 12.4.1 [proc-special-seq], page 195).

The epoch process is responsible for ordering the presentation of events to the network. Thus, at the beginning of the epoch (when the process is initialized), it tells the environment to initialize itself (using `InitEvents()`), and then obtains the total number of events in the environment (using `EventCount()`, see Section 11.1 [env-env], page 163). The epoch process then makes a list of event indexes, which represents the order in which events will be presented. Depending on the state of the `order` variable, this list will either remain sequential or be randomized.

The epoch process is also responsible for determining when to update the weights in the network, since this can usually be done either after each event or at the end of the epoch (depending on the state of the `wt_update` variable). The epoch process itself calls the `UpdateWeights` function on the network, even when it is doing updates after each event. Thus, lower-level processes should never call this function themselves.

Also see the following information about Epoch Processes:

The following variables are on the **EpochProcess**:

Variables

`Counter trial`
: The number of the current trial being executed. This is the counter for the epoch process. It is automatically initialized to be the number of events in the environment.

`Event* cur_event`
: This is a pointer to the current event being processed. After each trial, the EpochProcess updates this variable to point to the next event based on its list of event indexes. It gets the event from the environment using the `GetEvent` function of the environment (see Section 11.1 [env-env], page 163).

`Order order`

Controls the order in which Events are presented to the network. The values for this are:

`SEQUENTIAL`
: Present events in sequential order (i.e. in the order they currently are in the Environment `events` group).

`PERMUTED`
: Present events in permuted order. This ensures that each event is only presented once per epoch, but the order is randomized.

`RANDOM`
: This picks an event at random (with replacement) from the list of events. This does not use the epoch process's list of events, and it allows the same event to be presented multiple times in an epoch, while other events might not be presented at all.

`WtUpdate wt_update`

Determines when the network's weights are updated (if at all). The possible values are:

`TEST`
: Don't update weights at all (for testing the network). This also causes the training process to not increment the epoch counter of the network after each epoch, since the epoch counter is supposed to reflect the extent of training experience the network has had.

`ON_LINE`
: Update the weights on-line (after every event).

`BATCH`
: Update the weights after every epoch (batch mode).

`SMALL_BATCH`
: Update the weights after every `batch_n` events. This allows an intermediate level of batch mode learning which can be parameterized independent of the number of events in the epoch.

`int batch_n`

Specifies the number of events between weight updates if `wt_update` is `SMALL_BATCH`.

12.3.3.1 Distributed Memory Computation in the EpochProcess

The EpochProcess supports distributed memory (*dmem*) computation by farming out events across different distributed memory processors. For example, if you had 4 such processors available, and an environment of 16 events, each processor could process 4 of these events, resulting in a theoretical speedup of 4x.

In all dmem cases (see Section 10.1 [net-dmem], page 134 for Network-level dmem) each processor maintains its own copy of the entire simulation project, and each performs largely the exact same set of functions to remain identical throughout the computation process. Processing only diverges at carefully controlled points, and the results of this divergent processing are then shared across all processors so they can re-synchronize with each other. Therfore, 99.99% of the code runs exactly the same under dmem as it does under a single-process, making the code extensions required to support this form of dmem minimal.

If learning is taking place, the weight changes produced by each of these different sets of events must be integrated back together. *This is means that weights must be updated in SMALL_BATCH or BATCH mode when using dmem.*

Epoch-wise distributed memory computation can be combined with network-wise dmem (see Section 10.1 [net-dmem], page 134). The Network level `dmem_nprocs` parameter determines how many of the available processors are allocated to the network. If there are multiples of these numbers of processors left over, they are allocated to the Epoch-level dmem computation, up to a maximum specified by the EpochProcess `dmem_nprocs` (which defaults to 1024, essentially saying, take all the remaining processors available). For example, if there were 8 processors available, and each network was allocated 2 processors, then there would be 4 sets of networks available for dmem processing of events. Groups of two processors representing a complete network would work together on a given set of events.

If `wt_update` is set to `BATCH`, then weights are synchronized across processors at the end of each epoch. Results should be identical to those produced by running on a single-processor system under BATCH mode.

If `wt_update` is `SMALL_BATCH`, then the `batch_n` parameter is *divided* by the number of dmem processors at work to determine how frequently to share weight changes among processors. If `batch_n` is an even multiple of the number of dmem processors processing events, then results will be identical to those obtained on a single processor. Otherwise, the effective batch_n value will be different. For example, if there are 4 dmem processors, then a value of batch_n = 4 means that weights changes are applied after each processor processes one event. However, batch_n = 6 cannot be processed in this way: changes will occur as though batch_n = 4. Similarly, batch_n = 1 actually means batch_n = 4. If batch_n = 8, then weight changes are applied after every 2 sets of dmem event processing steps, etc.

Note that `wt_update` cannot be `ONLINE` in dmem mode, and will be set to `SMALL_BATCH` automatically by default.

For the **SequenceEpoch** process in `SMALL_BATCH` mode, weight updates can occur either at the `SEQUENCE` or `EVENT` level as determined by the `small_batch` field setting. At the sequence level, each processor gets a different *sequence* to process (instead of a different event), and weight changes are shared and applied every `batch_n` *sequences* (subject to the same principles as for events as just described above, to maintain equivalent performance in single and dmem processing modes). At the event level, each processor works on a different event within the sequence, and weight changes are applied every batch_n events as in a normal epoch process. In addition, it is guaranteed that things are always synchronized and applied at the end of the sequence.

Note that the event-wise model may not be that sensible under dmem if there is any state information carried between events in a sequence (e.g., a SRN context layer or any other form of active memory), as is often the case when using sequences, because this state information is NOT shared between processes within a sequence (it cannot be – events are processed in parallel, not in sequence).

12.3.4 Presenting a Single Event: TrialProcess

The **TrialProcess** executes a single trial of processing, which corresponds to the presentation of a single **Event** to the network. This process is never used in its base form. Instead,

different algorithms derive versions of this process which perform algorithm-specific computations.

The trial process obtains the current event to be processed from the epoch process, which it finds somewhere above it in the processing hierarchy. It keeps a pointer to this event in its own `cur_event` member. It also typically depends on the epoch process `wt_update` field to determine if it should be computing weight changes or just testing the network.

Some types of **TrialProcess** objects are terminal levels in the processing hierarchy, since they perform all of the basic computations directly on the network. This is true of feed-forward backpropagation, for example (see Section 14.1.4 [bp-proc], page 229). However, other types of trial process have sub-processes which perform iterative setting and cycling processes, which are described below. This is true of the constraint satisfaction trial process, for example (see Section 15.4 [cs-proc], page 249).

12.3.5 Iterating over Cycles: SettleProcess

The **SettleProcess** is a base type of process that is used to iterate over cycles of activation updating. Thus, it typically has a **CycleProcess** as its sub-process. In algorithms with recurrent connectivity, it is typically necessary to iteratively update the activation states of the units for some number of cycles. This process controls this settling procedure. Particular algorithms will derive their own version of the settle process.

The `cycle` counter records the number of cycles of updating that have been performed. Setting the `max` for this counter will limit settling to this number of cycles. In addition, some algorithms use a `loop_stat` that measures the change in activation. When this stat goes below its criterion threshold, the settle process will stop. Thus, the stat determines when the settling has reached an equilibrium state.

12.3.6 Performing one Update: CycleProcess

The **CycleProcess** performs algorithm-specific updating functions. It processes a single cycle of activation updating, typically. It is usually a sub-process of the **SettleProcess**. It is almost always a terminal level of processing (it has no sub-processes).

12.4 Specialized Processes

There are a number of specialized versions of the standard schedule processes, and other useful process objects for automating routine tasks. These are discussed in detail in the following sections.

12.4.1 Processes for Sequences of Events

As discussed in Section 11.3 [env-seq], page 169, environments can be constructed that specify sequences of events. Certain algorithms can learn temporal contingencies between events, so it is important to be able to present the events in the proper sequence. There are two specialized types of schedule processes that handle the presentation of sequences of events to the network, the **SequenceEpoch**, and the **SequenceProcess**.

The **SequenceEpoch** is a version of the epoch process (see Section 12.3.3 [proc-levels-epoch], page 192) which, instead of iterating over individual events, iterates over *groups* of events. Thus, this process uses the `GroupCount()` and `GetGroup()` functions on the environment instead of the event-wise ones (see Section 11.1 [env-env], page 163). Each group of events represents a collection of events that form a sequence. The sequence epoch adds a `cur_event_gp` field, which contains a pointer to the current event group that is being processed.

See Section 12.3.3.1 [proc-epoch-dmem], page 193 for how these processes operate under distributed memory parallel processing.

The `order` field on the sequence version of the epoch process now refers to the order of presentation of the groups (sequences) of events, and not the order of individual events within the sequences. Also, the `SMALL_BATCH` mode of `wt_update` in the sequence epoch can now take on one of two different possible meanings, depending on the state of the `small_batch` field. In `SEQUENCE` mode, it means that weight changes are applied after `batch_n` sequences are processed. In `EVENT` mode, it means that weight changes are applied after every `batch_n` events within the sequence (as in a standard EpochProcess). Also, an additional weight update is performed at the end of the sequence in this mode to ensure that the weights are always updated at sequence boundaries, because the batch_n counter starts over at the start of each sequence.

The **SequenceProcess** is typically created as a child of the **SequenceEpoch**, and it is the one that iterates over the particular events within a given group or sequence. It obtains the current group of events from its parent sequence epoch process, and iterates over them. It can control the order of presentation of events within the sequence, and has options for initializing the activation state of the network at the start of the sequence:

`Counter tick`
: Each presentation of an event within a sequence is called a "tick", and this member counts the number of ticks that have gone by. The `max` for this counter is automatically set to be the number of events in the current sequence.

`Event* cur_event`
: This is a pointer to the current event being processed.

`Event_MGroup* cur_event_gp`
: This is a pointer to the current event group (sequence) being processed. It is obtained from the parent **SequenceEpoch**.

`Order order`
: This determines the order of presentation of the events within a sequence. While sequences are usually presented in `SEQUENTIAL` order, it is conceivable that one might want `PERMUTED` or `RANDOM` orders as well, which are available (these work just like the equivalent in the epoch process, see Section 12.3.3 [proc-levels-epoch], page 192).

`StateInit sequence_init`
: This determines if and how the activation state of the network is initialized at the start of the sequence. `DO_NOTHING` means that no initialization is done, `INIT_STATE` means that the `InitState` function is called, and `MODIFY_STATE` means that the `ModifyState` function is called, which allows for algorithm-

specific ways of changing state between sequences (e.g. decaying the activations).

12.4.2 Processes for Interactive Environments

The **InteractiveEpoch** is a special epoch process for dealing with interactive environments (**InteractiveScriptEnv**, Section 11.11 [env-other], page 182), where events are created on the fly at the start of each new trial, instead of being created entirely at the beginning of the epoch (as with a **ScriptEnv**, Section 11.11 [env-other], page 182), or just being static, as with most environments.

This process calls `InitEvents()` on the `environment` at the start of the epoch, which resets the `event_ctr` on the environment to 0. It then calls `GetNextEvent()` on the environment at the start of each new trial. In standard environments, this just gets the event at index event_ctr, and increments the counter. If event_ctr is larger than the number of events, a NULL is returned indicating the end of the epoch. In the InterativeScriptEnv, GetNextEvent() calls the associated script, which can then create the next event and return it to the epoch process (via the next_event field in the environment).

See '`demo/leabra/nav.proj.gz`' for an example demonstrating the use of InteractiveEpoch and InteractiveScriptEnv.

12.4.3 Processing Multiple Networks/Environments

There are times when it is useful to be able to compare two different networks as they simultaneously learn the same task, or compare the learning of the same network on different environments. This can be accomplished by performing multiple streams of processing at the same time.

The **SyncEpochProc** runs two different sub-processes through the same set of events from a common environment. Thus, it can be used to train two different networks, even networks that use different algorithms, at the same time. It essentially just adds a set of pointers to a `second_network` and a `second_proc_type` and `second_proc`, which identify the second branch of the processes to run, and which network to run them on.

The **ForkProcess** can be used more generically to split processing at any level of the hierarchy. Like the sync epoch process, it adds pointers to the second fork of processing, including a `second_environment`.

IMPORTANT NOTE: You need to specifically add small script processes to the processes below a fork process that perform initialization and other operations on the second network – the standard processes will only perform these operations on the first network!

One use of multiple processing streams is to combine two different algorithms to form a hybrid network. This can be done by synchronously running both networks, each with their own algorithm-specific training process, and linking them together with a **BridgeProcess**, which is described in the next section.

The **MultiEnvProcess** can be used to iterate over multiple environments within one processing stream. This can be useful for testing a number of different possible environments.

12.4.4 Linking Networks Together with a Bridge

Most **TrialProcess** objects are implemented to work with a particular set of network objects that are part of a given algorithm. Thus, the **BpTrial** process expects to operate on **BpUnits** and **BpCon**s, etc., and crashes otherwise. This makes processing faster than if it had to check the type of every object operated on every time it did a computation.

This situation makes it difficult to implement hybrid networks which combine components that operate under two different algorithms. For example, a self-organizing network can be used to pre-process inputs to a backprop network.

The way to do solve this problem in PDP++ is to use a **BridgeProcess**, in conjunction with a **SyncEpochProc** as described in the previous section. Thus, there are two trial processes that each operate synchronously within an epoch on the same events. The bridge process copies activations or any other unit state variable from one network to the other, allowing them to act as if they were a single composite network.

An example of a bridge process can be found in the '`demo/bridge`' directory. This example just connects two backprop networks, but the principles are the same when you use two different kinds of algorithms.

The parameters of the bridge process are as follows:

`Network* second_network`
: This is the other network that is being bridged (the first network is the one pointed to by the process `network` pointer).

`BridgeDirection direction`
: This is the direction to copy–network one is the `network` pointer and network two is the `second_network` pointer. Note that the `network` pointer is set by the process hierarchy that this process is in, which means that it can't be set arbitrarily. This is why one might need to switch the direction with this field.

`String src_layer_nm`
: This is the name of the layer in the source network. Only entire layers can be copied with this process.

`String trg_layer_nm`
: This is the name of the layer in the target network.

`String src_variable`
: This is the variable name (e.g. "act") to copy from the unit.

`String trg_variable`
: This is the variable to copy the value into. Typically, this is "ext" so that the input appears like external input to the unit, which will then be treated appropriately by the processing algorithm.

`Unit::ExtType trg_ext_flag`
: This sets the unit flag on the target units to indicate that external input was received, if desired. Note that the flag on the layer is *not* set, which allows this external input to avoid being erased by the `InitExterns` call which usually precedes application of environmental patterns.

12.4.5 Miscellaneous other Process Types

The **SaveNetsProc** and **SaveWtsProc** will simply save the current network or network weights to a file which has a file name composed of the name of the network (from its `name` field) plus the current values of the epoch and, if present, the batch counters. This can be placed in the `loop_procs` or `final_procs` of the **TrainProcess** or the **BatchProcess** to save networks during training or after training, respectively. Set the mod parameters to get it to save at different intervals (e.g., `m = 25` saves every 25 epochs if in the train loop procs).

The **LoadWtsProc** will load in a set of weights from a specified file, which is useful for example for performing repeated lesion tests on a given trained network – build the network, load the weights, lesion it, test it, repeat!

The **InitWtsProc** will initialize the weights of a network. The **TrainProcess** does this automatically when it is initialized, but there are other possible configurations of processes where you might need to initialize the weights at a different point.

The **DispDataEnvProc** automatically performs an analysis of a data environment (a data environment contains data recorded from the network for the purposes of analysis, by the `CopyToEnvStat` stat (see Section 12.6.8 [proc-stats-misc], page 207). See Section 5.1 [how-proc], page 43 for an overview of how this analysis process works, and Section 11.9 [env-analyze], page 179 for the types of analyses that can be performed. The data environment to analyze is specified in `data_env` (the regular `environment` pointer points to the environment actually used by the entire process hierarchy, and is not used). The type of analysis is specified in the `disp_type` field, and the results of the analysis are displayed in the log pointed to by the `disp_log` field (if this is NULL, or the wrong type, a new one is made). The remaining parameters are used for different analysis routines, as described in greater detail in Section 11.9 [env-analyze], page 179.

The **DispNetWeightsProc** automatically displays the weights between two layers of the `network` in a grid log. Useful for monitoring the development of the entire set of weights as the network learns (the netview can only display the weights to one unit, whereas this displays the weights to all units in the layer). This is just a call to the `GridViewWeights` function on the Network (see Section 10.1 [net-net], page 131).

There are a set of processes that reset other processes, stats, or logs – these are usually placed at a higher level of the processing hierarchy in `init_procs`, to initialize things. They are: **UnitActRFStatResetProc**, **TimeCounterStatResetProc**, and **ClearLogProc** – their functions should be fairly obvious.

12.5 The Statistic Process

The Statistic Process is an extension of the basic Process object which is used for computing values that are then made available for recording and displaying in logs. The basic **Stat** object defines an interface for computing and reporting data. This interface is used by the schedule processes, who supervise the running of stats and the reporting of their data to the logs.

Each statistic object can operate in one of two capacities. The first is as the original *computer* (or collector) of some kind of data. For example, a squared-error statistic (**SE_Stat**) knows how to go through a network and compute the squared difference between

target values and actual activations. Typically, this would be performed after every event is presented to the network, since that is when the relevant information is available in the state variables of the network.

The second capacity of a statistic is as an *aggregator* of data computed by another statistic. This is needed in order to be able to compute the sum of the squared-errors over all of the trials in an epoch, for example. When operating in aggregation mode, statistics work from data in the statistic they are aggregating from, instead of going out and collecting data from the network itself.

Typically, the statistic and its aggregators are all of the same type (e.g., they are all **SE_Stat**s), and the aggregated values appear in the same member variable that the originally computed value appears in. Thus, this is where to look to set a stopping criterion for an aggregated stat value, for example.

Each statistic knows how to create a series of aggregators all the way up the processing hierarchy. This is done with the `CreateAggregates` function on the stat, which is available as an option when a statistic is created. Thus, one always creates a statistic at the processing level where it will do the original computation. If aggregates of this value are needed at higher levels, then make sure the `CreateAggregates` field is checked when the stat is created, or call it yourself later (e.g., from the *Actions* menu of a stat edit dialog). You can also `UpdateAllAggregators`, if you want to make sure their names reflect any changes (i.e., in `layer` or network aggregation operator), and `FindAggregator` to find the immediate aggregator of the current stat.

It is recommend that you use the *NewStat* menu from the *.processes* menu of the project to create a new statistic, or use the Project Viewer (see Section 9.2 [proj-viewer], page 120). This will bring up a dialog with the default options of where to create the stat (i.e., at what processing level) that the stat itself suggested (each stat knows where it should do its original computation).

There are several different kinds of aggregation operators that can be used to aggregate information over processing levels, including summing, averaging, etc. The operator is selected as part of the `time_agg` member of the statistic. See below for descriptions of the different operators.

Note that all aggregation statistics reside in the `loop_stats` group of the schedule processes, since they need to be run after every loop of the lower level statistic to collect its values and aggregate them over time.

In addition to aggregating information over levels of processing, statistics are often aggregating information over objects in the network. Thus, for example, the **SE_Stat** typically computes the sum of all the squared error terms over the output units in the network. The particular form of aggregation that a stat performs over network objects is controlled by the `net_agg` member. Thus, it is possible to have the **SE_Stat** compute the average error over output units instead of the sum by changing this variable.

Finally, the name of a statistic as recorded in the log and as it appears in the `name` field is automatically set to reflect the kinds of aggregation being performed. The first three-letter prefix (if there are two) reflects the `time_agg` operator. The second three-letter prefix (or the only one) reflects the `net_agg` operator. Further the layer name if the `layer` pointer is non-NULL is indicated in the name. The stat `name` field is not automatically set if it does

not contain the type name of the stat, so if you want to give a stat a custom name, don't include the type name in this.

12.5.1 Aggregation Operators and other Parameters

For both the `time_agg` and the `net_agg` members of a stat, the following aggregation operators are defined:

`LAST` This simply copies the last value seen. This is useful for aggregators that appear in the train and batch levels of processing, which typically just reflect the results of the most recent epoch of processing. In the network-level aggregation, this would give the value of the last unit, which is not terribly useful.

`SUM` This sums over all values.

`PROD` This gives a product over all values. When this operator is used, the result variable will be initialized to 1.

`MIN` This gives the minimum value of all seen. Note that when this is the aggregation operator, the result variable will be initialized to a very large number.

`MAX` This gives the maximum value of all seen. Note that when this is the aggregation operator, the result variable will be initialized to a very small number.

`AVG` This gives the average of all values seen. Since aggregation happens on-line, we use the on-line version of averaging, so the result is always the average of what has been seen so far.

`COPY` This results in the collection of individual values, which are kept in the `copy_vals` group of the stat. Thus, it does not form a single summary number of all the values, instead it simply copies them verbatim. This is useful for **MonitorStat** objects, which copy state variables from the network. It can be used to view per-event values at the end of the epoch by doing a `time_agg` with the `COPY` operator.

`COUNT` This counts up the the number of times the values meet the comparison expression given in the `count` field of the agg member. The count expression has relational operators and a comparison value, so one could for example count the number of times an error value was below some threshold.

In addition to the aggregation operator, the `time_agg` member has a pointer to the stat that this stat is aggregating `from`. If this is `NULL`, then the stat is computing original information instead of aggregating.

It is possible to control when the stat is computed, and if the data is logged, independently. The `mod` member of a stat determines when and if it is computed (and when its criterion is checked, when it is logged, etc). For stats located in the `loop_stats` group, this mod operator works on the process whose loop_stats the stat is in. For stats located in the `final_stats` group, the mod operator works on the next higher up process in the hierarchy (i.e., a stat in the final_stats of a TrialProcess would use the trial counter from the parent EpochProcess). The `log_stat` flag provides a way of turning on or off the logging of a statistic. If the flag is not checked, a stat is not logged, but it is run and its criterion

is checked (as per the mod settings). Thus, one can keep lower-level stats which might be just collecting data for aggregation from generating too much log data.

Finally, the computation of the stat over the objects in the network can be restricted to a given layer by setting the `layer` pointer. The layer name will also appear in the stat log output and in the name field of the stat itself.

12.5.2 Using Statistics to Control Process Execution

Often, one wants to stop training a network once it has reached some criterion of performance. This can be done by setting the criterion values associated with a statistic value. All statistic values are represented by a **StatVal** object, which has fields for representing the stopping criterion. The criterion is represented with a relational operator (less than, equal to, etc.) and a comparison value. The fields are as follows:

`float value`
: This holds the current computed or aggregated value of the statistic.

`bool flag`
: This flag indicates if a stopping criterion is active for this statistic. If it is not checked, the remaining fields are ignored.

`Relation rel`
: This is the relational operator to compare `value` and `val`.

`float val`
: This is the comparison value to compare `value` with.

`int cnt`
: This indicates how many times the relation must be met in order to pass criterion. This can be useful to make sure a network has reliable performance under criterion by requiring it to pass muster 2 or 3 times in a row, for example.

12.5.3 Implementational Details about Stats

If you will be writing your own statistic process, this provides some information that might be useful.

The stat object provides a scaffolding for looping through the various objects in a network. Thus, if you want to do something at the unit level, you can simply write a `Unit_Stat` function, and the stat will automatically iterate over layers and units to call the unit stat function on every unit. This makes it relatively easy to write a new statistic.

See the header file '`src/pdp/stats.h`' for more information about how a stat object works. In particular, notice that there are recommended ways of speeding up otherwise generic functions that rely on type-scanned information.

12.6 Different Types of Statistics

There are a number of built-in statistic types that come with the software. They are as follows:

12.6.1 Summed-Error Statistics

The **SE_Stat** object is a stat that iterates over the units that have target values in a network, and computes the difference between the activation and the target value. This is

useful for monitoring learning performance over training. The current value of this statistic can be found in the `se` member. Also, there is a `tolerance` parameter which causes absolute differences of less than this amount to result in zero error. Thus, if one only was interested in whether the network was on the right side of .5, you would set the tolerance to .5 (assuming a 0 to 1 activation range).

There is also a **CE_Stat** and a **RBpSE_Stat** defined in the Bp version of the executable. These compute the cross-entropy error statistic and a version of squared-error that takes into account the `dt` parameter of the recurrent backprop algorithm.

The **MaxActTrgStat** computes an error statistic based only on the most active unit in the target layer(s). If this most active unit (max act) has a target value of 1, then there is no error, otherwise there is an error. This statistic is useful when there are multiple possible correct answers, and the network is expected to just choose one of them. Thus, if its maximum act is a target, it is correct, and otherwise it is not.

12.6.2 Monitoring Network State Variables

In order to be able to view network stat variables (e.g., unit activations) in one of the log displays (or record them to disk files), these state variables need to be monitored with a **MonitorStat**.

The **NetView** provides a convenient interface for creating monitor stats and selecting which objects to monitor and what values to monitor from them (see Section 10.6.1 [net-view-actions], page 154), as does the **Wizard** object (see Section 5.5 [how-wizard], page 51).

There are basically two parameters of relevance in the monitor stat. One is the `objects` that are being monitored. This is a group which has links to the objects (see Section 8.2 [obj-group], page 109 for information on links). The other is the `variable` to record from these objects. Note that if the variable is one found on units, but the object(s) are layers, then the stat will automatically get the unit variable from all of the units in the layer. Similarly, if the variable is one on connections, but the object(s) are projections, all of the connections in the projection will be used.

Typically, the `net_agg` operator `COPY` is used. This results in a separate column of data for each object being monitored. This data is stored in the `mon_vals` group on the monitor stat. When these values are graphed or displayed in the grid log (see Section 13.3.4 [log-views-graph], page 216, Section 13.3.5 [log-views-grid], page 221), they appear as one big group that shares the same axis on the graph and is part of the same sub-grid on the grid.

However, one can compute any kind of network aggregation from the monitored statistics, including MAX, AVG, etc. These aggregations produce a single value in the `mon_vals` group.

It is also possible to perform three steps of pre-processing on the monitored values before they are recorded or aggregated into the monitor stat. This pre-processing is controlled by the `pre_proc_1,2,3` members, which specify an operation and, optionally, arguements to that operation in the `arg` member. Note that the thresholding function `THRESH` compares the value to the `arg`, and gives a result of `hi` if it is greater-than-or-equal, and `lo` if it is less-than the arg.

12.6.3 Finding The Event Closest to the Current Output Pattern

In order to understand what kinds of errors a network is making, or in the case where a network can produce multiple outputs for a given input, it is useful to be able to compare the actual output the network came up with against all of the possible training events to find the one that matches the closest. The **ClosestEventStat** does exactly that.

The closest event stat reports both the distance in the `dist` field, and the name of the event which was closest to the current output pattern in the `ev_nm` field. If the `ev_nm` matches that of the currently presented event (`cur_event` of the TrialProcess), then `sm_nm` is 1, else it is 0. The average of this value gives a "percent correct" measure for forced-choice performance among the different items in the environment. The distance can be computed in several different ways, as described below:

`CompareType cmp_type`
: This is the type of distance function to use in making the comparison:

  `SUM_SQUARES`
  : sum of squares distance: sum[(x-y)^2]

  `EUCLIDIAN`
  : euclidean distance: sqrt(sum[(x-y)^2])

  `HAMMING_DIST`
  : hamming distance: sum[abs(x-y)]

  `COVAR` covariance: sum[(x-<x>)(y-<y>)]

  `CORREL` correlation: sum[(x-<x>)(y-<y>)] / sqrt(sum[x^2 y^2])

  `INNER_PROD`
  : inner product: sum[x y]

  `CROSS_ENTROPY`
  : cross entropy: sum[x ln(x/y) + (1-x)ln((1-x)/(1-y))]

`float dist_tol`
: This is a tolerance value for distance comparisons, where absolute differences below this amount result in a 0 distance component.

`bool norm` If this flag is checked, and one of the distance comparisons is being performed, the values participating in the distance computation will be normalized to a zero-one range prior to computation. If the `INNER_PROD` is being taken, this will result in a normalized inner-product measure (dividing by the magnitudes of the individual weight vectors).

12.6.4 Comparing or Computing on Stat Values

The **CompareStat** provides a general way of comparing the results of different statistics with each other. Thus, one can actually use statistics to analyze one's data on-line, instead of dumping it all to a file and analyzing it after the fact.

Similarly, the **ComputeStat** provides a general way of performing simple math computations on the results of other stat computations. It can be used on one stat (e.g., for

thresholding, absolute-value or other single-argument operations), or on two stats (e.g., for multiplying, subtracting, etc between two stats).

The compare stat contains pointers to two other stats, `stat_1` and `stat_2`, which provide the data to compare. The data consists of any stat val data that can be found on these stats. Ideally, they both have the same number of data values, typically in their `copy_vals` group (e.g., from **MonitorStat**s that are `COPY`ing activations from two sets of units that are being compared).

The types of comparisons are simply different distance functions that measure the distances between the two stat's data:

`CompareType cmp_type`
: This is the type of distance function to use in making the comparison:

  `SUM_SQUARES`
  : sum of squares distance: sum[(x-y)^2]

  `EUCLIDIAN`
  : euclidean distance: sqrt(sum[(x-y)^2])

  `HAMMING_DIST`
  : hamming distance: sum[abs(x-y)]

  `COVAR` covariance: sum[(x-<x>)(y-<y>)]

  `CORREL` correlation: sum[(x-<x>)(y-<y>)] / sqrt(sum[x^2 y^2])

  `INNER_PROD`
  : inner product: sum[x y]

  `CROSS_ENTROPY`
  : cross entropy: sum[x ln(x/y) + (1-x)ln((1-x)/(1-y))]

`float dist_tol`
: This is a tolerance value for distance comparisons, where absolute differences below this amount result in a 0 distance component.

`bool norm` If this flag is checked, and one of the distance comparisons is being performed, the values participating in the distance computation will be normalized to a zero-one range prior to computation. If the `INNER_PROD` is being taken, this will result in a normalized inner-product measure (dividing by the magnitudes of the individual weight vectors).

`SimpleMathSpec pre_proc_1,2,3`
: These allow for three steps of pre-processing on the values before they are compared. These members specify an operation and, optionally, arguements to that operation in the `arg` member. Note that the thresholding function `THRESH` compares the value to the `arg`, and gives a result of `hi` if it is greater-than-or-equal, and `lo` if it is less-than the arg.

The **ComputeStat** is like the compare stat, except that instead of computing the distance, the `compute_1,2,3` math operators are applied on the stats, and the result is aggregated according to `net_agg`. Pre-processing of each stat independently is supported as with the compare stat. To compute something between two stats (e.g., subtract values), then you just

set the compute_1 operator `opr` to `SUB`, and stat_2 values are subtracted from stat_1 values, with the result going into stat_1. Subsequent compute_ operators can then manipulate this result (e.g, doing the `SQUARE`). Note that they don't all have to involve both stats, and you can only use one stat (in which case compute_x just works like pre_proc_x).

12.6.5 Activity-based Receptive Fields

The **UnitActRFStat** computes an effective receptive field for units based on their activation values when inputs are presented. The idea is to present a wide range of inputs to the network while performing a weighted average over these input patterns as a function of the unit's activation for each. When you do this averaging, all the things that the unit does not care about wash out, leaving an image of those things that reliably activate the unit. Assuming that the inputs span a large enough space and provide for sufficient averaging, the resulting receptive field can be much more informative than just looking at the weights, since it takes into account any network dynamics, etc., and can be computed on units any number of layers removed from the inputs. Finally, the "input" that you average over need not literally be the input layer to the network – it could be the output, or an intermediate hidden layer (or all of these at once).

This stat requires a stable database for accumulating the averaged receptive fields – an **Environment** is used for this purpose. Note that you have to do an `InitRFVals` in order to initialize this environment before you start collecting the stats. The statistics collect until `InitRFVals` is run again.

`InitRFVals` can be performed automatically via a **UnitActRFStatResetProc**, which can be put in the `init_procs` of a higher processing level in the hierarchy.

This stat has the following parameters:

`layer` Set this to point to the units that you want to record the receptive fields for.

`rf_layers`
This contains the layer(s) that you want to perform the averaging over (the input in the above description).

`data_env` This points to the **Environment** that contains the resulting receptive fields, with one **Event** per unit.

12.6.6 Reaction-time Based on Crossing an Activation Threshold

The **ActThreshRTStat** records a reaction time (RT) from the network based on when activation levels in given layer (typically the output layer) exceed threshold. Experience in a variety of cases has shown that human reaction times can be best modeled by recording the number of processing cycles it takes for a response/output layer unit to exceed some kind of activity threshold. In contrast, recording RT based on the change in activation over time going below a threshold (in '`cs++`' this is **CsMaxDa**; in '`leabra++`' it is **LeabraMaxDa**) is typically not such a good measure of human reaction time.

This stat should typically be created in the `loop_stats` of the **SettleProcess**. To configure this stat, just set the `layer` pointer to point to the response/output layer in the network, and set the `act_thresh` to the activation threshold. Two values are recorded in this stat:

`max_act` records the maximum activation in the response layer, and setting a stopcrit on this will actually result in stopping settling upon reaching threshold (note that the val of this stopcrit is automatically set to be the same as the `act_thresh` value, but the stopcrit flag is not active by default, so that it will not stop processing).

`rt_cycles` records the number of settling cycles at the point when the maximum activation in the layer exceeded the `max_act` threshold (regardless of whether the settle process actually stopped at this point).

12.6.7 Statistics that Provide Counters (Time, Epoch, etc)

The following statistics all provide counter data, which are useful for providing X-axes for graphing and other data analysis.

The **EpochCounterStat** records the current epoch number from the network. This is useful for testing process hierarchies which start at the epoch level and thus do not have an epoch counter from the training process.

The **ProcCounterStat** grabs a whole set of counters off of another process. It is used for the same reason an epoch counter stat is used, except it also gives one access to batch counters and any other counters that might be present on the training process hierarchy.

The **TimeCounterStat** simply increments its counter every time it is run, providing an ever-incrementing time count that spans across multiple loops through a given level of the process hierarchy. Use the **TimeCounterResetProc** to automatically reset this time counter at some higher level of the process hierarchy (e.g., at the start of training, in the `TrainProcess` `init_procs`).

12.6.8 Miscellaneous Other Stat Types

The **CyclesToSettle** statistics simply records the number of cycles it took to settle. It should be placed in the `loop_stats` of the trial process, where it will be able to grab the final counter value from the settle process. This is useful to record how fast a network is settling, which is often used as a proxy for reaction times, etc. See also the **ActThreshRTStat** Section 12.6.6 [proc-stats-rt], page 206.

The **ScriptStat** is a statistic that is meant to be used with a CSS script. It provides a generic group of `vals` into which the results of the statistic can be put, and an array of `s_args` for passing arguments to the script to control its behavior. A very simple example script code for computing the difference between two unit activations is as follows:

```
// this is a sample script stat script

void DoStat() {
  if(vals.size != 1) {  // first, create vals to hold results
    // anything in vals is automatically logged, etc.
    vals.EnforceSize(1);  // do whatever is necessary to get 1 val
    vals[0].name = "act_diff";  // this is the header for the stat val
  }
  // get the first unit (note that we can access 'network'
  // and other member variables from the ScriptStat the script is in)
  Unit* un1 = network.layers[1].units[0];
```

```
  // and then the second unit
  Unit* un2 = network.layers[1].units[1];

  // compute value to put in statistic
  float diff = un1->act - un2->act;

  // then store result in the val.  note you can have as many
  // vals as you want and compute as many things as you want!
  vals[0].val = diff;
}

// you *must* call the function so that when the script is run
// the above function is actually run!
DoStat();
```

The **CopyToEnvStat** takes data from another statistic and copies it into a data environment (just a basic environment that holds data for later analysis). This is a key piece of a chain of steps involved in analyzing network representations, etc. See Section 5.1 [how-proc], page 43 for an overview of how this analysis process works, and Section 11.9 [env-analyze], page 179 for the types of analyses that can be performed. The `DispDataEnvProc` can automate the performance and display of these analysis routines (see Section 12.4.5 [proc-special-misc], page 199).

The key parameters on this stat are the `stat`, which points to the statistic to get data from, the `data_env`, which points to the environment to save the data in, and the `accum_scope`, which determines how much data to accumulate in the data env (e.g., EPOCH accumulates over an entire epoch, etc).

The **ProjectionStat** projects data from another statistic (e.g., a `MonitorStat`) onto a stored vector (`prjn_vector`), and records the resulting scalar value in `prjn`. A projection is just computing the distance between two vectors, and the various parameters on this stat determine the type of distance computation to perform. This is useful for analyzing network representations as they are generated based on for example a principal components analysis performed on another batch of previously-generated network activations. To facilitate this, the buttons `VecFmPCA` and `VecFmEvent` provide a way to load the prjn vector from either a PCA (principal components analysis) performed on a data environment (see Section 11.9 [env-analyze], page 179) or just from a stored event pattern (e.g., the environment can be used to draw a pattern to compare).

12.7 Processes and CSS Scripts

Any kind of process can be configured to use a CSS script instead of its original hard-coded functions. One simply sets the process type to `SCRIPT` and opens a script file in the `script_file` member of the process.

When the process is run, it checks to see if it should run the script instead. Note that if you are replacing a schedule process with a script, you have to replace the entire C_Code function. This code can be used verbatim in CSS, and an example is given in '`css/include/script_proc.css`'.

Note that the script is given transparent access to all of the members and member functions defined on the script object it is attached to. This allows one to mix existing hard-coded functions with script versions by simply calling the existing ones in some places, and calling new script-defined ones in other places.

Where possible, it is generally preferable to use a **ScriptStat** or **ScriptProcess** instead of replacing an entire existing process with a script. This will tend to be simpler and a more modular solution.

13 Logs and Graphs

Logs provide a convenient method of recording information from the processes. This information can be as simple as the epoch number of training or as complicated as a multi-variable statistic. In PDP++ logs are stored on the Project, and pointed to by a process. Like the network, logs may have many views. Logs are color-coded brown in the default color scheme (get it?).

Processes interface with a log by sending it both header and data information. The header information provides the labels for the columns of data. Each line of data provides a row of information for each of the fields in header line.

Logs can record data to log files, which contain columns and rows of data in text format. Each line of the log file contains an identifier that indicates the name of the process which generated the data (or just _H:) if there is only one updating process. At the start of a log file, a row of header information is recorded, which identifies the data in the columns. Both of these identifiers can be removed with the shell script '`bin/getlogdata`', which strips them from the file, and writes a new file with a '`.logd`' extension.

Note that the log data file is distinct from the object save file, which will save the parameters associated with the log object, but not the data in the log itself. Thus, projects do not save any current log data. It has to be specifically saved in a log file. Functions to save and load log files are found in the `Logfile` menu.

Also, the log object maintains an internal buffer of log data (the size of which can be set by editing the log, see next section). This buffer can be directly accessed through the script language to perform various data analysis operations.

13.1 PDPLog Variables

`File log_file`
: File to use for saving the log. This field is set by using the File Requester (see Section 6.8 [gui-file-requester], page 69), or by using the log function `SetSaveFile()`. This field can be set to `NULL` in which case there is no file i/o.

`int log_lines`
: The number of lines recorded so far in the log. This number is incremented each time the log receives a new line of input.

`DataTable data`
: Holds and provides organization for the information sent to the log.

`int data_bufsz`
: The size in lines of the log buffer. If the log overflows the buffer size, information is dropped from the beginning of the Log and the buffer is "scrolled" by `data_shift` to make room for the new data.

`float data_shift`
: The percentage of the buffer to shift upon overflow.

`MinMax data_range`
: A structure containing the minimum and maximum line values of the log s buffer's view of the data in the `log_file`.

`Process_Group log_proc`
: A link group of processes which log information to this log. Usually, there is just one process that sends data to the log. The menu functions AddUpdater/RemoveUpdater manipulate this information.

`String_Array display_labels`
: Provides replacement labels for data columns.

13.2 PDPLog Functions

On all of the following functions, the optional no_dlg arg will prevent the popup file requester dialog if true - this dialog comes up even when a valid file name is specified.

`Actions/GetHeaders()`
: Tells all the processes that update this log to send their current headers to this log. Use this if you have changed the statistics being sent to this log. Note that it causes all existing data to be removed. The log will automatically adjust to the addition and removal of statistics, so this isn't necessary in those cases. However, it won't notice if a name of a statistic has changed.

`LogFile/SetSaveFile(char* file_nm, bool no_dlg = false)`
: Pulls up a file chooser dialog that allows one to select the file that the log will save data into, or sets the log file to the argument if called from the script language. This does not save any existing data in the file, it just opens the file so that any new data will be recorded. Use `BufferToFile` after setting the save file (or by itself without a save file set) to actually save currently buffered data to the file. Note that this will overwrite any existing file of the same name.

`LogFile/SetAppendFile(char* file_nm, bool no_dlg = false)`
: This is just like `SetSaveFile`, but it appends to an existing file instead of writing over it.

`LogFile/LoadFile(char* file_nm, bool no_dlg = false)`
: This will load data from a previously-saved log file into this log. Note that it is possible to load from a log file created by any process, since it will read the header information from the log file itself. Thus, one can open a new project, create a **GraphLog** object, and do a Load file on a log from a project that was run in the background, and get a graph of what happened.

`LogFile/CloseFile()`
: Closes any open files. Note that `LoadFile` does not close the log file, because if the file is longer than the current buffer, it needs to be read from as the user scrolls through the file.

`LogFile/BufferToFile(char* file_nm, bool no_dlg = false)`
: Sends the entire contents of the currently-buffered log data to a log file. If there is an open log file and no file name is specified, it sends to this currently open

file. If there is no open file, then it will do a SetSaveFile(file_nm, no_dlg) and then dump the buffer to the file, closing the file afterwards.

13.3 Log Views

The PDPLog structure has many options for viewing its data. Each of these options is represented by a corresponding LogView object. Like the network object, PDPLogs can have multiple views.

Each of the views has its own methods of interacting with the data, however all the views have some properties in common. All the views are window's into the data stored in the log's datatable `data`. While some of the views may show only a portion of the datatable at a time, the maximum amount of data they can show is limited by the size of the datatable itself, `data_bufsz`. The `log_file` however records all the information sent to log sequentially. Each of the LogViews provides four buttons which shift the the view's window on the datatable. If the view's window on the datatable moves outside the range of the datatable, the datatable is scrolled by moving log's `data_range` throughout the `log_file`.

13.3.1 The LogView Class

The LogView class provides the general variables, functions, and interface components for all the LogView subclasses. Visually all LogView objects have a common interface at the top of the LogView window. In the top middle of each LogView widow is a set of four buttons used for scrolling the visible region of the LogView throughout the data and data logfile:

`|<` Full Rewind
: Load the first `data_bufsz` lines of data from the file.

`<` Rewind
: Move back `data_shift` lines in the datatable.

`>` Forward
: Move forward `data_shift` lines in the datatable.

`>|` Full Forward
: Load the last `data_bufsz` lines of data from the file.

To the left of the scroll buttons is a toggle switch which sets the LogView's updating on or off. To the right of the scroll buttons are the Update, Init, and Clear buttons. The Update button is the most benign. Its action merely refreshes the display. The Init button is does a bit more by rescaling, recentering and recreating the graphical representation of the LogView based on the settings of the ViewSpecs and the LogView object. The Clear button destructively removes all the data from the log and re-initializes the display. It is useful if you are restarting your Training for example and want to clear out the statistics gathered for the last Training run. The Clear button does not affect the log files.

All LogViews have the following functions in their **View:Object** menu:

`AddUpdater(SchedProcess* proc)`
: Adds the given process as an updater of this log (i.e., it sends data to this log).

`RemoveUpdater(SchedProcess* proc)`
: Removes the given process as a sender of data to this log. This function can also be used check what process(es) are updating this log.

All LogViews have the following functions in their **Actions** menu:

`StartAnimCapture(img_fmt, img_ctr, sub_dir)`
: Starts automatic capturing of window updates to files of given format saved in given sub directory, with increasing counter extensions starting with img_ctr. These images can then be processed to create an animation of what was displayed in the view.

`StopAnimCapture()`
: Stops the automatic capturing of window updates to files.

`EditViewSpec(ViewSpec* column)`
: Brings up an edit dialog for parameters that control the display of given column or group of data.

`SetVisibility(ViewSpec* column, bool vis)`
: Determines whether a given column of data is visible (displayed) in the log view or not.

`SetLogging(ViewSpec* column, bool log_data, bool also_chg_vis)`
: Determines whether a given column of data is logged to the log file, and, if also_chg_vis is set, it also similarly affects whether the column is visible (displayed) in the log view or not. The logging flag (`save_to_file`) is actually stored in the `data` structure on the log object itself, and is thus shared by all views of this log.

`UpdateDispLabels()`
: Copies display_labels from the log object to the labels used to display data in the log view. These display_labels are not used by default.

Furthermore, LogViews have an **Analyze** menu that allows one to analyze columns of data in the log. This actually occurs by copying the data to an environment object, and then running analysis routines on that environment. See the environment documentation (see Section 11.9 [env-analyze], page 179) to see what other functions are available, and for greater explanation of these analysis functions.

`CopyToEnv(TAPtr data, TAPtr labels, Environment* env)`
: Outputs data column (must be group) with labels to environment env (WARNING: reformats env to fit data!) Further analysis and transformations can be performed within the environment.

`DistMatrixGrid(TAPtr data, TAPtr labels, GridLog* disp_log, DistMetric metric, bool norm, float tol)`
: Computes and displays in a grid log (NULL = make a new one) the distance matrix for data (must be group of data) with labels (typically event names).

`ClusterPlot(TAPtr data, TAPtr labels, GraphLog* disp_log, bool graphic, DistMetric metric, bool norm, float tol);`
: Produces a cluster plot (in graph log, NULL = make a new one) of the given data (with labels).

`CorrelMatrixGrid(TAPtr data, TAPtr labels, GraphLog* disp_log);`
: Produces a correlation matrix of the given data – how each column of values correlates with the other columns across the rows of different values, and plots this in a grid log.

`PCAEigenGrid(TAPtr data, TAPtr labels, GraphLog* disp_log);`
: Computes principal components (eigenvectors) of the correlation matrix of column values across rows of data, and plots these eigenvectors in a grid log.

`PCAPrjnPlot(TAPtr data, TAPtr labels, GraphLog* disp_log, int x_axis_component, y_axis_component);`
: Computes principal components (eigenvectors) of the correlation matrix of column values across rows of data, and plots the projection of each row of data against two of these principal components. This produces a simplified two-dimensional representation of the similarity structure of the data.

`MDSPrjnPlot(TAPtr data, TAPtr labels, GraphLog* disp_log, int x_axis_component, y_axis_component);`
: Computes multidimensional scaling (MDS) on the distance matrix of rows of data, and then plots a projection of each row onto two components – this also provides simplified two-dimensional representation of the similarity structure of the data.

And as mentioned above, all LogViews have the following Variables:

`bool display_toggle`
: Determines if screen updates from the Update Processes will affect the display or not. It is sometimes useful to turn off the complex display updating of the LogView to speed up the Processes. This variable does not affect the recording of the Log information. Thus if the `display_toggle` is toggled off while a Process is running, and then toggled back on, the LogView display will reflect all of the data received when the LogView display was turned off.

`int view_bufsz`
: The maximum number of lines of Log data visible in the Display.

`float view_shift`
: The percentage of `view_bufsz` to shift when the data exceeds the `view_range`

`MinMax view_range`
: The range of currently visible data lines.

`DT_ViewSpec viewspec`
: This DataTable View specification is the place holder for all the ViewSpecs for the individual columns of data in the log – edit this to get at all of the parameters that control the view display.

13.3.2 The Text LogView

The Text LogView provides a read-only scrollable window of a textual representation of the log's data. Across the top of the TextLogView are the common file I/O buttons and the display toggle. Beneath those buttons is a horizontally scrollable region of data. The

top line of data will always be the most recent header information the log has received. It will be printed in bold face type, with each field separated by the expected size of the data. Beneath the header info and its separator bar is the actual data. The fields in the data line correspond to the the fields of header line above.

Each column of data has an associated TextViewSpec which has but one field:

`width` The width (in 8 character blocks) allocated to the column of data

Resizing the TextLogView window changes the number of visible data lines, although you typically need to press *Init* to get everything synced up.

13.3.3 The Net LogView

The NetLogView is similar to the TextLogView in that it displays the log's data in a textual representation. Its main difference however is that instead of displaying the values in its own window, it displays its data as labels in an associated network's views. Thus the NetLogView has one important field, `network`, which is the network to display into. By default, the `network` is set to the one used by the process that updates the log. The data is displayed in each of the network's views by adding labels to the NetViews and changing the text in those labels. The labels can be repositioned (as a group or individually) and edited in the NetView to change their fonts. The labels are arranged in columns at the upper left of the netview.

The following functions on the **Actions** menu are available to control the display:

`SetNetwork(Network* net)`
Sets the network to display the log data into. Goes into the default display in this network.

`ArrangeLabels(int cols, int rows, int width, float left, float top)`
Controls positioning by specifying the number of columns and rows, their width, and where in the display they should appear (left = 0 = left side, 1 = right; top = 0 = bottom, 1 = top).

13.3.4 The Graph LogView

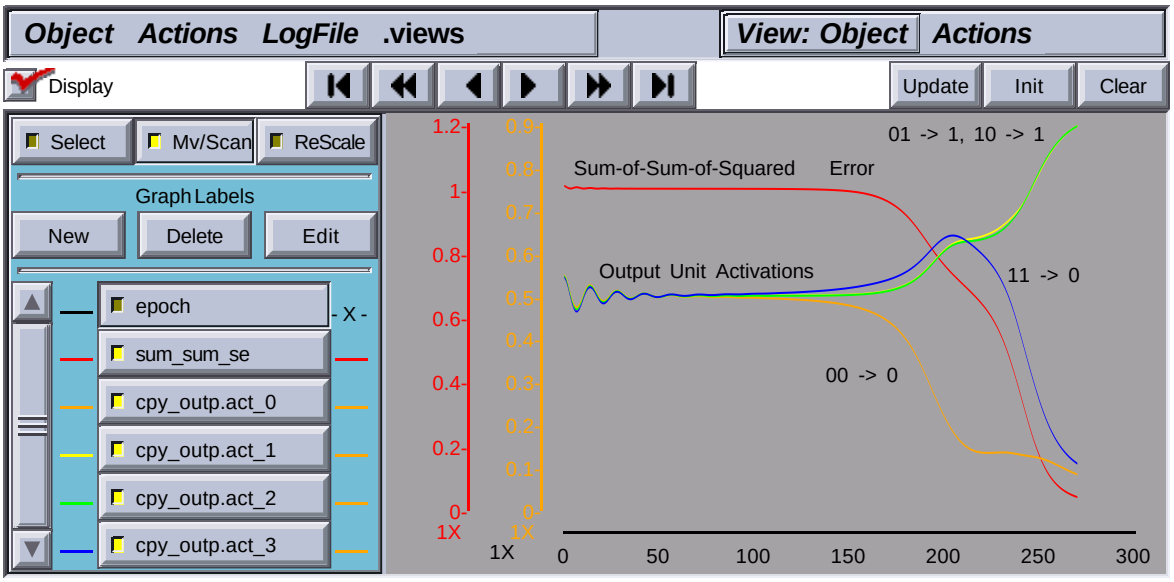


Figure: 9 The Graph Log Viewer

The GraphLogView provides a realtime 2D line graph representation of the log's data. The fields of the header of the log are each graphable as a separate line in the Graph. The color and style of the lines are fully customizable by the user as are the range and scaling factors of the graph itself.

Resizing the GraphLogView window rescales the graph to fit the size of the window.

13.3.4.1 The Field Toggle Buttons

Each of these fields of data in the GraphLogView's PDPLog are listed in a vertical scrollbox of toggle buttons in the lower left of the GraphLogView. Initially all the lines are toggled on, except those that represent counter data and String data. By pressing the toggle button with the left mouse button, the display of the line corresponding to that field is turned on or off. **Pressing the middle mouse button selects that field to represent the X axis – a second press on a group of data selects Row-wise plotting of data**. Pressing the right mouse button on the field toggle button brings up an edit dialog for the field, where you can specify all the detailed parameters controlling its display.

Initially each of the lines is assigned a default line style and color. The color of the line is displayed as a small line just to the left of the toggle button (for String fields, an "-S-" is displayed). To the right of the toggle button another small line will indicate the color of the line's y-axis, or a "-X-" if the toggle button's field represents the X-axis.

The GraphViewSpec provides the following customizable variables:

`RGBA line_color`
: The color of the line specified as a color name or Red,Green,Blue,Alpha

`LineType line_type`
: Determines if the line is drawn with `LINES`, `POINTS`, or both `LINES_AND_POINTS`. If the data is Strings, `STRINGS` is selected by default. `TRACE_COLORS` specifies that each trace of data gets a different color according to the color scale spectrum associated with the graph log. `VALUE_COLORS` specifies that the Y axis values are represented by colors using the color scale spectrum – useful for `vertical = NO_VERTICAL`. `THRESH_POINTS` draws points whenever the Y axis data value exceeds the `thresh` value.

`LineStyle line_style`
: Determines the how the line component of the GraphLine is drawn. Style choices are `SOLID (______)` , `DOT (......)`, `DASH (_ _ _ )`, `DASH_DOT (_._._ .)` .

`PointStyle point_style`
: Determines the how the point component of the GraphLine is drawn. Style choices are `NONE ( )` , `SMALL_DOT (.)`, `BIG_DOT (o)`, `TICK (|)`, `PLUS (+)`.

`point_mod`
: Specifies spacing of points – m = modulus value, or number of spaces between dots, and off = offset from start of data.

`bool negative_draw`
: Determines if a line is drawn from right to left if the values of the X-axis decrease (i.e., between traces).

`SharedYType shared_y`
: What to do with multiple graph lines that share the Y axis: `OVERLAY_LINES` overlays shared lines on top of each other, while `STACK_LINES` stacks shared values on top of each other in a non-overlapping manner.

`VerticalType vertical`
: How to manage the vertical (Y) axis. `FULL_VERTICAL` means use the full vertical axis for displaying Y values, `STACK_TRACES` means arrange subsequent traces of data (pass through the same X axis values) in non-overlapping vertically-arranged stacks, `NO_VERTICAL` means don't draw any vertical dimension at all (for `line_type = VLAUE_COLORS` or `THESH_POINTS`).

`FloatTwoDCoord trace_incr`
: Specifies increments in starting coordinates for each subsequent trace of a line across the same X axis values, producing a 3D-like effect. For example, x=-2,y=2 means move each subsequent trace to the left and upwards.

`float thresh`
: The threshold for `line_type` = `THRESH_POINTS`.

`GraphLogViewSpec* axis_spec`
: Determines which GraphLogViewSpec should be used as the Y-Axis for this ViewSpec's data to be scaled by. This can be any other column except if this column is used as a Y-axis for other columns.

`FixedMinMax range`
: This specifies fixed min or max values for the range of values to be plotted. Values are only applicable if corresponding `fix` buttons are on.

`int n_ticks`
: Maximum number of ticks to use in display of axis number labels (actual number may be less depending on the labels).

`GraphLogViewSpec* string_coords`
: Column that contains vertical coordinate values for positioning of String data labels.

The following functions are also available on the GraphViewSpec:

`PlotRows()`
: Plot the data across rows of grouped columns (x axis = column index, y axis = values for each column in one row) instead of down the columns.

`PlotCols()`
: Plot the data down columns (standard mode).

`GpShareAxis()`
: Make every element in this group share the same Y axis, which is the first in the group.

`GpSepAxes()`
: Make every element in this group have its own Y axis.

13.3.4.2 The Graph Area

The Graph Area of the GraphLogView appears as a region consisting of a field of background color overlaid with a horizontal X axis at the bottom of the Graph region, one or more vertical Y-axes on the left edge of the Graph region, lines of data, and optionally text labels scattered about. The lines of data will not initially be visible until data is actually recorded in the PDPLog. If separate graphs are selected, multiple such graphs may be arranged within the display.

The range of the Axes is determined by the GraphLogViewSpec of their corresponding fields. The tick marks are displayed as decimal numbers with a factor of 10 multiplier on the bottom of the Y-axes or left side of the x-axis. (i.e. If the axis tick read "2.15" and the multiplier is"10X" then the actual value is 21.5).

The GraphLogView Graph area can be zoomed and scrolled in a similar way as the Netview by selecting the *Mv/Scan* or *ReScale* mode buttons in the upper left of the GraphLogView. In Move/Scan mode, the graph's coordinates can be moved around by pressing and holding a mouse button in the background of the graph area. Moving with the left mouse button allows translations in both the Horizontal and Vertical planes, while the middle mouse button constrains movement to the Horizontal plane only and similarly the right mouse only allows Vertical movement. Likewise in Rescale mode, the left mouse button scales both dimensions, the middle mouse button only scales the Horizontal, and the right mouse button only scales the Vertical.

In Move/Scan mode if the mouse button is pressed while the pointer is over a line in the graph area the pointer will turn to a clenched hand and the value of the data at the mouse pointer location will appear beside the pointer. Dragging the mouse with left mouse button pressed will move a small box along the selected line, jumping from data point to data point and indicating the values. When the mouse button is released, the value and location box remain on the line until the `Init` button is pressed on the GraphLogView, or the log's buffer is overfilled.

13.3.4.3 GraphLogView Labels

In addition to the scanned value GraphLabels, general purpose GraphLabels can be manipulated using the *New*, *Delete*, and *Edit* GraphLabel action buttons on the left of the GraphLogView display. The "New" action button creates a label in the graph area. The "Delete" action button removes selected labels. The "Edit" action button brings up an Edit dialog for the selected labels. Editing a label allows the user to change the text or font of the label, using a standard XWindows font specification string. Note that *Select* mode must be activated to select these labels for the Edit and Delete operations, and that clicking wit the right-mouse-button (RMB) will automatically edit the label.

13.3.4.4 GraphLogView Variables

The GraphLogView provides the following variables for customizing the appearance of the Graph. To further customize the appearance of the individual lines of the graph, edit the GraphLogViewSpecs of the PDPLog's data fields.

`int x_axis_index`
: The index of the field to be used as the x (horizontal) axis. Usually it is easier to select this using the middle-mouse button on a field button.

`ViewLabel_Group labels`
: The listing of all the extra labels (string text) on the graph.

`ColorScaleSpec* colorspec`
: The color spectrum for this display (for TRACE_COLOR or VALUE_COLOR line displays).

`bool separate_graphs`
: Draw each group of lines sharing a Y axis using separate graphs

`PosTwoDCoord graph_layout`
: Arrangement of graphs for separate graphs – these are just rough constraints – actual geometry will be adjusted to fit number of graphs.

13.3.4.5 GraphLogView Functions

The GraphLogView provides the following functions:

`SetColorSpec(ColorScaleSpec* colors)`
: Set the color spectrum to use for color-coding values (NULL = use default)

`SetBackGround(RGBA background)`
: Sets the graph area's background color. When printing the graph it is useful to set this color to white.

`UpdateLineFeatures()`
: Update the update color, line type, point type, etc of lines in accordance with the current settings for the ordering of these features. Only visible lines are updated.

`SetLineFeatures(ColorType color_type, SequenceType sequence_1, SequenceType sequence_2, SequenceType sequence_3, bool visible_only)`
: Sets the features of the all the lines or only the visible lines if `visible_only` is set. The lines' features are determined by iterating over the possible choices in sequence_1, and then when those choices are exhausted, incrementing sequence_2 and repeating the possible choices in sequence_1 again. This is likewise extended to sequence_3 if need be until all of the lines' features have been set. The `color_type` argument is used to determine the colorscale to sequence over for the sequence arguments which are set to `COLORS`. The other possible choices of sequence types are `LINES` and `POINTS`.

`SetLineWidths(float line_width)`
: Set the widths of all lines in the graph to given value

`SetLineType(LineType line_type)`
: Set all line types to given type

`ShareAxisAfter(GraphViewSpec* axis_var)`
: Make all displayed variables after given axis_var share Y axis with axis_var

`ShareAxes()`
: Make all groups of columns share the same Y axis

`SeparateAxes()`
: Each column of data gets its own Y axis

`PlotRows()`
: Plot the data across rows of grouped columns (x axis = column index, y axis = values for each column in one row) instead of down the columns

`PlotCols()`
: Plot the data down columns (standard mode)

`SeparateGraphs(int x_layout, int y_layout)`
: Draw each group of lines sharing the same Y axis using separate graphs, with the given layout of graphs in the display. Sets the `graph_layout` and `separate_graphs` parameters.

`OneGraph()`
: Draw all data in one graph (default).

`TraceIncrement(float x_increment, float y_increment)`
: Each subsequent trace of data (pass through the same X axis values) is incremented by given amount, producing a 3D-like effect.

`StackTraces()`
: Arrange subsequent traces of data (pass through the same X axis values) in non-overlapping vertically-arranged stacks.

`UnStackTraces()`
: Subsequent traces of data (pass through the same X axis values) are plotted overlapping on top of each other.

`StackSharedAxes()`
: Arrange lines that share the same Y axis in non-overlapping vertically-arranged stacks.

`UnStackSharedAxes()`
: Lines that share the same Y axis are plotted overlapping on top of each other.

`SpikeRaster(float thresh)`
: Display spike rasters with given threshold for cutting a spike (`trace_incr.y = 1`, `vertical = NO_VERTICAL`, `line_type = TRESH_POINTS`).

`ColorRaster()`
: Display values as rasters of lines colored according to the values of the lines.

`StandardLines()`
: Get rid of raster-style display and return to 'standard' graph line display.

`SetXAxis (char* nm)`
: Sets the `x-axis_index` of the view to the field named `nm`. If there is no field named `nm` then the index is unchanged.

13.3.5 The Grid LogView

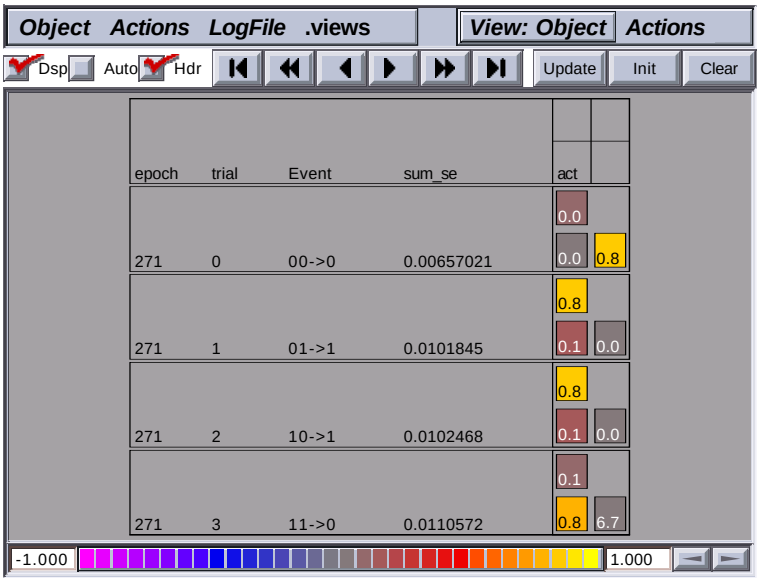


Figure: 10 The Grid Log Viewer

The GridLogView is similar to the TextLogView in that it displays from top to bottom sequential lines of Log data. It differs however in that the values of the data can be displayed

as colored blocks similar to the units in the NetView as well as the text output style of the TextLogView.

In addition, the columns of Log data can be arranged in a custom fashion in a 2-D grid for each "line" of data. This is useful especially for MonitorStats which generate a separate column of data for each value recorded from the NetView. For Example: If a MonitorStat was monitoring the activations a 5x5 layer it would generate 25 columns of data. In the GridLogView these 25 columns could be arranged as in a 5x5 grid in the same layout as the units in the monitored Layer. Indeed, a monitored layer automatically sets its geometry to be used for the default configuration of the GridLogView.

The header label information in the GridLogView is also different in that is is arranged in the same grid layout as the data and its field header name labels are selectable and manipulable with the mouse. This allows complete interactive configuration of the GridLogView display, as follows:

Move with left-mouse-button (LMB)
: Clicking and dragging on a header item will move it, in unit-level chunks, and will then update the display of all the corresponding data for that column. Items can be moved anywhere, and overlap is not a problem (previous versions would crash upon overlap).

Reshape with Shift-LMB or middle-mouse-button
: For items that are displayed in the header with a grid of boxes (e.g., monitored values), you can reshape the grid. The grid will not shrink below the number of elements to be displayed, so keep that in mind as you reshape it.

Edit with right-mouse-button (RMB)
: Clicking with the right mouse button will pull up the GridLogViewSpec for that item for more detailed configuration (see Section 13.3.5.2 [log-views-grid-custom], page 224).

Resizing the GridLogView window changes the number of visible data lines, always in such a way as to keep the grids roughly square. So, if you want to have more items in the display, this can also be achieved by making the window narrower!

13.3.5.1 The GridLogView Class

The GridLogView class provides the following variables for customizing the display:

`BlockFill fill_type`
: Controls the way in which the Log data values are displayed as blocks. The `COLOR` setting fills the block with its value's respective color from the colorbar at the bottom of the GridLogView display. The `AREA` and `LINEAR` fill the block with size-varying spirals in a manner similar to the spirals in the NetView and described in more detail in that section. (see Section 10.6 [net-view], page 149).

`ColorScale* colorspec`
: The colorscale to use for the colorbar at the bottom of the GridLogView display (see Section 6.7 [gui-colors], page 67).

`int block_size`
: The size (width and height) of the block area for the value blocks — note that this is only a *maximum* value – if the blocks need to be shrunk to fit everything in the current window size, they will be!

`int block_border_size`
: The horizontal and vertical spacing between the blocks in the GridLogView.

`bool header_on`
: Whether to display the header or not (also accessable through the *Hdr* check box on the display itself).

`bool auto_scale`
: Whether to scale the block colors by the actual values of items displayed, or to use a fixed range given by the colorscale at the bottom of the screen (also accessable through the *Auto* check box on the display itself).

`FontSpec view_font`
: Font to use for rendering of all text in the display.

The GridLogView also provides the following functions:

`SetColorSpec(ColorScaleSpec* colors)`
: Set the color spectrum to use for color-coding values (NULL = use default)

`SetBlockFill(BlockFill fill_typ)`
: Set's the way that data blocks representing values are displayed according to the `fill_type` variable of the GridLogView, either `COLOR`, `AREA`, or `LINEAR`.

`SetBlockSizes(int block_sz, int border sz)`
: Set's the *maximum* size of the data blocks – actual size can be smaller depending on size of display.

`UpdateGridLayout(MatrixLayout layout)`
: Arranges the columns of data to fit without gaps, according to the geometry of the `view_spec`. Possible choices for `layout` are:

  `LFT_RGT_BOT_TOP`

  `LFT_RGT_TOP_BOT`

  `BOT_TOP_LFT_RGT`

  `TOP_BOT_LFT_RGT`

  These are explained more fully in the next section.

`SetViewFontSize(int point_size)`
: Set the point size of the font used for labels in the display

`AllBlockTextOn()`
: Turn text on (in addition to the block) for all block displayed items.

`AllBlockTextOff()`
: Turn text off for all block displayed items (so they only display as colored blocks).

13.3.5.2 Customizing the GridLogView Display

In addition to the general customization variables available on the GridLogView object, the GridViewSpec objects allow precise control over the positioning and display style of the data elements in the GridLogView.

Each of the columns of data in the PDPLog has an associated GridViewSpec in the GridLogView. Some of the columns of data have been grouped together by their generating statistics and will also have a DT_GridViewSpec for the entire group of data columns.

The GridViewSpec provides the following customizable variables:

`TDGeometry pos`
: Indicates the **absolute** position in the grid in which this data column's values will appear (i.e., even if the GridLogViewSpec is part of an associated group of GridLogViewSpecs this variable will indicate the absolute position of this column's value). The Z component of the position is ignored. (0,0) is in the lower left corner of each line in the GridLogView.

`DisplayStyle display_style`
: Controls the way in which the Log data values for this column are displayed. The choices are:

  `TEXT` Display the values as text only

  `BLOCK` Display the values as a colored block

  `TEXT_AND_BLOCK`
  : Display the block overlaid with with text

`bool scale_on`
: If set, and if the display style includes BLOCK, then this column of data will participate in setting the overall scaling of values.

For data columns organized into groups (e.g., the output of a MonitorStat on a layer's worth of activations), the DT_GridViewSpec provides the following variables:

`TDGeometry pos`
: Indicate the position in the grid in which this group of data columns' values will appear. The Z component of the position is ignored. (0,0) is in the lower left corner of each line of the GridLogView.

`TDGeometry geom`
: Controls the size of the region in which the GridViewSpecs of the this DT_GridViewSpec are positioned.

`MatrixLayout layout`
: Controls how the individual elements within the group are positioned relative to each other, as follows (note this is also used in the UpdateLayout function):

  `LFT_RGT_BOT_TOP`
  : Increment column first, then decrement row, start at bottom left:

```
789
456
123
```

`LFT_RGT_TOP_BOT`
: Increment column first, then increment row, start at top left:

```
123
456
789
```

`BOT_TOP_LFT_RGT`
: Decrement row first, then increment column, start at bottom left:

```
369
258
147
```

`TOP_BOT_LFT_RGT`
: Increment row first, then increment column, start at top left:

```
147
258
369
```

`Bool use_gp_name`
: When this variable is set to `true` a single label of the name of the group's DT_GridViewSpec is used instead of individual labels for all the GridViewSpecs of the group. This also determines whether the display_style used will be that of the group or the individual elements.

`DisplayStyle display_style`
: Controls how the data is displayed, either TEXT, BLOCK, or both.

The DT_GridViewSpec provides the following functions:

`UpdateLayout(MatrixLayout ml)`
: Repositions the GridViewSpecs according to the `ml` and the `geom` variable. See `layout` variable above, which is what is used if `DEFAULT` is selected for the `ml`, for descriptions of the layout options.

14 Backpropagation

Backpropagation is perhaps the most commonly used neural network learning algorithm. Several different "flavors" of backpropagation have been developed over the years, several of which have been implemented in the PDP++ software, including the use of different error functions such as cross-entropy, and recurrent backprop, from the simple recurrent network to the Almeida-Pineda algorithm up to the real-time continuous recurrent backprop. The implementation allows the user to extend the unit types to use different activation and error functions in a straightforward manner.

Note that the simple recurrent networks (SRN, a.k.a. Elman networks) are described in the feedforward backprop section, as they are more like feedforward networks than the fully recurrent ones.

14.1 Feedforward Backpropagation

14.1.1 Overview of the Bp Implementation

The basic structure of the backpropagation algorithm is reviewed in the tutorial (see Section 4.1 [tut-bp], page 20). In short, there are two phases, an activation propagation phase, and an error backpropagation phase. In the simplest version of Bp, both of these phases are strictly feed-forward and feed-back, and are computed sequentially layer-by-layer. Thus, the implementation assumes that the layers are organized sequentially in the order that activation flows.

In the recurrent versions, both the activation and the error propagation are computed in two steps so that each unit is effectively being updated simultaneously with the other units. This is done in the activation phase by first computing the net input to each unit based on the other units current activation values, and then updating the activation values based on this net input. Similarly, in the error phase, first the derivative of the error with respect to the activation (dEdA) of each unit is computed based on current dEdNet values, and then the dEdNet values are updated based on the new dEdNet.

To implement a new algorithm like Bp in PDP++, one creates new class types that derive from the basic object types that make up a network (see Chapter 10 [net], page 131), and scheduling processes (see Chapter 12 [proc], page 184). These new classes inherit all the functionality from the basic types, and specify the details appropriate for a particular algorithm. There are two ways in which this specification happens—overloading (replacing) existing functions, and adding new ones (see Section 8.1.1 [obj-basics-obj], page 107).

The new classes defined in the basic Bp implementation include: **BpConSpec, BpCon, BpCon_Group, BpUnit, BpUnitSpec, BpTrial**, the role of which should be clear from their names. In addition, we have added a **CE_Stat** that computes the cross-entropy error statistic, much in the same way **SE_Stat** does (see Section 12.6.1 [proc-stats-se], page 202).

Bias weights in PDP++ are implemented by adding a **BpCon** object to the **BpUnit** directly, and not by trying to allocate some kind of self projection or some other scheme like that. In addition, the **BpUnitSpec** has a pointer to a **BpConSpec** to control the updating etc of the bias weight. Thus, while some code was written to support the special bias weights on units, it amounts to simply calling the appropriate function on the **BpConSpec**.

The processing hierarchy for feed-forward Bp requires only a specialized Trial process: **BpTrial**, which runs both the feed-forward activation updating and error backpropagation phases.

14.1.2 Bp Connection Specifications

In addition to the weight itself, the connection type in Bp, **BpCon**, has two additional variables:

`float dwt` The most recently computed change in the weight term. It is computed in the `UpdateWeights` function.

`float dEdW`
: The accumulating derivative of the error with respect to the weights. It is computed in the `Compute_dWt` function. It will accumulate until the `UpdateWeights` function is called, which will either be on a trial-by-trial or epoch-wise (batch mode) basis.

The connection specifications control the behavior and updating of connections (see Section 10.5 [net-con], page 146). Thus, in Bp, this is where you will find thinks like the learning rate and momentum parameters. A detailed description of the parameters is given below:

`float lrate`
: The learning rate parameter. It controls how fast the weights are updated along the computed error gradient. It should generally be less than 1 and harder problems will require smaller learning rates.

`float momentum`
: The momentum parameter determines how much of the previous weight change will be retained in the present weight change computation. Thus, weight changes can build up momentum over time if they all head in the same direction, which can speed up learning. Typical values are from .5 to .9, with anything much lower than .5 making little difference.

`MomentumType momentum_type`
: There are a couple of different ways of thinking about how momentum should be applied, and this variable controls which one is used. According to `AFTER_LRATE`, momentum is added to the weight change *after* the learning rate has been applied:

```
cn->dwt = lrate * cn->dEdW + momentum * cn->dwt;
cn->wt += cn->dwt;
```

This was used in the original pdp software. The `BEFORE_LRATE` model holds that momentum is something to be applied to the gradient computation itself, not to the actual weight changes made. Thus, momentum is computed *before* the learning rate is applied to the weight gradient:

```
cn->dwt = cn->dEdW + momentum * cn->dwt;
cn->wt += lrate * cn->dwt;
```

Finally, both of the previous forms of momentum introduce a learning rate confound since higher momentum values result in larger effective weight changes

when the previous weight change points in the same direction as the current one. This is controlled for in the NORMALIZED momentum update, which normalizes the total contribution of the previous and current weight changes (it also uses the BEFORE_LRATE model of when momentum should be applied):

```
cn->dwt = (1.0 - momentum) * cn->dEdW + momentum * cn->dwt;
cn->wt += lrate * cn->dwt;
```

Note that normalized actually uses a variable called momentum_c which is precomputed to be 1.0 - momentum, so that this extra computation is not incurred needlessly during actual weight updates.

`float decay`

Controls the magnitude of weight decay, if any. If the corresponding decay_fun is NULL weight decay is not performed. However, if it is set, then the weight decay will be scaled by this parameter. Note that weight decay is applied *before* either momentum or the learning rate is applied, so that its effects are relatively invariant with respect to manipulations of these other parameters.

`decay_fun`

The decay function to be used in computing weight decay. This is a pointer to a function, which means that the user can add additional decay functions as they wish. However, the default ones are Bp_Simple_WtDecay, which simply subtracts a fraction of the current weight value, and Bp_WtElim_WtDecay, which uses the "weight elimination" procedure of *Weigand, Rumelhart, and Huberman, 1991*. This procedure allows large weights to avoid a strong decay pressure, but small weights are encouraged to be eliminated:

```
float denom = 1.0 + (cn->wt * cn->wt);
cn->dEdW -= spec->decay * ((2 * cn->wt) / (denom * denom);
```

The ratio of the weight to the denom value is roughly proportional to the weight itself for small weights, and is constant for weights larger than 1.

14.1.3 Bp Unit Specifications

The unit in Bp contains a bias weight and the various derivative terms that are accumulated during the backpropagation phase:

`BpCon bias`

Contains the bias weight and its associated derivative and weight change values. The bias weight is controlled by the bias_spec on the **BpUnitSpec**.

`float err` Contains the actual error or cost associated with the unit. It is a function of the difference between the activation and the target value, so it only shows up for those units that have target activation values. It is not to be confused with the derivative of the activation with respect to the error, which is dEdA.

`float dEdA`

The derivative of the error with respect to the activation of the unit. For output units using the squared-error function, it is simply (targ - act). For hidden units, it is the accumulation of the backpropagated dEdNet values times the intervening weights from the units to which the unit sends activation.

`float dEdNet`
: The derivative of the error with respect to the net input of the unit. It is simply the `dEdA` times the derivative of the activation function, which is `act * (1 - act)` for standard sigmoidal units.

The unit specifications for Bp control what kind of error function is being used, the parameters of the activation function, and the functions on the spec orchestrate the computation of the activation and error backpropagation phases:

`SigmoidSpec sig`
: These are the parameters of the sigmoidal activation function. The actual range of this activations are determined by the `act_range` parameters, and the `sig` parameters determine the gain and any fixed offset of the function (the offset is like a fixed bias term).

`float err_tol`
: The error tolerance allows activation values that are sufficiently close to the target activation to be treated as though they were equal to the target value. Reasonable values of this parameter are from .02 to .1, and its use prevents the large accumulation of weight values that happens when the unit keeps trying to get closer and closer to an activation of 1 (for example), which is impossible.

`BpConSpec_SPtr bias_spec`
: A pointer to a **BpConSpec** that controls the updating of the unit's bias weight. Typically, this points to the same **BpConSpec** that updates the rest of the weights in the network, but it is possible to have special **BpConSpec's** that do different things to the bias weights, like initialize them to moderate negative values, for example.

`err_fun` A pointer to the error function to use in computing error for output units that have target values. The function computes both `err` and `dEdA` values (the former typically being the square of the latter). While the user can define additional error functions, the two that come with the standard distribution are `Bp_Squared_Error` and `Bp_CrossEnt_Error`, which compute the squared error and cross-entropy error functions, respectively.

14.1.4 The Bp Trial Process

The **BpTrial** process is the only Bp-specific process type needed to perform simple feed-forward backprop. The `Loop` function of this process simply propagates activation forwards through the network, and then propagates the error backwards. This assumes that the layers are ordered sequentially in a feed-forward manner. Note that the process does not actually "loop" over anything, so it has no counter. See Section 12.3.4 [proc-levels-trial], page 194 for more information on the trial process type.

The following functions are defined on the trial process, each of which performs one step of the backpropagation computations:

`Compute_Act()`
: Goes layer-by-layer and computes the net input and the activation of the units in the layer.

`Compute_Error()`
: Computes the error on the output units, which is only done during testing, and not training.

`Compute_dEdA_dEdNet()`
: Computes the derivative of the error with respect to the activation and then with respect to the net inputs. This goes in reverse order through the layers of the network.

`Compute_dWt()`
: Computes the dEdW for all the weights in the network.

14.1.5 Variations Available in Bp

There are several different **BpUnitSpec** and **BpConSpec** types available that perform variations on the generic backpropagation algorithm.

LinearBpUnitSpec implements a linear activation function

ThreshLinBpUnitSpec implements a threshold linear activation function with the threshold set by the parameter `threshold`. Activation is zero when net is below threshold, net-threshold above that.

NoisyBpUnitSpec adds noise to the activations of units. The noise is specified by the `noise` member.

StochasticBpUnitSpec computes a binary activation, with the probability of being active a sigmoidal function of the net input (e.g., like a Boltzmann Machine unit).

RBFBpUnitSpec computes activation as a Gaussian function of the distance between the weights and the activations. The variance of the Gaussian is spherical (the same for all weights), and is given by the parameter `var`.

BumpBpUnitSpec computes activation as a Gaussian function of the standard dot-product net input (not the distance, as in the RBF). The mean of the effectively uni-dimensional Gaussian is specified by the `mean` parameter, with a standard deviation of `std_dev`.

ExpBpUnitSpec computes activation as an exponential function of the net input (e^net). This is useful for implementing SoftMax units, among other things.

SoftMaxBpUnitSpec takes one-to-one input from a corresponding exponential unit, and another input from a LinearBpUnitSpec unit that computes the sum over all the exponential units, and computes the division between these two. This results in a SoftMax unit. Note that the LinearBpUnitSpec must have fixed weights all of value 1, and that the SoftMax-Unit's must have the one-to-one projection from exp units first, followed by the projection from the sum units. See '`demo/bp_misc/bp_softmax.proj.gz`' for a demonstration of how to configure a SoftMax network.

HebbBpConSpec computes very simple Hebbian learning instead of backpropagation. It is useful for making comparisons between delta-rule and Hebbian leanring. The rule is simply `dwt = ru->act * su->act`, where `ru->act` is the target value if present.

ErrScaleBpConSpec scales the error sent back to the sending units by the factor `err_scale`. This can be used in cases where there are multiple output layers, some of which are not supposed to influence learning in the hidden layer, for example.

DeltaBarDeltaBpConSpec implements the delta-bar-delta learning rate adaptation scheme *Jacobs, 1988*. It should only be used in batch mode weight updating. The connection type must be **DeltaBarDeltaBpCon**, which contains a connection-wise learning rate parameter. This learning rate is additively incremented by `lrate_incr` when the sign of the current and previous weight changes are in agreement, and decrements it multiplicatively by `lrate_decr` when they are not. The demo project '`demo/bp_misc/bp_ft_dbd.proj.gz`' provides an example of how to set up delta-bar-delta learning. The defaults file '`bp_dbd.def`' provides a set of defaults that make delta-bar-delta connections by default.

14.1.6 Simple Recurrent Networks in Bp

Simple recurrent networks (SRN) *Elman, 1988* involve the use of a special set of context units which copy their values from the hidden units, and from which the hidden units receive inputs. Thus, it provides a simple form of recurrence that can be used to train networks to perform sequential tasks over time. **New for 3.0:** the **BpWizard** has a `Network/SRNContext` function that will automatically build an SRN context layer as described below.

The implementation of SRN's in PDP++ uses a special version of the **BpUnitSpec** called the **BpContextSpec**. This spec overloads the activation function to simply copy from a corresponding hidden unit. The correspondence between hidden and context units is established by creating a single one-to-one projection into the context units from the hidden units. The context units look for the sending unit on the other side of their first connection in their first connection group for the activation to copy. This kind of connection should be created with a **OneToOnePrjnSpec** (see Section 10.3.2 [net-prjn-spec], page 138).

Important: The context units should be in a layer that *follows* the hidden units they copy from. This is because the context units should provide input to the hidden units before copying their activation values. This means that the hidden units should update themselves first.

The context units do not have to simply copy the activations directly from the hidden units. Instead, they can perform a time-averaging of information through the use of an updating equation as described below. The parameters of the context spec are as follows:

`float hysteresis`
: Controls the rate at which information is accumulated by the context units. A larger hysteresis value makes the context units more sluggish and resistant to change; a smaller value makes them incorporate information more quickly, but hold onto it for a shorter period of time:

```
u->act = (1.0 - hysteresis) * hu->act + hysteresis * u->act;
```

`Random initial_act`
: These parameters determine the initial activation of the context units. Unlike other units in a standard Bp network, the initial state of the context units is actually important since it provides the initial input to the hidden units from the context.

Note that the SRN typically requires a sequence model of the environment, which means using the sequence processes (see Section 12.4.1 [proc-special-seq], page 195). Typically, the activations are initialized at the start of a sequence (including the context units), and then a sequence of related events are presented to the network, which can then build up a context representation over time since the activations are not initialized between each event trial.

The defaults file '`bp_seq.def`' contains a set of defaults for Bp that will create sequence processes by default (see Section 14.1.7 [bp-defs], page 232).

The demo project '`demo/bp_srn/srn_fsa.proj.gz`' is an example of a SRN network that uses the sequence processes. It also illustrates the use of a **ScriptEnv** where a CSS script is used to dynamically create new events that are generated at random from a finite state automaton.

14.1.7 Bp Defaults

The following default files (see Section 9.5 [proj-defaults], page 126) are available for configuring different versions of the Bp objects:

'`bp.def`' This is the standard defaults file, for standard feedforward backpropagation with sigmoidal units.

'`bp_seq.def`'
This is for doing simple recurrent networks (see Section 14.1.6 [bp-srn], page 231) in Bp. It creates SequenceEpoch and SequenceProc processes instead of a standard EpochProcess.

'`bp_dbd.def`'
This creates delta-bar-delta connections by default (including bias connections).

14.1.8 Bp Implementation Details

Many of the relevant details are discussed above in the context of the descriptions of the basic Bp classes. This section provides a little more detail that might be useful to someone who wanted to define their own versions of Bp classes, for example.

Support for the activation updating phase of Bp is present in the basic structure of the PDP++ Section 10.4 [net-unit], page 143 and Section 10.5 [net-con], page 146 types, specifically in the `Compute_Net` and `Compute_Act` functions. We overload `Compute_Act` to implement the sigmoidal activation function.

The error backpropagation phase is implemented with three new functions at both the unit and connection level. The unit spec functions are:

`Compute_dEdA(BpUnit* u)`
Computes the derivative of the error with respect to the activation. If the unit is an output unit (i.e., it has the `ext_flag` `TARG` set), then it just calls the error function to get the difference between the target and actual output activation. If it is not an output unit, then it iterates through the sending connection groups (i.e., through the connections to units that this one sends activation to), and accumulates its `dEdA` as a function of the connection weight times the other unit's `dEdNet`. This is done by calling the function `Compute_dEdA` on the

sending connection groups, which calls this function on the **BpConSpec**, which is described below.

`Compute_dEdNet(BpUnit* u)`
: Simply applies the derivative of the activation function to the already-computed `dEdA` to get the derivative with respect to the net input.

`Compute_Error(BpUnit* u)`
: This function is not used in the standard training mode of Bp, but is defined so that the error can be computed when a network is being tested, but not trained.

`Compute_dWt(Unit* u)`
: Computes the derivative of the error with respect to the weight (`dEdW`) for all of the unit's connections. This is a function of the `dEdNet` of the unit, and the sending unit's activation. This function is defined as part of the standard **UnitSpec** interface, and it simply calls the corresponding `Compute_dWt` function on the `ConSpec` for all of the receiving connection groups. In Bp, it also calls `Compute_dWt` on for the bias weight.

`UpdateWeights(Unit* u)`
: Updates the weights of the unit's connections based on the previously computed `dEdW`. Like `Compute_dWt`, this function is defined to call the corresponding one on connection specs. Also, it updates the bias weights. Note that this function is called by the **EpochProcess**, and not by the algorithm-specific **BpTrial** directly.

The corresponding connection spec functions are as follows. Note that, as described in Section 10.5 [net-con], page 146, there are two versions of every function defined in the **ConSpec**. The one with a `C_` prefix operates on an individual **Connection**, while the other one iterates through a group of connections and calls the connection-specific one.

`float C_Compute_dEdA(BpCon* cn, BpUnit* ru, BpUnit* su)`

`float Compute_dEdA(BpCon_Group* cg, BpUnit* su)`
: These accumulate the derivative of the error with respect to the weights and return that value, which is used by the unit to increment its corresponding variable. Note that this is being called on the *sending* connection groups of a given unit, which is passed as an argument to the functions. The computation for each connection is simply the `dEdNet` of the unit that receives from the sending unit times the weight in between them.

`float C_Compute_dWt(BpCon* cn, BpUnit* ru, BpUnit* su)`

`float Compute_dWt(Con_Group* cg, Unit* ru)`
: These increment the `dEdW` variable on the receiving connections by multiplying the sending unit's activation value times the receiving unit's `dEdNet`.

`float B_Compute_dWt(BpCon* cn, BpUnit* ru)`
: The bias-weight version of this function. It does not multiply times the sender's activation value (since there isn't one!).

`float C_Compute_WtDecay(BpCon* cn, BpUnit* ru, BpUnit* su)`
: This calls the weight decay function on the given connection, if it is not NULL. It is meant to be called as part of a `C_UpdateWeights` function.

`float C_BEF_UpdateWeights(BpCon* cn, Unit* ru, Unit* su)`

`float C_AFT_UpdateWeights(BpCon* cn, Unit* ru, Unit* su)`

`float C_NRM_UpdateWeights(BpCon* cn, Unit* ru, Unit* su)`

`float UpdateWeights(Con_Group* cg, Unit* ru)`

These are the functions that update the weights based on the accumulated `dEdW`. There is a different version of the connection-specific code for each of the different `momentum_type` values, and the group-level function has a separate loop for each type, which is more efficient that checking the type at each connection.

`float B_UpdateWeights(BpCon* cn, Unit* ru)`

The bias-weight version of the function, which checks the `momentum_type` variable and calls the appropriate `C_` function.

The following is a chart describing the flow of processing in the Bp algorithm, starting with the epoch process, since higher levels do not interact with the details of particular algorithms:

```
EpochProcess: {
  Init: {
    environment->InitEvents();          // init events (if dynamic)
    event_list.Add() 0 to environment->EventCount(); // get list of events
    if(order == PERMUTE) event_list.Permute();       // permute if necessary
    GetCurEvent();                      // get pointer to current event
  }
  Loop (trial): {                       // loop over trials
    BpTrial: {                          // trial process (one event)
      Init: {                           // at start of trial
        cur_event = epoch_proc->cur_event; // get cur event from epoch
      }
      Loop (once): {                    // only process this once per trial
        network->InitExterns();         // init external inputs to units
        cur_event->ApplyPatterns(network); // apply patterns to network
        Compute_Act(): {                // compute the activations
          network->layers: {            // loop over layers
            if(!layer->ext_flag & Unit::EXT) // don't compute net for clamped
              layer->Compute_Net();     // compute net inputs
            layer->Compute_Act();       // compute activations from net in
          }
        }
        Compute_dEdA_dEdNet(): {        // backpropagate error terms
          network->layers (backwards): { // loop over layers backwards
            units->Compute_dEdA();   // get error from other units or targets
            units->Compute_dEdNet(); // add my unit error derivative
          }
        }
        network->Compute_dWt();         // compute weight changes from error
      }
    }
    if(wt_update == ON_LINE or wt_update == SMALL_BATCH and trial.val % batch_n)
      network->UpdateWeights(); // after trial, update weights if necc
    GetCurEvent();              // get next event
  }
  Final:
    if(wt_update == BATCH)  network->UpdateWeights(); // batch weight updates
}
```

14.2 Recurrent Backpropagation

Recurrent backpropagation (RBp) extends the basic functionality of feed-forward backprop to networks with recurrently interconnected units, which can exhibit interesting dynamical properties as activation propagates through the network over time.

14.2.1 Overview of the RBp Implementation

The recurrent backprop implementation (RBp) defines a new set of types that are derived from the corresponding Bp versions: **RBpConSpec, RBpUnit, RBpUnitSpec, RBpTrial,**

RBpSequence. Note that RBp uses the same Connection type as Bp. In addition, support for the Almeida-Pineda algorithm is made possible by the following set of process types, which control the activation and backpropagation phases of that algorithm, which otherwise uses the same basic types as RBp: **APBpCycle, APBpSettle, APBpTrial, APBpMaxDa_De**.

There are a couple of general features of the version of recurrent backprop implemented in PDP++ that the user should be aware of. First of all, the model used is that of a discrete approximation to a continuous dynamic system, which is defined by the sigmoidal activation of the net input to the units. The granularity of the discrete approximation is controlled by the `dt` parameter, which should be in the range between 0 and 1, with smaller values corresponding to a finer, closer to continuous approximation. Thus, the behavior of the network should be roughly similar for different `dt` values, with the main effect of `dt` being to make updating smoother or rougher.

Also, there are two ways in which the units can settle, one involves making incremental changes to the activation values of units, and the other involves making incremental changes to the net inputs. The latter is generally preferred since it allows networks with large weights to update activations quickly compared to activation-based updates, which have a strict ceiling on the update rate since the maximum activation value is 1, while the maximum net input value is unbounded.

As in standard backpropagation, recurrent backprop operates in two phases: activation propagation and error backpropagation. The difference in recurrent backprop is that both of these phases extend over time. Thus, the network is run for some number of activation update cycles, during which a record of the activation states is kept by each unit, and then a backpropagation is performed that goes all the way back in time through the record of these activation states. The backpropagation happens between the receiving units at time t and the sending units at the previous time step, time t-1. Another way of thinking about this process is to unfold the network in time, which would result in a large network with a new set of layers for each time step, but with the same set of weights used repeatedly for each time step unfolding. Doing this, it is clear that the sending units are in the previous time step relative to the receiving units.

The exact length of the activation propagation phase and the timing and frequency of the backpropagation phases can be controlled in different ways that are appropriate for different kinds of tasks. In cases where there is a clearly-defined notion of a set of distinct temporal sequences, one can propagate activation all the way through each sequence, and then backpropagate at the end of the sequence. This is the default mode of operation for the processes.

There are other kinds of environments where there is no clear boundary between one sequence and the next. This is known as "real time" mode, and it works by periodically performing a backpropagation operation after some number of activation updates have been performed. Thus, there is a notion of a "time window" over which the network will be sensitive to temporal contingencies through the weight updates driven by a single backpropagation operation. In addition, these backpropagations can occur with a period that is less than the length of the time window, so that there is some overlap in the events covered by successive backpropagation operations. This can enable longer-range temporal contingencies to be bootstrapped from a series of overlapping backpropagations, each with a smaller time window.

There is a simpler variation of a recurrent backpropagation algorithm that was invented by Almeida and Pineda, and is named after them. In this algorithm, the activation updating phase proceeds iteratively until the maximum change between the previous and the current activation values over all units is below some criterion. Thus, the network settles into a stable attractor state. Then, the backpropagation phase is performed repeatedly until it too settles on a stable set of error derivative terms (i.e., the maximum difference between the derivative of the error for each unit and the previously computed such derivative is below some threshold). These asymptotic error derivatives are then used to update the weights. Note that the backpropagation operates repeatedly on the asymptotic or stable activation values computed during the first stage of settling, and not on the trajectory of these activation states as in the "standard" version of RBp. The Almeida-Pineda form of the algorithm is enabled by using the **APBp** processes, which compute the two phases of settling over cycles of either activation propagation or error backpropagation.

14.2.2 RBp Connection Specifications

Since the difference between recurrent backprop and standard feed-forward backprop is largely a matter of the process-level orchestration of the different phases, the connection specifications are identical between the two, with one small difference. Please refer to Section 14.1.2 [bp-con], page 227 for a description of the relevant parameters.

The exception is only at the level of the equations used to compute the error derivative terms, which are modified to use the previous activation value of the sending unit, instead of the current activation, since the idea is to reduce the error that results at time t as a function of the activation states that lead up to it, those at time t-1.

Since RBp networks are complex dynamical systems, it is often useful to place restrictions on the kinds of weights that can develop, in order to encourage certain kinds of solutions to problems. One rather direct way of doing this is by simply limiting the weights to not go above or below certain values, or to restrict them to be symmetrical. These can be accomplished by editing the **ConSpec**.

14.2.3 RBp Unit Specifications

The unit-level specifications contain most of the RBp-specific parameters, although some important ones are also present in the **RBpTrial** process. Note that the `dt` parameter should be the same for all unit specs used in a given network. Also, this parameter is copied automatically to the **RBpTrial** process, which also needs to know the value of this parameter. Thus, the unit spec is the place to change `dt`, not the trial process.

The unit object in RBp is essentially the same as the **BpUnit**, except for the addition of variables to hold the previous values of all the state variables, and special circular buffers to hold the entire record of activation state variables over the update trajectory. These are described in greater detail in Section 14.2.9 [rbp-impl], page 242.

`float dt` Controls the time-grain of activation settling and error backpropagation as described above. In `ACTIVATION` mode, the activations are updated towards the raw activation value computed as a sigmoid function of the current net input by an amount proportional to `dt`:

```
u->da = dt * (u->act_raw - u->prv_act);
```

```
u->act = u->prv_act + u->da;
```

Similarly, in `NET_INPUT` mode, the net-inputs are moved towards the current raw net input proportional to the size of `dt`:

```
u->da = dt * (u->net - u->prv_net);
u->net = u->prv_net + u->da;
```

`TimeAvgType time_avg`
: Controls the type of time-averaging to be performed. `ACTIVATION` based time-averaging, as shown above, adapts the current activations towards the raw activation based on the current net input, while `NET_INPUT` based time-averaging, also shown above, adapts the net input towards the current raw value. The latter is generally preferred since it allows networks with large weights to update activations quickly compared to activation-based updates, which have a strict ceiling on the update rate since the maximum activation value is 1, while the maximum net input value is unbounded.

`bool soft_clamp`
: Soft clamping refers to the application of an environmental input to the network as simply an additional term in the unit's net input, as opposed to a hard-clamped pre-determined activation value. Soft clamping allows input units to behave a little more like hidden units, in that raw inputs are only one source of influence on their activation values.

`float soft_clamp_gain`
: A strength multiplier that can be used to set the level of influence that the inputs have in soft-clamp mode. This allows the user to use the same environments for hard and soft clamping, while still giving the soft-clamp values stronger influence on the net input than would be the case if only 0-1 values were being contributed by the external input.

`bool teacher_force`
: A modification of the RBp algorithm where the activation values are "forced" to be as given by the teaching (target) values. Given that the error is back-propagated over a long series of time steps, this can help error on previous time steps be computed as if the later time steps were actually correct, which might help in the bootstrapping of representations that will be appropriate when the network actually is performing correctly.

`bool store_states`
: This flag determines if activity states are stored over time for use in performing a backpropagation through them later. This usually must be true, except in the Almeida-Pineda algorithm, or when just testing the network.

`Random initial_act`
: Sets the parameters for the initialization of activation states at the beginning of a sequence. This state forms the 0th element of the sequence of activations.

14.2.4 The RBp Trial Process

In order to accommodate both the real-time and discrete sequence-based processing modes of RBp, the lowest level of processing in RBp is still the Trial, just as in regular Bp.

Thus, each cycle of activation update is performed by the trial. In addition, the trial process looks at the number of stored activation states that have been accumulated in the network's units, and if this is equal to the time-window over which backprop is supposed to occur, the trial process will then perform a backpropagation through all of these stored activation states. Thus, the scheduling of backpropagations is fairly autonomous, which makes real-time mode work well. When not operating in real-time mode, the time-window for error backpropagation is automatically set to be the total duration of the current sequence. This makes it easy to use variable length sequences, etc.

The distinction between whether the network is trained in real-time or sequence-based mode is based on the kinds of processes that are created above the level of the **RBpTrial** process, and on the setting of the `real_time` flag on the **RBpTrial** process itself. If using the sequence-based mode, where backpropagations are performed at the end of discrete sequences of events, then the user should create a Sequence-based process hierarchy, which includes a **SequenceEpoch**, an **RBpSequence** process, and finally a **RBpTrial** process. If one is using real-time mode, only a regular **EpochProcess** and a **RBpTrial** process need to be used.

The following parameters are available on the **RBpTrial**. Note that all of the parameters are expressed in terms of the abstract time units, and not in terms of the specific ticks of the discrete clock on which the actual events are presented to the network, activations are updated, etc. This makes the parameters invariant with respect to changes in `dt`, which controls the size of a tick of discrete time.

`float time`
: The current time, relative to the start of the most recent sequence, or since the units were last initialized if in real-time mode. It is a read-only variable.

`float dt`
: The delta-time increment, which is copied automatically from the units in the network. It is used in updating the time in the trial process.

`float time_window`
: Determines the time window over which error backpropagation will be performed. Thus, units will hold their activation states for this long before a backpropagation will occur.

`float bp_gap`
: The time to wait in between successive backpropagations after an initial activation settling time of `time_window` in duration. This is used primarily in real-time mode, and controls the amount of overlap between successive backpropagations. For example, a `time_window` of 4 and a `bp_gap` of 2 would result in the following schedule of backpropagations:

```
time:   0 1 2 3 4 5 6 7 8
bp:           x   x   x
```

where each backprop goes back 4 time steps, resulting in an overlap of 2 time steps for each backprop.

`bool real_time`
: Checking this flag will cause the network to shift the activation buffers after each backpropagation, so that the appropriate amount of activation state information will be available for the next backpropagation (i.e., it shifts them by

the size of `bp_gap`). Not checking this flag will cause the `time_window` to be automatically set to the length of the current sequence.

14.2.5 RBp Sequence Processes and TimeEvents

The **RBpSequence** process is a special **SequenceProcess** that knows how to appropriately treat sequences of events that have time values associated with them. The classes **TimeEnvironment, TimeEvent_MGroup**, and **TimeEvent** together define an environment which consists of sequences (groups) of events where each event is specified to occur at a particular time. Furthermore, the environment defines certain simple forms of interpolation, which allows trajectories to be formed by specifying crucial points on the trajectory, but not everything in between. Also, the **RBpSequence** process uses the `end_time` of the **TimeEvent_MGroup** to set the `time_window` of the trial process, so that exactly one backprop phase will happen per sequence.

If a **TimeEnvironment** is not being used, a **RBpSequence** will simply run the sequence of events in each event group, one by one, through the trial process. This would make each event in the sequence appear at tick-wise intervals (i.e., every `dt`). In contrast, the **TimeEnvironment** based events have the benefit of making the environment invariant with respect to changes in `dt`, which can be very useful when `dt` is changed during training, etc.

A **TimeEnvironment**, aside from setting the default types of events and event groups to also be time-based ones, has a default interpolation parameter:

`Interpolate interpolate`
: The following forms of interpolation are defined:

  `PUNCTATE` Each event appears for the single slice of time that it has specified.

  `CONSTANT` Events persist with the same activations from the time on the event until the next event comes along.

  `LINEAR` Performs linear interpolation from one event to the next.

As with all sequence-based environments (see Section 11.3 [env-seq], page 169), a sequence of events is defined by putting all the events in a subgroup. **TimeEvent**s should be put in subgroups of type **TimeEvent_MGroup**, which is where the specific form of interpolation to be used for this particular sequence, and the total duration of the sequence, are specified:

`Interpolate interpolate`
: This is just like the interpolation variable on the environment, except it includes the `USE_ENVIRO` option, which uses whatever is set on the environment object. Thus, one can have different sequences use different kinds of interpolation, or they can defer to the environment.

`float end_time`
: The total duration of the sequence. It is automatically set to be as long as the latest event in the group, but you can set it to be longer to cause a `CONSTANT` interpolation to hold onto the event until `end_time` has been reached.

The time event object is just like a regular event except that it adds a time field, which specifies when this event is to first be presented to the network. How long it is presented depends on the interpolation scheme and how soon another event follows it.

14.2.6 Variations Available in RBp

NoisyRBpUnitSpec adds noise into the activation states of the RBp unit. The type and parameters of the noise are defined by the `noise` settings.

14.2.7 The Almeida-Pineda Algorithm in RBp

The Almeida-Pineda backprop (APBp) algorithm is a lot like the recurrent backpropagation algorithm just described, except that instead of recording the activation trajectory over time, and the backpropagating back through it, this algorithm performs activation propagation until the change in activation goes below some threshold, and then it performs backpropagation repeatedly until the change in error derivatives also goes below threshold.

This algorithm is implemented by using the standard RBp unit and connection types, even though APBp doesn't require the activation trace that is kept by these units. Indeed, you should set the `store_states` flag on the **RBpUnitSpec** to `false` when using APBp.

The only thing that is needed is a set of processes to implement the settling process over cycles of activation and error propagation. Thus, three new processes were implemented, including a cycle process (see Section 12.3.6 [proc-levels-cycle], page 195) to perform one cycle of activation or error propagation, a settle process (see Section 12.3.5 [proc-levels-settle], page 195) to iterate over cycles, and a train process (see Section 12.3.2 [proc-levels-train], page 191) to iterate over two phases of settling (activation and backpropagation).

The **APBpCycle** and **APBpSettle** processes don't have any user-settable parameters. The **APBpTrial** adds a couple of options to control settling:

`Counter phase_no`
: The counter that controls what phase the process is in.

`Phase phase`
: The phase, which is either `ACT_PHASE` or `BP_PHASE`. It is essentially just a more readable version of the phase_no counter.

`StateInit trial_init`
: Determines what to do at the start of settling. One can either `DO_NOTHING` or `INIT_STATE`, which initializes the unit activation state variables, and is the default thing to do.

`bool no_bp_stats`
: This flag, if set, does not collect any statistics during the Bp phase of settling.

`bool no_bp_test`
: This flag, if set, means that no backpropagation settling phase will be computed if the epoch process is in `TEST` `wt_update` mode.

The threshold that determines when the settling is cut off is determined by a **APBpMaxDa_De** statistic object, which measures the maximum change in activation or the maximum change in error derivative. The stopping criterion (see Section 12.5.2 [proc-stat-crit],

page 202) of this stat determines the cutoff threshold. It assumes that the same threshold is used for activation as is used for error, which seems to be reasonable in practice.

14.2.8 RBp Defaults

The following default files (see Section 9.5 [proj-defaults], page 126) are available for configuring different versions of the RBp objects:

'`rbp.def`' This is the standard defaults file.

'`rbp_ap.def`'
This is for doing Almeida-Pineda (see Section 14.2.7 [rbp-ap], page 241) in RBp.

14.2.9 RBp Implementation Details

An attempt was made to make the implementation of RBp very flexible, modular, and robust. Thus, units keep track of their own sense of time and record their own history of activations. This is done with a **CircBuffer** object, which is derived from an **Array** type (see Section 8.3 [obj-array], page 114). Values in this buffer can wrap-around, and can be shifted without performing any memory copying. Thus, the unit activation records can be quickly shifted after performing a backpropagation enough to make room for new states to be recorded. The trial process will shift the buffers just enough so that they will be full again the next time a backpropagation will occur (i.e., they are shifted by the `bp_gap` value converted into tick units).

However, the buffers are robust in the sense that if the bp_gap parameter is changed during processing, they will simply add new states and dynamically increase the size of the buffer if it is already full. Indeed, when a unit first starts processing, the buffer is automatically added to by the same mechanism–it is always full until some number of values have been shifted off the end.

The units only record their activation values in the buffers. Thus, there is a `StoreState` function which takes a snapshot of the current activation state. It is called at the end of the `Compute_Act` function. During backpropagation, the `StepBack` function is called, which will take one step back in time. The activation state recorded in the `prv_` version of the unit variables are copied into the current variables, and the new previous values are loaded from the buffers at the given tick.

The backpropagation loop looks as follows:

```
PerformBP(): {
  InitForBP();                  // clear current and prev error vals
  int buf_sz = GetUnitBufSize(); // get unit buffer size (states in units)
  for(i=buf_sz-2; i>=0; i--) { // loop backwards through unit states
    Compute_dEdA_dEdNet();      // backpropagate based on current state
    Compute_dWt();              // compute weight changes from error
    if(i > 0)                   // if not all the way back yet
      StepBack(i-1); // step back to previous time
  }
  RestoreState(buf_sz-1); // restore activations to end values
  if(real_time)
    ShiftBuffers(); // don't shift unless real time
}
```

Thus, error derivatives are computed on the current and `prv_` activation state variables, and then these are shifted backwards one step, and this continues for the entire length of stored activation values. The above routine is called by the trial process whenever the buffer size of the units is equal to or greater than the bp time window.

During backpropagation, the `prv_dEdA` and `prv_dEdNet` values are kept, and are used to time-average the computations of these values, much in the same way the activations or net inputs are time averaged during the activation computation phase.

The following is a chart describing the flow of processing in the RBp algorithm, starting with the epoch process, since higher levels do not interact with the details of particular algorithms, and assuming sequences are being used:

```
SequenceEpoch: {
  Init: {                              // at start of epoch
    environment->InitEvents();          // init events (if dynamic)
    event_list.Add() 0 to environment->GroupCount(); // get list of groups
    if(order == PERMUTE) event_list.Permute(); // permute list if necessary
    GetCurEvent();                      // get pointer to current group
  }
  Loop (trial): {                      // loop over trials
    SequenceProcess: {                 // sequence process (one sequence)
      Init: {                          // at start of sequence
        tick.max = cur_event_gp->EventCount(); // set max no of ticks
        event_list.Add() 0 to tick.max; // get list of events from group
        if(order == PERMUTE) event_list.Permute(); // permute if necessary
        GetCurEvent();                  // get pointer to current event
        InitNetState() {               // initialize net state at start
          if(sequence_init == INIT_STATE) network->InitState();
        }
      }
      Loop (tick): {                   // loop over ticks (sequence events)
        RBpTrial: {                    // trial process (one event)
          Init: {                      // at start of trial
            cur_event = epoch_proc->cur_event; // get event from sequence
          }
          Loop (once): {               // process this once per trial
            network->InitExterns();     // init external input to units
            cur_event->ApplyPatterns(network); // apply patterns from event
            if(unit buffer size == 0) { // units were just reset, time starting
              time = 0;                 // reset time
              StoreState();             // store initial state at t = 0
            }
            Compute_Act(): {           // compute acts (synchronous)
              network->Compute_Net();   // first get net inputs
              network->Compute_Act();   // then update acts based on nets
            }
            if(unit buffer size > time_win_ticks) // if act state buffers full
              PerformBP();              // backpropagate through states
            time += dt;                 // time has advanced..
          }
        }
        if(wt_update == ON_LINE) network->UpdateWeights(); // after trial
      }
    }
    if(wt_update == SMALL_BATCH)        // end of sequence
      network->UpdateWeights();         // update weights after sequence
    GetCurEvent();                      // get next event group
  }
  Final:                               // at end of epoch
    if(wt_update == BATCH)  network->UpdateWeights(); // batch mode updt
}
```

15 Constraint Satisfaction

Constraint satisfaction is an emergent computational property of neural networks that have recurrent connectivity, where activation states are mutually influenced by each other, and settling over time leads to states that satisfy the constraints built into the weights of the network, and those that impinge through external inputs.

Constraint satisfaction can solve complicated computational problems where the interdependencies among different possible solutions and high-dimensional state spaces make searching or other techniques computationally intractable. The extent to which a network is satisfying its constraints can be measured by a global energy or "goodness" function. Proofs regarding the stability of equilibrium states of these networks and derivations of learning rules have been made based on these energy functions.

In the PDP++ software, a collection of constraint satisfaction style algorithms have been implemented under a common framework. These algorithms include the binary and continuous Hopfield style networks *Hopfield, 1982, 1984*, the closely related Boltzmann Machine networks *Ackley, Hinton and Sejnowski, 1985*, the interactive activation and competition (IAC) algorithm *McClelland and Rumelhart, 1981*, and GRAIN networks *Movellan and McClelland, 1994*.

In addition to recurrent activation propagation and settling over time, these algorithms feature the important role that noise can play in avoiding sub-optimal activation states. The work with the GRAIN algorithm extends the role of noise by showing that the network can learn to settle into different distributions of activation states in a probabilistic manner. Thus, one can teach a network to go to one state roughly 70 percent of the time, and another state roughly 30 percent of the time. These distributions of possible target states can be specified by using a probability environment, which is described in a subsequent section.

Also, learning takes place in these networks through a more local form of learning rule than error backpropagation. This learning rule, developed for the Boltzmann machine, has been shown to work in a wide variety of activation frameworks, including deterministic networks. This rule can be described as a "contrastive Hebbian learning" (CHL) function, since it involves the subtraction of two simple Hebbian terms computed when the network is in two different "phases" of settling.

The two phases of settling required for learning are known as the *minus* (negative) and *plus* (positive) phase. The minus phase is the state of the network when only inputs are presented to the network. The plus phase is the state of the network when both inputs and desired outputs are presented. The CHL function states that the weights should be updated in proportion to the difference of the coproduct of the activations in the plus and minus phases:

```
cn->dwt = lrate * (ru->act_p * su->act_p - ru->act_m * su->act_m)
```

where `ru` is the receiving unit and `su` is the sending unit across the connection `cn`, and `act_m` is the activation in the minus phase, and `act_p` is the activation in the plus phase.

It turns out that in order to learn distributions of activation states, one needs to collect many samples of activation states in a stochastic network, and update the weights with the expected values of the coproducts of the activations, but the general idea is the same. This learning rule can be shown to be minimizing the cross-entropy between the distributions of

the activations in the minus and plus phases, which is the basis of the Boltzmann machine derivation of the learning rule.

The PDP++ implementation allows you to perform learning in both the stochastic mode, and with deterministic networks using the same basic code. Also, there is support for *annealing* and *sharpening* schedules, which adapt the noise and gain parameters (respectively) over the settling trajectory. Using these schedules can result in better avoidance of sub-optimal activation states.

15.1 Overview of the Cs Implementation

The Cs implementation defines a type of connection and connection spec that is used by all flavors of Cs networks. There is a basic CsUnit which is also common to all algorithms, but each different type of activation function defines its own version of the basic CsUnitSpec. Thus, to switch activation functions, one needs only to point one's units at a new unit spec type.

The new schedule process objects consist of three essential levels of processing, starting at the trial level and moving down through settling to the cycle, which implements one activation update of the network. Thus, the **CsTrial** process loops over the plus and minus phases of settling in the **CsSettle** process, which in turn iterates over cycles of the **CsCycle** process, which updates the activations of the units in the network. There is an optional level of processing which involves sampling over repeated settlings of the same pattern. This repeated sampling is used for learning to obtain reliable statistical estimates of the probabilities of certain activation states, and is thus used when learning to match a propability distribution over the output layer (see Section 15.7 [cs-prob-env], page 252). This **CsSample** process loops over samples of the **CsTrial** process.

There are several specialized statistic objects that compute the global goodness or energy function of the network (**CsGoodStat**), and record its probability of being in one of a number of desired activation states (**CsDistStat**), and the extent to which the probabilities of these states match their desired probabilities (**CsTIGstat**, **CsTargStat**).

15.2 Cs Connection Specifications

The Cs connection type contains a computed change in weight term `dwt`, a previous change in weight term `pdw`, and a change in weight term that is used in aggregating weight change statistics over time, `dwt_agg`. The previous change in weight term enables momentum-based learning to be performed.

The Cs connection specification type contains the following parameters:

`float lrate`
: Controls the rate of learning. It simply multiplies the CHL learning term.

`float momentum`
: The momentum, which includes a weighted average (weighted by the `momentum` parameter) of prior weight changes in the current weight change term.

`float decay`
: The rate of weight decay, if a weight decay function is being used.

`decay_fun`
: There are two weight decay functions defined by default, but others can be added. These two are `Cs_Simple_WtDecay`, and `Cs_WtElim_WtDecay`, which are just like their counterparts in the Bp algorithm, to which you are referred for more details, see Section 14.1.2 [bp-con], page 227.

15.3 Cs Unit Specifications

The **CsUnit** is fairly simple. As was described for backpropagation (see Section 14.1.3 [bp-unit], page 228) the bias weight is implemented as a connection object on the unit. It also keeps a record of the minus and plus phase activation values, `act_m` and `act_p` (primarily for display purposes), and the change in activation state for this unit, `da`.

All of the different flavors of constraint satisfaction algorithms differ only in their activation function. Thus, each one has a different type of unit spec. However, they all derive from a common **CsUnitSpec**, which has parameters shared by all the different algorithms. This unit spec has the parameters for noise and gain, the step size taken in updating activations, and schedules for adapting noise and gain over time:

`CsConSpec_SPtr bias_spec`
: Points the the connection spec that controls the learning of the bias weight on the unit.

`MinMaxRange real_range`
: The actual range to allow units to be in. Typically, units are kept within some tolerance of the absolute `act_range` values, in order to prevent saturation of the computation of inverse-sigmoid functions and other problems.

`Random noise`
: These are the parameters of the noise to be added to the unit. `GAUSSIAN` noise with zero mean is the standard form of noise to use. Noise is always added to the activations in an amount proportional to the square-root of the step size (except in the BoltzUnitSpec, where no noise is added).

`float step`
: The step size to take in updating activation values. A smaller step leads to smoother updating, but longer settling times.

`float gain`
: The sharpness of the sigmoidal activation function, or 1 over the temperature for the Boltzmann units. A higher gain makes the units act more like binary units, while a lower gain makes them act more continuous and graded.

`ClampType clamp_type`
: This controls the way in which external inputs (from the environment) are applied to the network. `HARD_CLAMP` means that the activation is exactly the `ext` value from the environment. `HARD_FAST_CLAMP` is like hard clamp, but optimized so that all of the inputs from clamped layers are computed once at the start of settling, saving considerable computational overhead. This should not be used if the inputs are noisy, since this noise will not be included! `SOFT_CLAMP` means that external inputs are added into the net input to a unit, instead

of forcing the activation value to take on the external value. `SOFT_THEN_HARD_CLAMP` performs soft clamping in the minus phase, and then hard clamping in the plus phase.

`float clamp_gain`
: When soft clamping, this parameter determines how strongly the external input contributes to the unit net input. It simply multiplies the value in the `ext` field.

`Random initial_act`
: Controls the random initialization of unit activations in the `InitState` function.

`bool use_annealing`
: Controls whether an annealing schedule is used to adapt the variance of the noise distribution over time (`noise_sched`).

`Schedule noise_sched`
: This schedule contains values which are multiplied times the `var` parameter of the `noise` field to get an effective variance level. The value from the schedule is the linear interpolation of the `cycle` count from the settle process based on points listed in the schedule. Thus, each point in the schedule gives a variance multiplier for a particular cycle count, and intermediate cycles yield interpolated multiplier values.

`bool use_sharp`
: Controls whether a sharpening schedule is used to adapt the gain parameter over time (`gain_sched`).

`Schedule gain_sched`
: This is a schedule for the gain multiplier. The effective gain is the `gain` parameter times the value from this schedule. The schedule works just like the `noise_sched` described above.

The basic **CsUnitSpec** uses the inverse-logistic activation function developed by *Movellan and McClelland, 1994*. Thus, the change in activation is a function of the difference between the actual net input, and the inverse logistic of the current activation value. This formulation ends up being an exact solution to the objective function used in their derivation.

The **SigmoidUnitSpec** uses a simple sigmoidal function of the net input, which is like the formulation of *Hopfield, 1984*, and is also the same as the RBp units described in Section 14.2.3 [rbp-unit], page 237. This type also has the option with the `time_avg` parameter of computing time averaging (i.e., as a function of the `step` parameter) on either the `ACTIVATION` or the `NET_INPUT`, as was the case with the RBp implementation. As was described there the `NET_INPUT` option allows units to settle faster.

The **BoltzUnitSpec** implements a binary activation function like that used Boltzmann machine and the network of *Hopfield, 1982*. Here, the unit takes on a 0 or 1 value probabilistically as a sigmoidal function of the net input. The gain of this sigmoid can also be represented by its inverse, which is known as temperature, by analogy with similar systems in statistical physics. Thus, we have a `temp` parameter which is used to update the gain parameter. Noise is intrinsic to this function, and is not added in any other way.

The **IACUnitSpec** implements the interactive activation and competition function. This requires two new parameters, `rest` and `decay`. If the net input to the unit is positive,

the activation is increased by `net * (max - act)`, and if it is negative it is decreased by `net * (act - min)`. In either case, the activation is also decayed towards the resting value by subtracting off a `decay * (act - rest)` term. IAC also has the option of only sending activation to other units when it is over some threshold (`send_thresh`). Doing this requires a different way of computing the net input to units, so it must be selected with the `use_send_thresh` flag, and by setting the `update_mode` in the **CsCycle** process to `SYNC_SENDER_BASED`. Pressing *ReInit* or *NewInit* at any level of process including and above the **CsTrial** process will check for the consistency of these settings, and prompt to change them.

The **LinearCsUnitSpec** computes activation as a simple linear function of the net input.

The **ThreshLinCsUnitSpec** computes activation as a threshold-linear function of the net input, where net input below threshold gives an activity of 0, and (net - threshold) above that.

15.4 Cs Proceses

The core set of Cs processes consist of a **CsTrial** process that performs the two phases of contrastive Hebbian learning (CHL), where each phase of settling is controlled by a **CsSettle** process, which in turn iterates over a number of individual **CsCycle** processing steps, each of which updates the activation state of the network. These processes fit nicely within the standard settling and cycle processes (see Section 12.3 [proc-levels], page 191).

In addition, the **CsSample** process can be used to sample over trials in order to obtain a better sampling of activation states for weight updating, which is usually only necessary when trying to match a probabalistic output distribution (see Section 15.7 [cs-prob-env], page 252). It is created by default if using the '`cs_prob_env.def`' defaults file, but not otherwise.

The **CsTrial** process iterates over two loops of settle processing, one for each phase. It has the following variables:

`Counter phase_no`
: The counter for this process, it goes from 1 to 2 for the two different phases.

`Phase phase`
: The phase the process is in, which is just a more readable version of the counter: `MINUS_PHASE` and `PLUS_PHASE`.

`StateInit trial_init`
: Indicates what to do at the start of each trial process. Typically, one wants to `INIT_STATE`, but it is also possible to `DO_NOTHING` or `MODIFY_STATE`, which could be redefined by a unit type to decay the activation state, for example (by default it just does an init state).

`bool no_plus_stats`
: This flag means that statistics will not be recorded in the plus phase. This is useful because squared error, for example, is only meaningful in the minus phase, since the correct answer is clamped in the plus phase.

`bool no_plus_test`
: This flag means that the plus phase will not be run if the epoch process indicates that it is in `TEST` mode (i.e., no learning is taking place).

The **CsSettle** process iterates over cycles of settling (activation updating). Like the `trial_init` of the trial process, the `between_phases` field of the settle process determines how the network state will be initialized between phases. Some people have claimed that learning works better if you do not `INIT_STATE` between phases (*Movellan, 1990*). However, the default is set to initialize the state, so that settling starts from the same initial conditions for both phases.

Also, the settle process allows the network to settle for some time before collecting statistics for learning. This time, determined by the `start_stats` parameter, allows the network to get into a stochastic equilibrium before measuring the activation states. If using a deterministic network, however, equilibrium is considered to simply be the final state of the network after settling. Setting the `deterministic` flag will only update the weights based on the final activation state.

The settle process will use the `time` field from a **TimeEvent** as a *duration* to settle on the event. Note that this is a non-standard use of this time field (see Section 14.2.5 [rbp-seq], page 240).

The **CsCycle** process updates the activation state of the network. This can be done in one of two ways — either all of the units are updated simultaneously (`SYNCHRONOUS` mode), or units are selected at random and updated one at a time, which is `ASYNCHRONOUS` mode. If the asynchronous mode is selected, one can perform multiple asynchronous updates per cycle. The number of units updated is determined by the `n_updates` variable. Note that the total number of updates performed is the product of `n_updates` and the number of cycles made during settling, so be sure to take that into account when setting the values of either of these parameters.

A variation of the synchronous mode is the `SYNC_SENDER_BASED` mode, which must be selected if IACUnitSpec's are being used, and the `use_send_thresh` flag is set. In this case, net input is computed in a sender-based fashion, meaning that the net input for each unit must first be initialized to zero, then accumulated. In standard receiver-based net input computation, the net input can be reset at the point an individual unit's net input is computed, so it doesn't have to be done for the entire network first. This is the different between SYNC_SENDER_BASED and SYNCHRONOUS. Asynchronous updating is not supported for sender-based updating. When the **CsTrial** process is *ReInit*'d or *NewInit*'d, it will automatically check for the correct pairing of the use_send_thresh and update_mode flags, and inform you if it needs to be changed.

The **CsSample** process has `sample` counter to keep track of the number of samples run. Set the `max` of this counter to 1 if you are running a deterministic network, or are not doing learning on probability distributions. This process iterates over the trial process, and only needs to be present when learning probabalistic output distributions.

15.5 Cs Statistics

There are several statistics which are specific to the Cs package, including one that compute the global "goodness" (aka energy) of the network's activation state (**CsGoodStat**), another set of statistics for measuring the probabilities of different activation states (**CsDistStat, CsTIGstat, CsTargStat**), and a statistic that can be used to control the length of settling based on the amount of activation change (**CsMaxDa**).

15.5.1 The Goodness (Energy) Statistic

The **CsGoodStat** computes the overall goodness of the activation state, which is composed of two terms, the *harmony* and *stress*. Harmony reflects the extent to which the activations satisfy the constraints imposed by the weights. It is just a sum over all units of the product of the activations times the weights:

$$H = \sum_j \sum_i a_j w_{ij} a_i$$

The stress term reflects the extent to which unit's activations are "stressed" at their extreme values. It is just the inverse sigmoidal function of the unit activation values:

$$S = \sum_j f^{-1}(a_j)$$

The total goodness is just the harmony minus the stress. These values are stored in the `hrmny`, `strss`, and `gdnss` stat value members of the goodness stat. Also, there is an option (`netin_hrmny`) to use the net input to a unit in computing the harmony term, since harmony is just the unit's activation times its net input. This should be used whenever it is safe to assume that the net input reflects the current activation states (i.e., most of the time). The example project in '`demo/cs/figgr.proj.gz`' shows how the goodness function measures the quality of the constraint satisfaction over time in a large-scale constraint satisfaction problem.

15.5.2 Statistics for Measuring Probability Distributions

The **CsDistStat** is used to measure the percentage of time (i.e., the probability) that the units in the network which have target patterns in the environment spend in any of the possible target patterns. This is used when there are multiple possible target states defined for any given event (see Section 15.7 [cs-prob-env], page 252), which means that a simple squared-error comparison against any one of these would be relatively meaningless — one wants to know how much time is spent in each of the possible states. The dist stat generates one column of data for each possible target pattern, and each column represents the probability (proportion of time) that the network's output units were within some distance of the target pattern. The tolerance of the distance measure is set with the `tolerance` parameter, which is the absolute distance between target and actual activation that will still result in a zero distance measure. A network is considered to be "in" a particular target state whenever its total distance measure is zero, so this tolerance should be relatively generous (e.g., .5 so units have to be on the right side of .5).

The **CsTIGstat** is essentially a way of aggregating the columns of data produced by the **CsDistStat**. It is automatically created by the dist stat's `CreateAggregates` function (see Section 12.5 [proc-stat], page 199) at the level of the **CsSample** process (note that unlike other aggregators, it is in the `final_stats` group of the sample process, and it feeds off of the aggregator of the dist stat in the `loop_stats` of the same process). The TIG stat measures the total information gain (aka cross-entropy) of the probability distribution of target states observed in the network (as collected by the dist stat pointed to by the

`dist_stat` member), and the target probability distribution as specified in the probability patterns themselves (see Section 15.7 [cs-prob-env], page 252). This measure is zero if the two distributions are identical, and it goes up as the match gets worse. It essentially provides a distance metric over probability distributions.

The **CsTargStat**, like the TIG stat, provides a way of aggregating the distribution information obtained by the dist stat. This should be created in the sample `final_stats` group (just like the TIG stat), and its `dist_stat` pointer set to the aggregator of the dist stat in the sample process `loop_stats`. This stat simply records the sum of each column of probability data, which provides a measure of how often the network is settling into one of the target states, as opposed to just flailing about in other random states.

15.5.3 Measuring the Maximum Delta-Activation

The **CsMaxDa** statistic computes the maximum delta-activation (change in activation) for any unit in the network. This is used to stop settling once the network has reached equilibrium. The stopping criterion for this statistic is the tolerance with which equilibrium is measured. It is created by default if the `deterministic` flag is checked in the **CsSettle** process when it is created, which can be done by using the '`cs_det.def`' defaults (det = deterministic). This stat typically goes in the settle process `loop_stats`.

15.6 Cs Defaults

The following default files (see Section 9.5 [proj-defaults], page 126) are available for configuring different versions of the Cs objects:

'`cs.def`' This is the standard defaults file, which uses a SigmoidUnitSpec, and the standard process heirarchy, assuming stochastic processing.

'`cs_det.def`'
This is for deterministic Cs networks, which means that only one sample is made of activation states for computing weight changes at the end of settling, and the CsMaxDa settling stat is created by default.

'`cs_prob_env.def`'
This configures the probabalistic output patterns version of Cs, meaning that the CsSettle process is created, and the environment will create probablistic events (see Section 15.7 [cs-prob-env], page 252).

15.7 The Probability Environment and Cs

An environment can be specified where each event has multiple possible target patterns, each of which has an associated probability of occurring. Thus, when an event is presented to the network, one out of a set of possible target patterns is chosen according to their relative probabilities. These probabilistic distributions of target states can actually be learned in a stochastic constraint satisfaction network.

Probabilities can be associated with different patterns by creating **ProbPattern** objects instead of the usual **Pattern** ones. The prob pattern has a `prob` field, which contains its probability of occurrence.

However, in order for these probabilities to be used in the presentation of patterns, one needs to use a **ProbEventSpec**, which has special code that chooses which pattern to present based on its probability. Not all patterns in an event need to be probabilistic. Indeed, the usual setup is to have a deterministic input pattern, and then a set of possible outputs that follow from this one input. The determiniation of which patterns are probabilistic and which are deterministic is made by putting all of the mutually-exclusive probabilistic alternative patterns in a **ProbPatternSpec_Group**.

Thus, one needs to make a sub-group in the `patterns` group on the **ProbEventSpec**, and this sub-group has to be of type **ProbPatternSpec_Group**. Then, regular **PatternSpec** objects, one for each alternative target pattern, should be created in the sub group. Make sure to set the `pattern_type` field of these pattern specs to be **ProbPattern**, so that the patterns which are created in the event will have probabilities associated with them.

When events are actually created from a prob event spec, one needs to edit the patterns within the sub-group and assign each of them probabilities that sum to 1 for all the patterns in the group. Thus, the network will be certain of choosing one of them to present.

15.8 Cs Implementational Details

Note that the weight change operation in Cs is viewed as the process of collecting statistics about the coproducts of activations across the weights. Thus, there is a new function at the level of the **CsConSpec** called `Aggregate_dWt`, which increments the `dwt_agg` value of the connection by the phase (+1.0 for plus phase, -1.0 for the minus phase) times the coproduct of the activations. This function is called repeatedly if learning is taking place after the `start_stats` number of cycles has taken place.

The standard `Compute_dWt` function is then called at the end of the sample process, and it divides the aggregated weight change value by a count of the number of times it was aggregated, so that the result is an expected value measure.

Also, note that the type of momentum used in Cs corresponds to the `BEFORE_LRATE` option of backpropagation (see Section 14.1.2 [bp-con], page 227).

The following is a chart describing the flow of processing in the Cs algorithm, starting with the epoch process, since higher levels do not interact with the details of particular algorithms:

```
EpochProcess: {
  Init: {                                // at start of epoch
    environment->InitEvents();           // init events (if dynamic)
    event_list.Add() 0 to environment->EventCount(); // get list of events
    if(order == PERMUTE) event_list.Permute();       // permute if necessary
    GetCurEvent();                       // get pointer to current event
  }
  Loop (sample): {                       // loop over samples
    CsSample: {                          // sample process (one event)
      Init: {                            // at start of sample
        cur_event = epoch_proc->cur_event; // get cur event from epoch
      }
      Loop (sample): {                   // loop over samples of event
        CsTrial: {                       // phase process (one phase)
          Init: {                        // at start of phase
            phase = MINUS_PHASE;         // start in minus phase
            if(phase_init == INIT_STATE)  network->InitState();
            cur_event = epoch_proc->cur_event;  // get cur event from epoch
            cur_event->ApplyPatterns(network);  // apply patterns to network
          }
          Loop (phase_no [0 to 1]): {  // loop over phases
            CsSettle: {                  // settle process (one settle)
              Init: {                    // at start of settle
                if(CsPhase::phase == PLUS_PHASE) {
                  network->InitState(); // init state (including ext input)
                  cur_event->ApplyPatterns(network);  // re-apply patterns
                  Targ_To_Ext();         // turn targets into external inputs
                }
              }
              Loop (cycle): {            // loop over act update cycles
                CsCycle: {               // cycle process (one cycle)
                  Loop (once): {         // only process this once per cycle
                    Compute_SyncAct() or Compute_AsyncAct(); // see below
                    if(!deterministic and cycle > start_stats)
                      Aggregate_dWt();  // aggregate wt changes
                  }
                }
              }
              Final: {                   // at end of phase
                if(deterministic) Aggregate_dWt();  // do at end
                PostSettle();            // copy act to act_p or act_m
              }
            }
            phase = PLUS_PHASE;          // change to plus phase after minus
          }
        }
      }
      Final: {                           // at end of sample (done with event)
        network->Compute_dWt();          // compute wt changes based on aggs
      }
    }
    if(wt_update == ON_LINE or wt_update == SMALL_BATCH and trial.val % batch_n)
      network->UpdateWeights();          // update weights after sample if necc
    GetCurEvent();                       // get next event to present
  }
  Final:                                 // at end of epoch
```

The activation computing functions are broken down as follows:

```
Compute_SyncAct(): {           // compute synchronous activations
  network->InitDelta();         // initialize net-input terms
  network->Compute_Net();       // aggregate net inputs

  network->layers: {           // loop over layers
    units->Compute_Act(CsSettle::cycle); // compute act from net
  }                            // (cycle used for annealing, sharpening)
}

Compute_AsyncAct(): {          // compute asynchronous activations
  for(i=0...n_updates) {       // do this n_updates times per cycle
    rnd_num = Random between 0 and CsSettle::n_units;  // pick a random unit
    network->layers: {         // loop over layers
      if(layer contains rnd_num unit) { // find layer containing unit
        unit = layer->unit[rnd_num];    // get unit from layer
        unit->InitDelta();              // initialize net input terms
        unit->Compute_Net();            // aggregate net inputs
        unit->Compute_Act(CsSettle::cycle);  // compute act from net
      }                        // (cycle used for annealing, sharpening)
    }
  }
}
```

16 Self-organizing Learning

The defining property of self-organizing learning is that it operates without requiring an explicit training signal from the environment. This can be contrasted with error back-propagation, which requires target patterns to compare against the output states in order to generate an error signal. Thus, many people regard self-organizing learning as more biologically or psychologically plausible, since it is often difficult to imagine where the explicit training signals necessary for error-driven learning come from. Further, there is some evidence that neurons actually use something like a Hebbian learning rule, which is commonly used in self-organizing learning algorithms.

There are many different flavors of self-organizing learning. Indeed, one of the main differences between self-organizing algorithms and error-driven learning is that they need to make more assumptions about what good representations should be like, since they do not have explicit error signals telling them what to do. Thus, different self-organizing learning algorithms make different assumptions about the environment and how best to represent it.

One assumption that is common to many self-organizing learning algorithms is that events in the environment can be *clustered* together according to their "similarity." Thus, learning amounts to trying to find the right cluster in which to represent a given event. this is often done by enforcing a competition between a set of units, each of which represents a different cluster. The *competitive learning* algorithm (CL) of *Rumelhart and Zipser, 1985* is a classic example of this form of learning, where the single unit which is most activated by the current input is chosen as the "winner" and therefore gets to adapt its weights in response to this input pattern.

The PDP++ implementation of self-organizing learning, called *So*, includes competitive learning and several variations of it, including "soft" competitive learning *Nowlan, 1990*, which replaces the "hard" competition of standard competitive learning with a more graded activation function. Also included are a couple of different types of modified Hebbian learning rules that can be used with either hard or soft activation functions.

An additional assumption that can be made about the environment is that there is some kind of *topology* or ordered relationship among the different clusters. This notion is captured in the *self-organizing map* (SOM) algorithm of *Kohonen, 1989; 1990; 1995*. This algorithm adds to the basic idea of competition among the units that represent a cluster the additional assumption that units which are nearby in 2-D space should represent clusters that are somehow related. This spatial-relatedness constraint is imposed by allowing nearby units to learn a little bit when one of their neighbors wins the competition. This algorithm is also implemented in the So package.

The directory '`demo/so`' contains two projects which demonstrate the use of both the competitive-learning style algorithms, and the self-organizing maps.

16.1 Overview of the So Implementation

The So implementation is designed to be used in a mix-and-match fashion. Thus, there are a number of different learning algorithms, and several different activation functions, each of which can be used with the other. The learning algorithms are implemented as

different connection specs derived from a basic **SoConSpec** type. They all use the same **SoCon** connection type object.

Unlike the other algorithms in the PDP++ distribution (Bp and Cs), the So implementation uses **LayerSpec** objects extensively. These layer specifications implement competition among units in the same layer, which is central to the self-organizing algorithms. Thus, there are three different layer specs all derived from a common **SoLayerSpec** which implement hard competitive learning (**Cl**), soft competitive learning (**SoftCl**), and the self-organizing map (**Som**) activation functions.

There are no new statistics defined for self-organizing learning, and only one process object, which performs a simple feed-forward processing trial (all of the So algorithms are feed-forward).

16.2 So Connection Specifications

The basic connection type used in all the algorithms, **SoCon** has a delta-weight variable `dwt` and a previous-delta-weight variable `pdw`. `dwt` is incremented by the current weight change computations, and then cleared when the weights are updated. `pdw` should be used for viewing, since `dwt` of often zero. While it has not been implemented in the standard distribution, `pdw` could be used for momentum-based updating (see Section 14.1.2 [bp-con], page 227).

The basic **SoConSpec** has a learning rate parameter `lrate`, and a range to keep the weight values in: `wt_range`. Unlike error-driven learning, many self-organizing learning algorithms require the weights to be forcibly bounded, since the positive-feedback loop phenomenon of associative learning can lead to infinite weight growth otherwise. Finally, there is a variable which determines how to compute the average and summed activation of the input layer(s), which is needed for some of the learning rules. If the network is fully connected, then one can set `avg_act_source` to compute from the `LAYER_AVG_ACT`, which does not require any further computation. However, if the units receive connections from only a sub-sample of the input layer, then the layer average might not correspond to that which is actually seen by individual units, so you might want to use `COMPUTE_AVG_ACT`, even though it is more computationally expensive.

The different varieties of **SoConSpec** are as follows:

HebbConSpec

This computes the most basic Hebbian learning rule, which is just the coproduct of the sending and receiving unit activations:

```
cn->dwt += ru->act * su->act;
```

Though it is perhaps the simplest and clearest associative learning rule, its limitations are many, including the fact that the weights will typically grow without bound. Also, for any weight decrease to take place, it is essential that activations be able to take on negative values. Keep this in mind when using this form of learning. One application of this con spec is for simple pattern association, where both the input and output patterns are determined by the environment, and learning occurs between these patterns.

ClConSpec

This implements the standard competitive learning algorithm as described in *Rumelhart & Zipser, 1985*. This rule can be seen as attempting to align the weight vector of a given unit with the center of the cluster of input activation vectors that the unit responds to. Thus, each learning trial simply moves the weights some amount towards the input activations. In standard competitive learning, the vector of input activations is *normalized* by dividing by the sum of the input activations for the input layer, `sum_in_act` (see `avg_act_source` above for details on how this is computed).

```
cn->dwt += ru->act * ((su->act / cg->sum_in_act) - cn->wt);
```

The amount of learning is "gated" by the receiving unit's activation, which is determined by the competitive learning function. In the winner-take-all "hard" competition used in standard competitive learning, this means that only the winning unit gets to learn. Note that if you multiply through in the above equation, it is equivalent to a Hebbian-like term minus something that looks like weight decay:

```
cn->dwt += (ru->act * (su->act / cg->sum_in_act)) - (ru->act * cn->wt);
```

This solves both the weight bounding and the weight decrease problems with pure Hebbian learning as implemented in the **HebbConSpec** described above.

SoftClConSpec

This implements the "soft" version of the competitive learning learning rule *Nowlan, 1990*. This is essentially the same as the "hard" version, except that it does not normalize the input activations. Thus, the weights move towards the center of the actual activation vector. This can be thought of in terms of maximizing the value of a multi-dimensional Gaussian function of the distance between the weight vector and the activation vector, which is the form of the learning rule used in soft competitive learning. The smaller the distance between the weight and activation vectors, the greater the activation value.

```
cn->dwt += ru->act * (su->act - cn->wt);
```

This is also the form of learning used in the self-organizing map algorithm, which also seeks to minimize the distance between the weight and activation vectors. The receiving activation value again gates the weight change. In soft competitive learning, this activation is determined by a soft competition among the units. In the SOM, the activation is a function of the activation kernel centered around the unit with the smallest distance between the weight and activation vectors.

ZshConSpec

This implements the "zero-sum" Hebbian learning algorithm (ZSH) *O'Reilly & McClelland, 1992*, which implements a form of *subtractive* weight constraints, as opposed to the *multiplicative* constraints used in competitive learning. Multiplicative constraints work to keep the weight vector from growing without bound by maintaining the length of the weight vector normalized to that of the activation vector. This normalization preserves the ratios of the relative correlations of the input units with the cluster represented by a given unit. In contrast, the subtractive weight constraints in ZSH exaggerate the weights

to those inputs which are greater than the average input activation level, and diminish those to inputs which are below average:

```
cn->dwt += ru->act * (su->act - cg->avg_in_act);
```

where `avg_in_act` is the average input activation level. Thus, those inputs which are above average have their weights increased, and those which are below average have them decreased. This causes the weights to go into a corner of the hypercube of weight values (i.e., weights tend to be either 0 or 1). Because weights are going towards the extremes in ZSH, it is useful to introduce a "soft" weight bounding which causes the weights to approach the bounds set by `wt_range` in an exponential-approach fashion. If the weight change is greater than zero, then it is multiplied by '`wt_range.max - cn->wt`', and if it is less than zero, it is multiplied by '`cn->wt - wt_range.min`'. This is selected by using the `soft_wt_bound` option.

MaxInConSpec

This learning rule is basically just the combination of SoftCl and Zsh. It turns out that both of these rules can be derived from an objective function which seeks to maximize the input information a unit receives, which is defined as the signal-to-noise ratio of the unit's response to a given input signal *O'Reilly, 1994*. The formal derivation is based on a different kind of activation function than those implemented here, and it has a special term which weights the Zsh-like term according to how well the signal is already being separated from the noise. Thus, this implementation is simpler, and it just combines Zsh and SoftCl in an additive way:

```
cn->dwt += ru->act * (su->act - cg->avg_in_act) +
           k_scl * ru->act * (su->act - cn->wt);
```

Note that the parameter `k_scl` can be adjusted to control the influence of the SoftCl term. Also, the `soft_wt_bound` option applies here as well.

16.3 So Unit and Layer Specifications

Activity of units in the So implementation is determined jointly by the unit specifications and the layer specifications. The unit specifications determine how each unit individually computes its net input and activation, while the layer specifications determine the actual activation of the unit based on a competition that occurs between all the units in a layer.

All So algorithms use the same unit type, **SoUnit**. The only thing this type adds to the basic **Unit** is the `act_i` value, which reflects the "independent" activation of the unit prior to any modifications that the layer-level competition has on the final activations. This is primarily useful for the soft Cl units, which actually transform the net input term with a Gaussian activation function, the parameters of which can be tuned by viewing the resulting `act_i` values that they produce.

There are three basic types of unit specifications, two of which derive from a common **SoUnitSpec**. The **SoUnitSpec** does a very simple linear activation function of the net input to the unit. It can be used for standard competitive learning, or for Hebbian learning on linear units.

The **SoftClUnitSpec** computes a Gaussian function of the distance between the weight and activation vectors. The variance of the Gaussian is given by the `var` parameter, which is not adapting and shared by all weights in the standard implementation, resulting in a fixed spherical Gaussian function. Note that the `net` variable on units using this spec is the distance measure, not the standard dot product of the weights and activations.

The **SomUnitSpec** simply computes a sum-of-squares distance function of the activations and weights, like the **SoftClUnitSpec**, but it does not apply a Gaussian to this distance. The winning unit in the SOM formalism is the one with the weight vector closest to the current input activation state, so this unit provides the appropriate information for the layer specification to choose the winner.

There are three algorithm-specific types of layer specifications, corresponding to the Cl, SoftCl, and SOM algorithms, and the parent **SoLayerSpec** type which simply lets the units themselves determine their own activity. Thus, the **SoLayerSpec** can be used when one does not want any competition imposed amongst the units in a layer. This can be useful in the case where both layers are clamped with external inputs, and the task is to perform simple pattern association using Hebbian learning. Note that all layer specs do not impose a competition when they are receiving external input from the environment.

There is one parameter on the **SoLayerSpec** which is used by the different algorithms to determine how to pick the winning unit. If `netin_type` is set to `MAX_NETIN_WINS`, then the unit with the maximum net input value wins. This is appropriate if the net input is a standard dot-product of the activations times the weights (i.e., for standard competitive learning). If it is `MIN_NETIN_WINS`, then the unit with the minimal net input wins, which is appropriate when this is a measure of the distance between the weights and the activations, as in the SOM algorithm. Note that soft competitive learning does not explicitly select a winner, so this field does not matter for that algorithm.

The **ClLayerSpec** selects the winning unit (based on `netin_type`), and assigns it an activation value of 1, and it assigns all other units a value of 0. Thus, only the winning unit gets to learn about the current input pattern. This is a "hard" winner-take-all competition.

The **SoftClLayerSpec** does not explicitly select a winning unit. Instead, it assigns each unit an activation value based on a *Soft Max* function:

$$a_j = \frac{e^{g_j}}{\sum_k e^{g_k}}$$

Where *g_j* is the Gaussian function of the distance between the unit's weights and activations (stored in `act_i` on the **SoUnit** object). Thus, the total activation in a layer is normalized to add up to 1 by dividing through by the sum over the layer. The exponential function serves to magnify the differences between units. There is an additional `softmax_gain` parameter which multiplies the Gaussian terms before they are put through the exponential function, which can be used to sharpen the differences between units even further.

Note that **SoftClLayerSpec** can be used with units using the **SoUnitSpec** to obtain a "plain" SoftMax function of the dot product net input to a unit.

Finally, the **SomLayerSpec** provides a means of generating a "neighborhood kernel" of activation surrounding the winning unit in a layer. First, the unit whose weights are closest

to the current input pattern is selected (assuming the **SomUnitSpec** is being used, and the `netin_type` is set to `MIN_NETIN_WINS`). Then the neighbors of this unit are activated according to the `neighborhood` kernel defined on the spec. The fact that neighboring units get partially activated is what leads to the development of topological "map" structure in the network.

The shape and weighting of the neighborhood kernel is defined by a list of **NeighborEl** objects contained in the `neighborhood` member. Each of these defines one element of the kernel in terms of the offset in 2-D coordinates from the winning unit (`off`), and the activation value for a unit in this position (`act_val`). While these can be created by hand, it is easier to use one of the following built-in functions on the **SomLayerSpec**:

`KernelEllipse(int half_width, int half_height, int ctr_x, int ctr_y)`
: This makes a set of kernel elements in the shape of an ellipse with the given dimensions and center (typically 0,0).

`KernelRectangle(int width, int height, int ctr_x, int ctr_y)`
: This makes a rectangular kernel with the given dimensions and center.

`KernelFromNetView(NetView* view)`
: This makes a kernel based on the currently selected units in the NetView (see Section 10.6 [net-view], page 149). Select the center of the kernel first, followed by the other elements. Then call this function.

`StepKernelActs(float val)`
: This assigns the `act_val` values of the existing kernel elements to be all the same value, `val`.

`LinearKernelActs(float scale)`
: This assigns the `act_val` values of the existing kernel elements as a linear function of their distance from the center, scaled by the given scale parameter.

`GaussianKernelActs(float scale, float sigma)`
: This assigns the `act_val` values of the existing kernel elements as a Gaussian function of their distance from the center, scaled by the given scale parameter, where the Gaussian has a variance of `sigma`.

One can see the resulting kernel function by testing the network and viewing activations. Also, there is a demo project that illustrates how to set up a SOM network in '`demo/so/som.proj.gz`'.

16.4 The So Trial Process

The only process defined for the self-organizing algorithms is a trial process which simply performs a feed-forward update of the unit's net input and activations. Thus, the layers in the network must be ordered in the order of their dependencies, with later layers depending on input from earlier ones.

16.5 So Implementational Details

The following is a chart describing the flow of processing in the So algorithm, starting with the epoch process, since higher levels do not interact with the details of particular algorithms:

```
EpochProcess: {
  Init: {
    environment->InitEvents();          // init events (if dynamic)
    event_list.Add() 0 to environment->EventCount(); // get list of events
    if(order == PERMUTE) event_list.Permute();       // permute if necessary
    GetCurEvent();                      // get pointer to current event
  }
  Loop (trial): {                       // loop over trials
    SoTrial: {                          // trial process (one event)
      Init: {                           // at start of trial
        cur_event = epoch_proc->cur_event; // get cur event from epoch
      }
      Loop (once): {                    // only process this once per trial
        network->InitExterns();         // init external inputs to units
        cur_event->ApplyPatterns(network); // apply patterns to network
        Compute_Act(): {                // compute the activations
          network->layers: {            // loop over layers
            layer->Compute_Net();       // compute net inputs
            layer->Compute_Act();       // compute activations from net in
          }
        }
        network->Compute_dWt();         // compute weight changes from acts
      }
    }
    if(wt_update == ON_LINE or wt_update == SMALL_BATCH and trial.val % batch_n)
      network->UpdateWeights(); // after trial, update weights if necc
    GetCurEvent();              // get next event
  }
  Final:
    if(wt_update == BATCH)  network->UpdateWeights(); // batch weight updates
}
```

The `layer->Compute_Act()` function has several sub-stages for different versions of algorithms, as detailed below:

For non-input layer hard competitive learning units:

```
ClLayerSpec::Compute_Act() {
  SoUnit* win_u = FindWinner(lay);
  float lvcnt = (float)lay->units.leaves;
  lay->avg_act =  // compute avg act assuming one winner and rest losers..
        (act_range.max / lvcnt) + (((lvcnt - 1) * act_range.min) / lvcnt);
  win_u->act = act_range.max; // winning unit gets max value
  win_u->act_i = lay->avg_act; // and average value goes in _i
}
```

For non-input layer soft competitive learning units:

```
SoftClLayerSpec::Compute_Act() {
  float sum = 0;
  for(units) {                 // iterate over the units
    unit->Compute_Act();                          // compute based on netin
    unit->act = exp(softmax_gain * unit->act);  // then exponential
    sum += unit->act;                             // collect sum
  }
  for(units) {                 // then make a second pass
    unit->act =                 // act is now normalized by sum
        act_range.min + act_range.range * (unit->act / sum);
  }
  Compute_AvgAct();             // then compute average act over layer
}
```

The code for the SOM case is more complicated than the description, which is just that it finds the winner and pastes the kernel onto the units surrounding the winner.

17 Leabra

Leabra stands for "Local, Error-driven and Associative, Biologically Realistic Algorithm", and it implements a balance between Hebbian and error-driven learning on top of a biologically-based point-neuron activation function with inhibitory competition dynamics (either via inhibitory interneurons or a fast k-Winners-Take-All approximation thereof). Extensive documentation is available from the book: "Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain", O'Reilly and Munakata, 2000, Cambridge, MA: MIT Press. For more information, see the website: `http://psych.colorado.edu/~oreilly/comp_ex_cog_neuro.html`.

Hebbian learning is performed using conditional principal components analysis (CPCA) algorithm with correction factor for sparse expected activity levels.

Error driven learning is performed using GeneRec, which is a generalization of the Recirculation algorithm, and approximates Almeida-Pineda recurrent backprop. The symmetric, midpoint version of GeneRec is used, which is equivalent to the contrastive Hebbian learning algorithm (CHL). See *O'Reilly (1996; Neural Computation)* for more details.

The activation function is a point-neuron approximation with both discrete spiking and continuous rate-code output.

Layer or unit-group level inhibition can be computed directly using a k-winners-take-all (KWTA) function, producing sparse distributed representations, or via inihibitory interneurons.

The net input is computed as an average, not a sum, over connections, based on normalized, sigmoidaly transformed weight values, which are subject to scaling on a connection-group level to alter relative contributions. Automatic scaling is performed to compensate for differences in expected activity level in the different projections.

Weights are subject to a contrast enhancement function, which compensates for the soft (exponential) weight bounding that keeps weights within the normalized 0-1 range. Contrast enhancement is important for enhancing the selectivity of self-organizing learning, and generally results in faster learning with better overall results. Learning operates on the underlying internal linear weight value, which is computed from the nonlinear (sigmoidal) weight value prior to making weight changes, and is then converted back. The linear weight is always stored as a negative value, so that shared weights or multiple weight updates do not try to linearize the already-linear value. The learning rules have been updated to assume that wt is negative (and linear).

There are various extensions to the algorithm that implement things like reinforcement learning (temporal differences), simple recurrent network (SRN) context layers, and combinations thereof (including an experimental versions of a complex temporal learning mechanism based on the prefrontal cortex and basal ganglia). Other extensions include a variety of options for the activation and inhibition functions, self regulation (accommodation and hysteresis, and activity regulation for preventing overactive and underactive units), synaptic depression, and various optional learning mechanisms and means of adapting parameters. These features are off by default but do appear in some of the edit dialogs — any change from default parameters should be evident in the edit dialogs.

17.1 Overview of the Leabra Algorithm

The pseudocode for Leabra is given here, showing exactly how the pieces of the algorithm described in more detail in the subsequent sections fit together.

```
Iterate over minus and plus phases of settling for each event.
  o At start of settling, for all units:
    - Initialize all state variables (activation, v_m, etc).
    - Apply external patterns (clamp input in minus, input & output in
      plus).
    - Compute net input scaling terms (constants, computed
      here so network can be dynamically altered).
    - Optimization: compute net input once from all static activations
      (e.g., hard-clamped external inputs).
  o During each cycle of settling, for all non-clamped units:
    - Compute excitatory netinput (g_e(t), aka eta_j or net)
       -- sender-based optimization by ignoring inactives.
    - Compute kWTA inhibition for each layer, based on g_i^Q:
      * Sort units into two groups based on g_i^Q: top k and
        remaining k+1 -> n.
      * If basic, find k and k+1th highest
        If avg-based, compute avg of 1 -> k & k+1 -> n.
      * Set inhibitory conductance g_i from g^Q_k and g^Q_k+1
    - Compute point-neuron activation combining excitatory input and
      inhibition
  o After settling, for all units, record final settling activations
    as either minus or plus phase (y^-_j or y^+_j).
After both phases update the weights (based on linear current
    weight values), for all connections:
  o Compute error-driven weight changes with soft weight bounding
  o Compute Hebbian weight changes from plus-phase activations
  o Compute net weight change as weighted sum of error-driven and Hebbian
  o Increment the weights according to net weight change.
```

17.1.1 Point Neuron Activation Function

```
Default parameter values:

Parameter | Value | Parameter | Value
-----------------------------------------
E_l       | 0.15  | gbar_l    | 0.10
E_i       | 0.15  | gbar_i    | 1.0
E_e       | 1.00  | gbar_e    | 1.0
V_rest    | 0.15  | Theta     | 0.25
tau       | .02   | gamma     | 600
k_hebb    | .02   | epsilon   | .01
```

Leabra uses a *point neuron* activation function that models the electrophysiological properties of real neurons, while simplifying their geometry to a single point. This function is nearly as simple computationally as the standard sigmoidal activation function, but the more biologically-based implementation makes it considerably easier to model inhibitory

competition, as described below. Further, using this function enables cognitive models to be more easily related to more physiologically detailed simulations, thereby facilitating bridge-building between biology and cognition.

The membrane potential `V_m` is updated as a function of ionic conductances `g` with reversal (driving) potentials `E` as follows:

$$\Delta V_m(t) = \tau \sum_c g_c(t)\overline{g_c}(E_c - V_m(t))$$

with 3 channels (c) corresponding to: e excitatory input; l leak current; and i inhibitory input. Following electrophysiological convention, the overall conductance is decomposed into a time-varying component g_c(t) computed as a function of the dynamic state of the network, and a constant gbar_c that controls the relative influence of the different conductances. The equilibrium potential can be written in a simplified form by setting the excitatory driving potential (E_e) to 1 and the leak and inhibitory driving potentials (E_l and E_i) of 0:

$$V_m^\infty = \frac{g_e\overline{g_e}}{g_e\overline{g_e} + g_l\overline{g_l} + g_i\overline{g_i}}$$

which shows that the neuron is computing a balance between excitation and the opposing forces of leak and inhibition. This equilibrium form of the equation can be understood in terms of a Bayesian decision making framework *(O'Reilly & Munakata, 2000)*.

The excitatory net input/conductance g_e(t) or eta_j is computed as the proportion of open excitatory channels as a function of sending activations times the weight values:

$$\eta_j = g_e(t) = \langle x_i w_{ij} \rangle = \frac{1}{n}\sum_i x_i w_{ij}$$

The inhibitory conductance is computed via the kWTA function described in the next section, and leak is a constant.

Activation communicated to other cells (y_j) is a thresholded (Theta) sigmoidal function of the membrane potential with gain parameter gamma:

$$y_j(t) = \frac{1}{\left(1 + \frac{1}{\gamma[V_m(t) - \Theta]_+}\right)}$$

where [x]_+ is a threshold function that returns 0 if x<0 and x if X>0. Note that if it returns 0, we assume y_j(t) = 0, to avoid dividing by 0. As it is, this function has a very sharp threshold, which interferes with graded learning learning mechanisms (e.g., gradient descent). To produce a less discontinuous deterministic function with a softer threshold, the function is convolved with a Gaussian noise kernel (\mu=0, \sigma=.005), which reflects the intrinsic processing noise of biological neurons:

$$y_j^*(x) = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-z^2/(2\sigma^2)} y_j(z - x)dz$$

where x represents the [V_m(t) - \Theta]_+ value, and y^*_j(x) is the noise-convolved activation for that value. In the simulation, this function is implemented using a numerical lookup table.

17.1.2 k-Winners-Take-All Inhibition

Leabra uses a kWTA (k-Winners-Take-All) function to achieve inhibitory competition among units within a layer (area). The kWTA function computes a uniform level of inhibitory current for all units in the layer, such that the k+1th most excited unit within a layer is generally below its firing threshold, while the kth is typically above threshold. Activation dynamics similar to those produced by the kWTA function have been shown to result from simulated inhibitory interneurons that project both feedforward and feedback inhibition (OReilly & Munakata, 2000). Thus, although the kWTA function is somewhat biologically implausible in its implementation (e.g., requiring global information about activation states and using sorting mechanisms), it provides a computationally effective approximation to biologically plausible inhibitory dynamics.

kWTA is computed via a uniform level of inhibitory current for all units in the layer as follows:

$$g_i = g^\Theta_{k+1} + q(g^\Theta_k - g^\Theta_{k+1})$$

where 0<q<1 (.25 default used here) is a parameter for setting the inhibition between the upper bound of g^Theta_k and the lower bound of g^Theta_k+1. These boundary inhibition values are computed as a function of the level of inhibition necessary to keep a unit right at threshold:

$$g^\Theta_i = \frac{g^*_e \bar{g}_e (E_e - \Theta) + g_l \bar{g}_l (E_l - \Theta)}{\Theta - E_i}$$

where g^*_e is the excitatory net input without the bias weight contribution — this allows the bias weights to override the kWTA constraint.

In the basic version of the kWTA function, which is relatively rigid about the kWTA constraint and is therefore used for output layers, g^Theta_k and g^Theta_k+1 are set to the threshold inhibition value for the kth and k+1th most excited units, respectively. Thus, the inhibition is placed exactly to allow k units to be above threshold, and the remainder below threshold. For this version, the q parameter is almost always .25, allowing the kth unit to be sufficiently above the inhibitory threshold.

In the *average-based* kWTA version, g^Theta_k is the average g_i^Theta value for the top k most excited units, and g^Theta_k+1 is the average of g_i^Theta for the remaining n-k units. This version allows for more flexibility in the actual number of units active depending on the nature of the activation distribution in the layer and the value of the q parameter (which is typically .6), and is therefore used for hidden layers.

17.1.3 Hebbian and Error-Driven Learning

For learning, Leabra uses a combination of error-driven and Hebbian learning. The error-driven component is the symmetric midpoint version of the GeneRec algorithm *(O'Reilly,*

1996), which is functionally equivalent to the deterministic Boltzmann machine and contrastive Hebbian learning (CHL). The network settles in two phases, an expectation (minus) phase where the network's actual output is produced, and an outcome (plus) phase where the target output is experienced, and then computes a simple difference of a pre and post-synaptic activation product across these two phases. For Hebbian learning, Leabra uses essentially the same learning rule used in competitive learning or mixtures-of-Gaussians which can be seen as a variant of the Oja normalization *(Oja, 1982)*. The error-driven and Hebbian learning components are combined additively at each connection to produce a net weight change.

The equation for the Hebbian weight change is:

$$\Delta_{hebb} w_{ij} = x_i^+ y_j^+ - y_j^+ w_{ij} = y_j^+ (x_i^+ - w_{ij})$$

and for error-driven learning using CHL:

$$\Delta_{err} w_{ij} = (x_i^+ y_j^+) - (x_i^- y_j^-)$$

which is subject to a soft-weight bounding to keep within the 0-1 range:

$$\Delta_{sberr} w_{ij} = [\Delta_{err}]_+ (1 - w_{ij}) + [\Delta_{err}]_- w_{ij}$$

The two terms are then combined additively with a normalized mixing constant k_hebb:

$$\Delta w_{ij} = \epsilon [k_{hebb} (\Delta_{hebb}) + (1 - k_{hebb})(\Delta_{sberr})]$$

17.1.4 Implementational Overview

The Leabra implementation defines a full range of specialized network objects, from connections through layers. Unlike most PDP++ simulations, Leabra makes extensive use of the **LayerSpec** type, specifically the **LeabraLayerSpec**, which specifies the k-winners-take-all inhibition.

The new schedule process objects consist of three essential levels of processing, starting at the trial level and moving down through settling to the cycle, which implements one activation update of the network. Thus, the **LeabraTrial** process loops over the plus and minus phases of settling in the **LeabraSettle** process, which in turn iterates over cycles of the **LeabraCycle** process, which updates the activations of the units in the network.

17.2 Leabra Connection Specifications

The Leabra connection type **LeabraCon** contains the following parameters:

`wt` The weight value (shows up in NetView as r.wt for receiving, s.wt for sending

`dwt` Accumulated change in weight value computed for current trial: this is usually zero by the time NetView is updated

`pdw` Previous dwt weight change value: this is what is visible in NetView.

The Leabra connection specification **LeabraConSpec** type contains the following parameters:

`rnd` Controls the random initialization of the weights:

- `type` Select type of random distribution to use (e.g., UNIFORM (default), NORMAL (Gaussian)).
- `mean` Mean of the random distribution (mean rnd weight val).
- `var` Variance of the distribution (range for UNIFORM).
- `par` 2nd parameter for distributions like BINOMIAL and GAMMA that require it (not typically used).

`wt_limits` Sets limits on the weight values —Leabra weights are constrained between 0 and 1 and are initialized to be symmetric:

- `type` Type of constraint (GT_MIN = greater than min, LT_MAX = less than max, MIN_MAX (default) within both min and max)
- `min` Minimum weight value (if GT_MIN or MIN_MAX).
- `max` Maximum weight value (if LT_MAX or MIN_MAX).
- `sym` Symmetrizes the weights (only done at initialization).

`inhib` Makes the connection inhibitory (net input goes to g_i instead of net).

`wt_scale` Controls relative and absolute scaling of weights from different projections:

- `abs` Absolute scaling (s_k): directly multiplies weight value.
- `rel` Relative scaling (r_k): effect is normalized by sum of rel values for all incoming projections.

`wt_sig` Parameters for the sigmoidal weight contrast enhancement function:

- `gain` Gain parameter: how sharp is the contrast enhancement. 1=linear function.
- `off` Offset parameter: for values >1, how far above .5 is neutral point on contrast enhancement curve (1=neutral is at .5, values <1 not used, 2 is probably the maximum usable value).

`lrate` Learning rate (epsilon).

`cur_lrate` Current learning rate as affected by `lrate_sched`: note that this is only updated when the network is actually run (and only for ConSpecs that are actually used in network).

`lrate_sched` Schedule of learning rate over training epochs: to use, create elements in the list, assign start_ctr's to epoch vals when lrate's (given by start_val's) take effect. These start_val lrates *multiply* the basic lrate, so use .1 for a cur_lrate of .001 if basic lrate = .01.

`lmix` Sets mixture of Hebbian and err-driven learning:

- `hebb` Amount of Hebbian learning: unless using pure Hebb (1), values greater than .05 are usually to big. For large networks trained on many patterns, values as low as .00005 are still useful.
- `err` Amount of error-driven: automatically set to be 1-hebb, so you can't set this independently.

`fix_savg` Sets fixed sending avg activity value for normalizing netin (aka alpha_k): g_e_k = 1 / alpha_k < x_i w_ij >_k. This is useful when expected activity of sending region that projection actually receives is different from that of sending layer as a whole.

- `fix` Toggle for actually fixing the sending avg activation to value set in savg.
- `savg` The fixed sending average activation value — should be between 0 and 1.
- `div_gp_n` Divide by group n, not layer n, where group n is the number of actual connections in the connection group that this unit receives from (corresponds to a given projection). Usually, the netinput is averaged by dividing by layer n, so it is the same even with partial connectivity — use this flag to override where projection n is more meaningful.

`savg_cor` Correction for sending average activation levels in hebbian learning — renormalizes weights to use full dynamic range even with sparse sending activity levels that would otherwise result in generally very small weight values.

- `cor` Amount of correction to apply (0=none, 1=all, .5=half, etc): (aka q_m): alpha_m = .5 - q_m (.5 - alpha), where m = .5 / alpha_m), and Delta w_ij = epsilon [y_j x_i (m - wij) + y_j (1-x_i)(0 - wij)]
- `src` Source of the sending average act for use in correction. SLAYER_AVG_ACT (default) = use actual sending layer average activation. SLAYER_TRG_PCT = use sending layer target activation level. FIXED_SAVG = use value specified in fix_savg.savg. COMPUTED_SAVG = use actual computed average sending activation *for each specific projection* — this is very computationally expensive and almost never used.
- `thresh` Threshold of sending average activation below which Hebbian learning does not occur — if the sending layer is essentially inactive, it is much faster to simply ignore it. Note that this also has the effect of preserving weight values for projections coming from inactive layers, whereas they would otherwise uniformly decrease.

The **LeabraBiasSpec** connection specification is for bias weight (bias weights do not have the normal weight bounding and `wt_limits` settings, are initialized to zero with zero variance, and do not have a Hebbian learning component). The parameters are:

`dwt_thresh`
: Don't change weights if `dwt` (weight change) is below this value — this prevents bias weights from slowly creeping up or down and growing ad-infinitum even when the network is basically performing correctly — essentially a tolerance factor for how accurate the actual activation has to be relative to the target.

17.3 Leabra Unit Specifications

The parameters for the **LeabraUnit** unit object are as follows:

`spec`
: Determines the spec that controls this unit (not in NetView).

`pos`
: Determines location of unit within layer (not in NetView).

`ext_flag`
: Reflects type of external input to unit (not in NetView)

`targ`
: Target activity value (provided by external input from event).

`ext`
: External activation value when clamped (provided by external input from event).

`act`
: Activation value (what is sent to other units, y_j).

`net`
: Net input value (eta_j) computed as normalized weights times activations — excitation only, inhibition is computed separately as g_i `gc.i` either by kWTA or directly by unit inhib.

`bias`
: The bias weight is a LeabraCon object hanging off of the unit — it is managed by its own LeabraBiasSpec in the LeabraUnitSpec.

`act_eq`
: Rate-code equivalent activity value (time-averaged spikes when using discrete spiking activation, or just a copy of `act` when already using rate code activation).

`act_avg`
: Average activation over long time intervals, as integrated by time constant in `adapt_thr` (see LeabraUnitSpec). Useful to see which units are dominating, and to adapt their thresholds if that is enabled.

`act_m`
: Minus phase activation value, set after settling in minus phase and used for learning.

`act_p`
: Plus phase activation value, set after settling in plus phase and used for learning.

`act_dif`
: Difference between plus and minus phase activations — equivalent to the error contribution for unit (delta_j).

`da`
: Delta activation: change in activation from one cycle to the next, used for determining when to stop settling.

`vcb`
: Voltage-gated channel basis variables that integrate activation over time to determine if channels should be open or closed (channels are not active by default) – there are two types: hyst and acc, described next, followed by the parameters common to both.

`hyst`
: Hysteresis channel (excitatory) basis variable — typically hysteresis is triggered after unit achieves brief sustained level of excitation as reflected in this basis variable.

`acc` Accommodation channel (inhibitory) basis variable — typically accommodation (fatigue) is triggered after unit is active for a relatively long time period as reflected in this more slowly-integrating basis variable.

- `gc` Channel conductances for the different input channel types except excitatory input (which is in `net`).
- `l` Leak channel conductance (a constant, not visible in NetView).
- `i` Inhibition channel conductance, computed by kWTA or direct unit inhibition.
- `h` Hysteresis (voltage-gated excitation) channel conductance.
- `a` Accommodation (voltage-gated inhibition) channel conductance.

`I_net` Net current produced by all channels: what drives the changes in membrane potential.

`v_m` The membrane potential, integrates over time weighted-average inputs across different channels, provides basis for activation output via thresholded, saturating nonlinear function.

`i_thr` Inhibitory threshold value used in computing kWTA (g_i_Theta).

`spk_amp` Amplitude of spiking output (for depressing synapse activation function)

The **LeabraUnitSpec** unit-level specifications:

`act_range`
Range of activation for units: Leabra units are bounded between 0 (`min`) and 1 (`max`).

`bias_con_type`
Type of bias connection to make: almost always LeabraCon.

`bias_spec`
The LeabraBiasSpec that controls the bias connection on the unit.

`act_fun` The activation function to use: NOISY_XX1 (default), XX1 (not convolved with noise), LINEAR (`act` is linear function of `v_m` above threshold (0 below threshold)), SPIKE (discrete spiking).

`act` Specifications for the activation function:

- `thr` The threshold value Theta in: y_j = (gamma [V_m - Theta]_+) / (gamma [V_m - Theta]_+ + 1).
- `gain` Gain of the activation function (gamma in: y_j = (gamma [V_m - Theta]_+)(gamma [V_m - Theta]_+ + 1).
- `nvar` Variance of the Gaussian noise kernel for convolving with XX1 function in NOISY_XX1.

`spike` Specifications for the discrete spiking activation function (SPIKE):

- `dur` Spike duration in cycles — models extended duration of effect on postsynaptic neuron via opened channels, etc.

`v_m_r` Post-spiking membrane potential to reset to, produces a refractory effect and controls overall rate of firing (0 std).

`eq_gain` Gain for computing `act_eq` relative to actual time-average spiking rate (gamma_eq in: y_j^eq = gamma_eq (N_spikes)(N_cycles)).

`ext_gain` Gain for clamped external inputs, multiplies the `ext` value before clamping, needed because constant external inputs otherwise have too much influence compared to spiking ones.

`act_reg` Activity regulation via global weight scaling specifications (not used by default):

`on` whether to activity regulation is on (active) or not

`min` Increase weights for units below this level of average activation

`max` Decrease weights for units above this level of average activation

`wt_dt` rate constant for making weight changes to rectify over-activation (dwt ~= wt_dt * wt)

`opt_thresh`
Optimization thresholds for speeding up computation:

`send` Don't send activation when act <= send.

`learn` Don't learn on recv unit weights when both phase acts <= learn.

`updt_wts` Whether to apply `learn` threshold to updating weights (otherwise always update).

`phase_dif`
Don't learn when +/- phase difference ratio (- / +) < phase_dif. This is off (0) by default, but can be useful if network is failing to activate output (e.g., in a deep network) on minus phase of some trials — learning in this case is just massive increase in all weights, and tends to produce "hog" units for all the active units. To use, set to .8 as a good initial value.

`clamp_range`
Range of clamped (external) activation values (`min, max`) — Don't clamp to 1 because NOISY_XX1 activations can't reach that value, so use .95 as max.

`vm_range` Membrane potential range (`min, max`), 0-1 for normalized, -90-50 for bio-based.

`v_m_init` Random distribution for initializing the membrane potential (constant by default).

`dt` Time constants for integrating values over time:

`vm` Membrane potential `v_m` time constant: dt_vm in: V_m(t+1) = V_m(t) + dt_vm I_net-.

`net` Net input `net` time constant: dt_net in: g_e(t) = (1 - dt_net) g_e(t-1) + dt_net (1/n_p sum_k g_e_k + 1 / (beta/N)).

`g_bar` Maximal conductances for channels:

`e` Excitatory (glutamatergic synaptic sodium (Na) channel).

`l` Constant leak (potassium, K+) channel.

`i` Inhibitory GABA-ergic channel (computed by kWTA or directly).

`h` Hysteresis (excitation) voltage-gated channel (Ca++).

`a` Accommodation (fatigue, inhibition) voltage-gated channel (K+).

`e_rev` Reversal potentials for each channel (see above, defaults: 1, .15, .15, 1, 0).

`hyst` Hysteresis (excitation) voltage-gated channel specs, see accommodation (`acc`) for details, defaults are: false, .05, .8, .7, .1 and true).

`acc` Accommodation (fatigue, inhibition) voltage-gated channel:

`on` Activate use of channel if true.

`b_dt` Time constant for integrating basis variable, dt_b_a in: b_a(t) = b_a(t-1) + dt_b_a (y_j(t) - b_a(t-1)).

`a_thr` Activation threshold for basis variable, when exceeded opens the channel, aka Theta_a.

`d_thr` Deactivation threshold for basis variable, when less than closes channel (after having been opened), aka Theta_d.

`g_dt` Time constant for changing conductance when activating or deactivating, aka dt_g_a

`init` If true, initialize basis variables when state is initialized (else with weights).

`noise_type`
Where to add noise in the processing (if at all): NO_NOISE (default) = no noise, VM_NOISE = add to `v_m` (most commonly used), NETIN_NOISE = add to `net`, ACT_NOISE = add to activation `act`.

`noise` Distribution parameters for random added noise, default = GAUSSIAN, mean = 0, var = .001.

`noise_sched`
Schedule of noise variance over settling cycles, can be used to make an *annealing* schedule (rarely needed), use same logic as `lrate_sched` described in LeabraConSpec.

17.4 Leabra Layer Specifications

The **LeabraLayer** layer object has the following variables:

`n_units` Number of units to create with Build command (0=use geometry).

`geom` Geometry (size) of units in layer (or of each subgroup if geom.z > 1).

`pos` Position of layer within network.

`gp_geom` Geometry of subgroups (if geom.z > 1).

`projections`
Group of receiving projections for this layer.

`units` Units or groups of units in the layer.

`unit_spec`
Default unit specification for units in this layer: only applied during Build or explicit SetUnitSpec command.

`lesion` Inactivate this layer from processing (reversible).

`ext_flag` Indicates which kind of external input layer received.

`netin` Average and maximum net input (`net`) values for layer (`avg`, `max`). These values kept for information purposes only.

`acts` Avg and max activation values for the layer — `avg` is used for sending average activation computation in `savg_cor` in the ConSpec.

`acts_p` Plus-phase activation stats for the layer.

`acts_m` Minus-phase activation stats for the layer.

`acts_dif` Difference between plus and minus phase vals above.

`phase_dif_ratio`
Phase-difference ratio (`acts_p.avg / acts_m.avg`) that can be used with `phase_dif` in UnitSpec to prevent learning when network is inactive in minus phase but active in plus phase.

`kwta` values for kwta – activity levels, etc:

- `k` Actual target number of active units for layer.
- `pct` Actual target percent activity in layer.
- `k_ithr` Inhib threshold for kth most active unit (top k for avg-based).
- `k1_ithr` Inhib threshold for k+1th unit (other units for avg-based).
- `ithr_r` Log of ratio of ithr values, indicates sharpness of differentiation between active and inactive units.

`i_val` Computed inhibition values: `kwta` = kWTA inhibition, `g_i` = overall inhibition (usually same as kwta, but not for UNIT_INHIB).

`un_g_i` Average and stdev (not max) values for unit inhib.

`adapt_pt` Adapting kwta point values (if adapting, not by default).

`spec` Determines the spec that controls this layer.

`layer_links`
List of layers to link inhibition with (not commonly used).

`stm_gain` Actual stim gain for soft clamping, can be incremented to ensure clamped units active.

`hard_clamped`
If true, this layer is actually hard clamped.

The **LeabraLayerSpec** layer-level specifications consist of:

`kwta` How to calculate desired activity level:

- `k_from` How is the actual k determined: USE_K = directly by given k, USE_PCT = by pct times number of units in layer (default), USE_PAT_K = by number of units where external input `ext > .5` (`pat_q`).
- `k` Desired number of active units in the layer (default is meaningless — change as appropriate).
- `pct` Desired proportion of activity (used to compute a k value based on layer size).

`gp_kwta` Desired activity level for the unit groups (not applicable if no unit subgroups in layer, or if not in inhib_group). See `kwta` for values.

`inhib_group`
What to consider the inhibitory group. ENTIRE_LAYER = layer (default), UNIT_GROUPS = unit subgroups within layer each compute kwta separately, LAY_AND_GPS = do both layer and subgroup, inhib is max of each value.

`compute_i`
How to compute inhibition (g_i): KWTA_INHIB = basic kWTA between k and k+1 (default), KWTA_AVG_INHIB = average based, between avg of k and avg of k+1-n, UNIT_INHIB = units with `inhib` flag send g_i directly.

`i_kwta_pt`
Point to place inhibition between k and k+1 for kwta (.25 std), between avg of k and avg of k+1-n for avg-based (.6 std).

`adapt_i` Adapt either the `i_kwta_pt` point based on difference between actual and target pct activity level (for avg-based only, and rarely used), or or g_bar.i for unit-inhib.

- `type` What type of adaptation: NONE = nothing, KWTA_PT = adapt kwta point (i_kwta_pt) based on running-average layer activation as compared to target value, G_BAR_I = adapt g_bar.i for unit inhibition values based on layer activation at any point in time, G_BAR_IL adapt g_bar.i and g_bar.l for unit inhibition & leak values based on layer activation at any point in time
- `tol` Tolerance around target before changing value.
- `p_dt` Time constant for changing parameter.
- `mx_d` Maximum deviation from initial i_kwta_pt allowed (as proportion of initial)
- `l` Proportion of difference from target activation to allocate to the leak in G_BAR_IL mode

`a_dt` Time constant for integrating average average activation, which is basis for adapting i_kwta_pt

`clamp` How to clamp external inputs.

`hard` Whether to hard clamp external inputs to this layer (directly set activation, resulting in much faster processing), or provide external inputs as extra net input (soft clamping, if false).

`gain` tarting soft clamp gain factor (net = gain * ext).

`d_gain` For soft clamp, delta to increase gain when target units not >.5 (0 = off, .1 std when used).

`decay` Proportion of decay of activity state vars between various levels of processing:

`event` Decay between different events.

`phase` Decay between different phases.

`phase2` Decay between 2nd set of phases (if applicable).

`clamp_phase2`
If true, hard-clamp second plus phase activations to prev plus phase (only special layers will then update – optimizes speed).

`layer_link`
Link inhibition between layers (with specified gain), rarely used. Linked layers are in layer objects. `link` = Whether to link the inhibition, `gain` = Strength of the linked inhibition.

17.5 Leabra Proceses

The core set of Leabra processes consist of a **LeabraTrial** process that performs the two phases of contrastive Hebbian learning (CHL), where each phase of settling is controlled by a **LeabraSettle** process, which in turn iterates over a number of individual **LeabraCycle** processing steps, each of which updates the activation state of the network. These processes fit nicely within the standard settling and cycle processes (see Section 12.3 [proc-levels], page 191).

The **LeabraTrial** process iterates over two loops of settle processing, one for each phase. It has the following variables:

`Counter phase_order`
Different orders of phases can be presented, as indicated by the relatively self-explanatory options. The `MINUS_PLUS_NOTHING` option allows the network to learn to reconstruct its input state by taking away any external inputs in an extra third phase, and using this as a minus phase relative to the immediately preceding plus phase. The `MINUS_PLUS_PLUS` is used by more complex working-memory/context algorithms for an extra step of updating of context reps after the standard minus-plus learning phases. Note that the **PhaseOrderEventSpec** can be used to set the phase order on a trial-by-trial basis.

`Counter phase_no`
: The counter for this process, it goes from 0 to 1 for the two different phases (or 2 for more phases).

`Phase phase`
: The phase the process is in, which is just a more readable version of the counter: `MINUS_PHASE` and `PLUS_PHASE`.

`StateInit trial_init`
: Indicates what to do at the start of each trial process. Typically, one wants to `DECAY_STATE`, but it is also possible to `INIT_NOTHING` or `DO_NOTHING`. Decay state allows for working memory across trials, as needed by several modifications to Leabra. The default decay parameters decay 100%, which makes it equivalent to INIT_STATE.

`bool no_plus_stats`
: This flag means that statistics will not be recorded in the plus phase. This is useful because squared error, for example, is only meaningful in the minus phase, since the correct answer is clamped in the plus phase.

`bool no_plus_test`
: This flag means that the plus phase will not be run if the epoch process indicates that it is in `TEST` mode (i.e., no learning is taking place).

The **LeabraSettle** process iterates over cycles of settling (activation updating). `min_cycles` ensures that at least this many cycles of processing occur. It also contains several important variables that control how activations are computed during settling. `netin_mod` allows one to skip computing the net input every other cycle (or more), which can result in more efficient computation by allowing the membrane potentials to keep up better with netinput changes, but values higher than 2 are not recommended and have resulted in worse overall learning performance. `send_delta` is **very important for large networks** – it results in substantially faster processing by only sending netinput when the sending activation *changes*, instead of sending it all the time. The amount of change that is required is specified in the unit spec `opt_thresh` params.

The settle process will use the `duration` field from a **DurEvent** to set the max number of cycles to settle on a given event.

The **LeabraCycle** process updates the activation state of the network. It has no user-settable parameters.

17.6 Leabra Statistics

There are several statistics which are specific to the Leabra package, including one that compute the global "goodness" (aka energy) of the network's activation state (**LeabraGoodStat**), another set of statistics for measuring the probabilities of different activation states (**LeabraDistStat, LeabraTIGstat, LeabraTargStat**), and a statistic that can be used to control the length of settling based on the amount of activation change (**LeabraMaxDa**).

17.6.1 The Goodness (Energy) Statistic

The **LeabraGoodStat** computes the overall goodness of the activation state, which is composed of two terms, the *harmony* and *stress*. Harmony reflects the extent to which the activations satisfy the constraints imposed by the weights. It is just a sum over all units of the product of the activations times the weights:

$$H = \sum_j \sum_i a_j w_{ij} a_i$$

The stress term reflects the extent to which unit's activations are "stressed" at their extreme values. It is just the inverse sigmoidal function of the unit activation values:

$$S = \sum_j f^{-1}(a_j)$$

The total goodness is just the harmony minus the stress. These values are stored in the `hrmny`, `strss`, and `gdnss` stat value members of the goodness stat. The net input to a unit is used for computing the harmony term, since harmony is just the unit's activation times its net input.

17.6.2 Measuring the Maximum Delta-Activation

The **LeabraMaxDa** statistic computes the maximum delta-activation (change in activation) for any unit in the network. This is used to stop settling once the network has reached equilibrium. The stopping criterion for this statistic is the tolerance with which equilibrium is measured. It is created by default in the **LeabraSettle** process.

It can use the change in net current (`INET`) in addition to actual activation change (`da`) so as to not trigger a false alarm based on sub-treshold activations not changing (their net currents can be changing even if the activations are not). Once the layer average activation goes over `lay_avg_thr`, the stat switches over to measuring `da` instead of inet.

17.7 Leabra Defaults

The following default files (see Section 9.5 [proj-defaults], page 126) are available for configuring different versions of the Leabra objects:

'`leabra.def`'
: This is the standard defaults file.

'`leabra_seq.def`'
: Sequence-based processes for doing sequences of events within event groups.

'`leabra_ae.def`'
: An auto-encoder configuration for doing MINUS_PLUS_NOTHING kind of learning automatically.

'`leabra_seq_ae.def`'
: Sequence plus auto-encoder.

'`leabra_seq_time.def`'
: Uses LeabraTimeUnit units that can learn based on temporal differences across sequential trials. This never worked very well and is superceeded by various context memory approaches.

'`leabra_seq_ae_time.def`'
: Sequences plus auto-encoder plus time-based units!

17.8 Leabra Misc Special Classes

Leabra has a number of special derived classes for doing special things beyond the standard learning mechanisms. These are not well documented here – interested users should refer to the source code. The **LeabraWiz** wizard object has some specialized functions for creating some of these classes, and can also setup unit-based inhibition in the network.

The **LebraACLayerSpec** implements an Adaptive Critic for performing Temporal Differences reinforcement learning. This implementation is not particularly good relative to the new improved **RewPredLayerSpec**, but it is simpler and is widely used.

The **LeabraContextLayerSpec** implements a Simple Recurrent Network context layer, and can be constructed automatically using the **LeabraWiz**. The fancier **LeabraGatedCtxLayerSpec** takes gating control signals from an AC (adpative critic) unit uses them to control the rate of context updating. The **LeabraACMaintLayerSpec** is another version of this idea, which uses intrinsic maintenance currents (i.e., hysteresis currents) to maintain information in the context layer over time, as modulated by the AC signal. See *Rougier & O'Reilly, 2002* for further documentation on this mechanism.

The **PhaseOrderEventSpec** allows for an event to control its own set of phases that it will use.

The **LeabraTimeConSpec**, **LeabraTimeUnit**, and **LeabraTimeUnitSpec** implement learning across two adjacent events. The units store prior trial minus and plus phase activations (`p_act_m` and `p_act_p`) and use these for learning.

The **LeabraNegBiasSpec** makes the bias weight only learn based on negative error derivatives (i.e., the bias weight can only go negative). The `decay` parameter allows this negative bias to recover back towards zero. This is useful for implementing a simple form of search, where things that produce errors are made less likely to be activated in the near future.

The **LeabraTabledConSpec** allows for learning to be driven by a lookup table, e.g., for exploring biologically-derived learning rules that do not have a simple analytic form.

The **ScalarValLayerSpec** implements a layer that represents a single scalar value usign a coarse-coded representation, where each unit has a designated target value, and it "votes" for this value in proportion to its activation strength. The overall coded value is the weighted average of these target values times the unit activations. The first unit in the layer contains a convenient readout of the represented scalar value, and is otherwise prevented from participating in normal network updating. Clamping a value to this first unit causes the rest of the units to be clamped into Gaussian-shaped bump representing that value.

The **MarkerConSpec** is useful for marking special connections that operate according to non-standard functions. It turns off learning and does not contribute to normal netinput computations.

The **TdModUnit** and **TdModUnitSpec** add a temporal-differences modulation value to the unit variables. They also include the time-based variables found in the Time classes. These are used in the PFC/BG learning classes.

The **RewPredLayerSpec** is an improved version of the AC unit, which is based on the ScalarValLayerSpec, so it produces a coarse-coded representation of reward expectation. It also deals with non-absorbing rewards much better.

The following specs implement an experimental version of dynamically-gated working memory based on the biology of the prefrontal cortex (PFC) and basal ganglia (BG). See the `HelpConfig` button for more specific information. **PatchLayerSpec**, **MatrixLayerSpec**, **MatrixUnitSpec**, **ImRewLayerSpec**, **SNcLayerSpec**, **PFCLayerSpec**.

17.9 Leabra Implementational Details

TBW. Sorry.

18 Programming in PDP++

This chapter contains some useful information for those who want to add new functionality to the PDP++ software by compiling their own executable. By creating new subclasses of existing classes, and using these new classes in your simulations, it should be possible to make PDP++ do exactly what you want it to.

Before taking this step, you should be reasonably comfortable with the CSS language and using it to access objects in the simulator. Further, you will need to know (or learn about) C++ in a bit more detail than is covered in the CSS section of this manual. There are a number of good books on this subject available in most bookstores.

This chapter describes how to set up the makefiles in your own directory where you will compile your executable. It then describes various coding conventions and extensions to the basic C++ language that we have added to facilitate programming in PDP++. We have established a standard way of dealing with creating, copying, and deleting objects. In addition, each object has special functions that allow groups to manage them. All of these "coding conventions" are described in this chapter.

We have developed a run-time-type-information (RTTI) system called TypeAccess, which provides type information about most classes at run-time. This can be used to determine what kind of unit a Unit* object *really* is, for example (i.e., is it a BpUnit or a MyWackyBpUnit?).

The TypeAccess system requires a more complicated than normal set of makefiles. Fortunately, it is reasonably straightforward to use the makefiles we have developed, so you won't have to deal with much of this complexity.

Most of the graphical interface (i.e., edit dialogs, menus, etc) is generated automatically from the information provided by TypeAccess. The same is true for the way you can transparently access hard-coded types and objects through CSS. Thus, you don't need to do anything special to be able to use your newly defined classes exactly in the way that you use the ones that come with the software.

There are some special keywords that you can put in the comments for your classes and class members and methods called "comment directives". These comment directives allow you to control various aspects of how the GUI and CSS treat your objects. These comment directives are described in this chapter.

18.1 Makefiles and Directory Organization

The PDP++ code should be installed in '`/usr/local/pdp++`', or some such similar place. This path will be referred to from here on out as PDPDIR. This directory contains a set of sub-directories, like '`demo`' and '`manual`', etc. which contain different pieces of the distribution. See Section 2.2 [inst-prog], page 9 for instructions on how to install and compile the source distribution of the PDP++ software. In order compile your own additions to the software, you must install and compile the source code distribution!

The critical directories from a programmer's perspective are the '`src`', which contains the source code, '`config`', which contains the Makefile configuration stuff, '`include`' which has links to the header files, and '`lib`' which has links to the various libraries.

Each sub-directory within the '`src`' directory contains code relevant to some sub-component of the PDP++ software. These directories are as follows:

'`ta_string`'
: The basic String class used throughout the software. It is a slightly modified version of the GNU String class that is distributed with the libg++ distribution (version 2.6.2).

'`iv_misc`'
: Contains a number of extra pieces of code that supplement the InterViews GUI toolkit.

'`ta`'
: Contains the TypeAccess system, which gives classes the ability to know all about themselves and other classes at run time. The use of this software is what makes the largely automatic interface used in PDP++ possible. It is described further in Section 18.2 [prog-typea], page 286. This directory also contains a lot of basic objects, like **Array** (Section 8.3 [obj-array], page 114), **List** and **Group** (Section 8.2 [obj-group], page 109) objects.

'`css`'
: Contains the code for the CSS script language (see Chapter 7 [css], page 71).

'`iv_graphic`'
: Contains a set of objects which implement a graphical object manipulation environment that is used in the network viewer (see Section 10.6 [net-view], page 149) and the graph log (see Section 13.3.4 [log-views-graph], page 216).

'`ta_misc`'
: Contains a smorgasbord of various objects that might have general applicability, and are not specifically PDP objects.

'`pdp`'
: Where all of the specific pdp code is.

'`bp`'
: Implements bp and rbp.

'`cs`'
: Implements cs.

'`so`'
: Implements so.

'`bpso`'
: Implements the combined bp and so executable (just links the libraries).

'`leabra`'
: Implements the leabra algorithm.

Each directory has a set of include files which can be accessed as <xxx/yyy.h>, where xxx is one of the directory names given above. In addition, each directory has its own library, which is just '`libxxx.a`', where xxx is the name of the directory (without any underbars). The bp, cs and so directories have a library name of libpdp_xx.a, to indicate that they are part of the pdp software.

All of the compilation results (e.g. object files) go in a subdirectory named after the CPU type being used. The user must set the CPU environmental variable appropriately, as per the definitions used in the InterViews system. The ones that everything has been tested and compiled on are listed in the installation guide (see Chapter 2 [inst], page 4), and the INSTALL file.

Other possibilities are listed in Section 2.2 [inst-prog], page 9. This should be the same as when the system was first installed.

The include files and library are made in two stages. The first stage involves compiling the object files into the CPU subdirectory. Then, if everything goes ok, the library is made, which is then copied into a further subdirectory of the CPU subdirectory called '`lib_include`'. Also, all of the header files are compared with those already in the '`lib_include`' subdirectory (if any), and those ones that are different are copied over. It is these header files in '`lib_include`' that the '`PDPDIR/include`' directory makes links to, and thus these are the ones that are included by other programs. This setup allows one to test out a set of code by making an executable in a given directory and getting things working before installing the new header files and library for the rest of the system to use.

In order to add functionality to the software, one needs to create a new directory, and then include various files from the above directories, and link in their respective libraries. This directory can be located in the same master directory as the main distribution, or it can be located in your own home directory somewhere. This latter option is the preferred one.

We have developed a shell file that does all of the things necessary to create your own directory. The first step is to make a master directory off of your home directory, typically called '`pdp++`'.

Then, run the '`PDPDIR/bin/mknewpdp`' command from this new '`home/pdp++`' directory with an argument which is the name of the directory/executable that you want to make. This will give you step-by-step instructions. In the end, you will end up with a directory that contains some sample code in the form of a .h and .cc file with the same name as the directory.

The script will have installed a '`Makefile`' in your directory which is the same as the one's found in the main PDP++ directories. These makefiles are constructed by concatenating together a bunch of pieces of makefiles, some of which contain standard make actions, and others which contain specific defaults for particular machines. All of the pieces are found in the '`PDPDIR/config`' directory.

The makefiles named '`Makefile.CPU.[CC|g++]`' are the machine-specific files that you should edit to make sure they have all the right settings for your system. This should have been done already during the installation of the PDP++ source code distribution, see Section 2.2 [inst-prog], page 9 for details.

To these pieces is added the specific list of files that need to be made in your directory. This is specified in the '`Makefile.in`' file. This is the only makefile you should edit. It can be used to override any of the settings in the standard makefiles, simply by redefining definitions or actions. If you add files to your directory, follow the example for the one already in your default '`Makefile.in`' that was installed with the '`mknewpdp`' command.

Note that there are a couple of compiler-specific "extra" files in the directory. These have the name of the directory plus a '`_vt.cc`' or '`_it.cc`' suffix. The '`_vt`' file is for virtual table instantiation, which is controlled in cfront with the +e0/+e1 arguments. It simply includes most of the header files in the software. We have found that by compiling everything except the '`_vt`' file with +e0 that the executables are much smaller. This is even true in cfront versions where they had "fixed" this problem. You can try doing the other way by leaving out the +e args and not using the '`_vt`' file (see the definitions in

'PDPDIR/config/Makefile.defs' for how to do this: change your 'Makefile.CPU' file and recompile the entire distribution first..).

The '_ti.cc' is the template instantiation file needed by gnu g++ version 2.6.3 (reportedly, it won't be needed in 2.7). It contains explicit instantiations of all of the templates used in each library. For user directories, this probably isn't needed, but its there if you do declare any templates and encounter link problems with g++. Also, the 'Makefile.CPU.g++' show how this file gets included in the making of a given project.

New for 2.0: All of the makefile actions, as shown below, are now available using a consistent syntax structure: all lower case, with underbars separating different words. This makes it much easier to remember what command to type. The old eclectic combinations of upper and lower case words, etc are still available if you already know them.

The commonly-used actions that are defined in the makefile are as follows:

'make bin, make opt_bin, make dbg_bin'
: Makes the binary from the files in this directory. Bin makes the default form specified in the make file, while opt and dbg make optimized and debug versions, respectively.

'make re_bin, make opt_re_bin, make dbg_re_bin'
: Same as above, except it first removes the executable before making. This is useful if a library has changed but no header files from that library were changed.

'make lib, make opt_lib, make dbg_lib'
: Like the above, except it makes a library containing the relevant .o files.

'make new_makefile'
: This makes a new version of the 'Makefile' file in the current directory. This concatenates all of the different parts that together make up a single 'Makefile'. However, it does not make a CPU directory, which is necessary to actually compile (see cpu_dir next).

'make cpu_dir, make local_cpu_dir'
: This makes and configures a directory with the same name as the CPU environmental variable (reflecting the CPU type of the machine) suitable for compiling the object files into. If local_cpu_dir is made first, then this directory is actually a symbolic link to a directory created on a disk local to the current machine, so that compilation will be faster than if the directory where the source is located is a networked (slow) directory (i.e., NFS). The cpu_dir action copies the current 'Makefile' into the directory, and configures the directory for compiling. Note that these actions remove any existing dependency information, so that a depend action should be made following either of them.

'make depend'
: This automatically adds dependency information for the files in this directory onto the 'CPU/Makefile' file. This allows the make command to know when to compile these files after something they depend on has been touched (edited).

'`make makefiles, make make_depend, make new_make_depend`'
: These actions simply combine some of the above steps together into one action. `makefiles` does a `new_makefile` and then a `cpu_dir`, `make_depend` does a `cpu_dir` and then a `depend`, and `new_make_depend` does all three of the necessary steps: `new_makefiles`, `cpu_dir`, and `depend`. The only reason you should not use the latter all the time is if your `make` program has trouble using a new '`Makefile`' (i.e., as created by the `make new_makefiles` action) for calling the subsequent actions. In this case, you have to first do a `make new_makefiles` and then you can do a `make make_depend`.

'`make force_ta`'
: Forces a call to the TypeAccess scanning program '`maketa`'.

18.2 The TypeAccess System

The TypeAccess system consists of a set of objects that can hold type information about class objects. This type information includes the names and types of all the members and methods in a class, the parents of the class, etc. This information can be used by classes to get information about themselves at run time. In addition, the TypeAccess system provides a set of type-aware base classes and macros for defining derived versions of these that can be used to easily incorporate run-time type information into any C++ system.

In addition to being type-aware, the base classes can use their own type information to save and load themselves automatically to and from ASCII format text files. Further, there is an extensible graphical interface based on InterViews which can automatically build editing dialogs for filling in member values and calling member functions on arbitrary objects. Finally, the type information can be used to provide a transparent script-level interface to the objects from the CSS script language. This provides the benefits of compiled C++ for fast execution, and the ability to perform arbitrary interactive processing in an interpreter using the C++ language supported by CSS.

Many features of the interface and script level interface, as well as various options that affect the way objects are saved and loaded, can be specified in comments that follow the declaration of classes, members, and methods. These *comment directives* constitute a secondary programming language of sorts, and they greatly increase the flexibility of the interface. They are documented in Section 18.3 [prog-comdir], page 295.

Thus, the PDP++ software gets much of its functionality from the TypeAccess system. It provides all of the basic interface and file-level functionality so that the programmer only needs to worry about defining classes that perform specific tasks. These classes can then be flexibly used and manipulated by the end user with the generic TypeAccess based interface.

18.2.1 Scanning Type Information using '`maketa`'

Type information for TypeAccess is scanned from the header files using a program called '`maketa`', which looks for `class` and `typedef` definitions, and records what it finds. It operates on all the header files in a given directory at the same time, and it produces three output files: '`xxx_TA_type.h`', '`xxx_TA_inst.h`', and '`xxx_TA.cc`', where xxx is given by a "project name" argument. The first file contains a list of `extern` declarations of instances of the **TypeDef** type, which is the basic object that records information about types. Each

type that was defined with a `class` or `typedef`, or ones that are modifications of basic types, such as reference or pointer types, are given their own **TypeDef** object, which is named with the name of the type with a leading `TA_` prefix. Thus, a class named *MyClass* would have corresponding **TypeDef** object named *TA_MyClass*, which can be used directly in programs to obtain type information about the *MyClass* object. a Pointers have a `_ptr` suffix, and references have a `_ref` suffix. Template instances are represented by replacing the angle brackets with underbars. The '`xxx_TA_type.h`' file must be included in any header files which reference their own type information.

The '`xxx_TA_inst.h`' file contains declarations of "instance" objects, which are pointers to a token of each of the classes for which type information is available. These instances are named `TAI_` with the rest the same as the corresponding `TA_` name. The `-instances` argument to '`maketa`' determines if instances are made, and this can be overridden with the `#NO_INSTANCE` and `#INSTANCE` comment directives (see Section 18.3 [prog-comdir], page 295). The **TypeDef** object can use an instance object of one of the type-aware base classes to make a new token of that object given only the name of the type to be created. This gives the system the power to create and delete objects at will, which is necessary for the file saving and loading system to work.

Finally, the '`xxx_TA.cc`' file contains the actual definitions of all the type information. It must be compiled and linked in with the project, and its `ta_Init_xxx` function must be called at the start of the execution of the program before any type information is used.

Note that while '`maketa`' does process complexities like `template` and multiply inherited classes properly, it does not deal with multiple versions of the same function which differ only in argument type in the same way that C++ does. Instead, the scanner just keeps the last version of a given method defined on the class. This makes the type information compatible with the limitations of CSS in this respect, since it does not know how to use argument types to select the proper function to be called (see Section 7.3.1 [css-c++-diff], page 84). This limitation greatly simplifies the way that functions are called by CSS. It is recommended that you create methods which have some hint as to what kinds of arguments they expect, in order to get around this limitation. The **taList** and **taGroup** classes, for example, contain both overloaded and specific versions of the `Find` function, so the C++ programmer can call `Find` with any of a number of different argument types, while the CSS programmer can use the `FindName` or `FindType` versions of the function.

18.2.2 Startup Arguments for '`maketa`'

The type-scanning program '`maketa`' takes the following arguments:

`[-v<level>]`
: Verbosity level, 1-5, 1=results,2=more detail,3=trace,4=source,5=parse.

`[-hx | -nohx]`
: Generate .hx, .ccx files instead of .h, .cc. This is used in conjunction with a makefile that compares the .hx with the .h version of a file and only updates the .h if it actually differs from the .hx version. This prevents lots of needless recompiling when the type-scanned information is not actually different when a header file was touched.

`[-css]` Generate CSS stub functions. The stub functions take **cssEl*** arguments, and call member functions on classes. These must be present to use CSS to call member functions on classes, or to call functions from the edit dialog menus and buttons.

`[-instances]`
Generate instance tokens of types. Instances are needed to make tokens of class objects.

`[-class_only | -struct_union]`
Only scan for `class` types (else `struct` and `union` too). The default is to only scan for `class` types because they are always used in the definition of a class object. `struct` and `union` can be used to modify the type name in old-style C code, which can throw off the scanner since these don't amount to class definitions.

`[-I<include>]...`
Path to include files (one path per -I).

`[-D<define>]...`
Define a pre-processor macro.

`[-cpp=<cpp command>]`
Explicit path for c-pre-processor. The default is to use '`/usr/lib/cpp`', which doesn't work very well on C++ code, but its there. It is recommended that you use cccp, which is the gnu preprocessor that comes with gcc.

`[-hash<size>]`
Size of hash tables (default 2000), use -v1 to see actual sizes after parsing all the types.

`project` This is the stub project name (generates project_TA[.cc|_type.h|_inst.h]).

`files...` These are the header files to be processed.

18.2.3 Structure of TypeAccess Type Data

The classes used in storing type information in the TypeAccess system are all defined in the '`ta/typea.h`' header file. Basically, there are a set of **Space** objects, which all derive from a basic form of the `List` object (defined in '`ta/ta_list.h`', which represent type spaces, member spaces, method spaces, etc. These are just containers of objects. The spaces are: **TypeSpace**, **MemberSpace**, **MethodSpace**, **EnumSpace**, **TokenSpace**. Note that they contain functions for finding, printing, and generally manipulating the objects they contain.

There are corresponding **TypeDef**, **MemberDef**, **MethodDef**, and **EnumDef** objects which hold specific information about the corresponding aspect of type information. The **TypeDef** contains the following fields:

`String name`
Holds the name of the type.

`String desc`
: A description which is obtained from the user's comment following the declaration of the type.

`uint size`
: The size of the object in bytes.

`int ptr`
: The number of pointers this type is from a basic non-pointer type.

`bool ref`
: True if this is a reference type.

`bool internal`
: True if this type information was automatically or internally generated. This typically refers to pointer and reference types which were created when the scanner encountered their use in arguments or members of other classes that were being scanned.

`bool formal`
: True for basic level objects like `TA_class` and `TA_template` which are are formal parents (`par_formal`) of types that users declare. These provide a way of determining some basic features of the type. Formal type objects are declared and installed automatically by the type scanning system.

`bool pre_parsed`
: True if this type was registered as previously parsed by the type scanning system (i.e., it encountered an '`extern TypeDef TA_xxx`' for this type, where xxx is the name of the type). These types don't get included in the list of types for this directory. This makes it possible to do type scanning on a complex set of nested libraries.

`String_PArray inh_opts`
: These are the options (comment directives) that are inherited by this type (i.e., those declared with a `##` instead of a `#`).

`String_PArray opts`
: These are all of the options (comment directives) for this type, including inherited and non-inherited ones.

`String_PArray lists`
: A list of the `#LIST_xxx` values declared for this type.

`TypeSpace parents`
: A list of parents of this type. There are multiple parents for multiple-inheritance `class` types, and for `internal` types which are the combination of basic types, such as `unsigned long`, etc.

`int_PArray par_off`
: A list of offsets from the start of memory occupied by this class where the parent object begins. These are used for multiply inherited class types. They are in a one-to-one correspondence with the `parents` entries.

`TypeSpace par_formal`
: A list of the formal parents of this type, including `TA_class`, etc.

`TypeSpace par_cache`
: A special cache of frequently-queried type parents. Currently if a type derives from **taBase**, then `TA_taBase` shows up here (because a lot of the TypeAccess code checks if something is derived from the basic type-aware type **taBase**).

`TypeSpace children`
: A list of all the types that are derived from this one.

`void** instance`
: A pointer to a pointer of an instance of this type, if it is kept. The `GetInstance` function should be used to get the actual instance pointer.

`TokenSpace tokens`
: A list of the actual instances or tokens of this type that have been created by the user (the `TAI_xxx` instance object is not registered here). These are not kept if the type does not record tokens (see the `#NO_TOKENS` comment directive, Section 18.3.1 [comdir-objs], page 296).

`taivType* iv`
: A pointer to an object which defines how a token of this type appears in a GUI edit dialog. There is a "bidding" procedure which assigns these objects, allowing for the user to add new specialized representations which out-bid the standard ones. This bidding takes place when the gui stuff is initialized, and the results are stored here.

`taivEdit* ive`
: This is like the `iv` pointer, except it is the object which is used to generate the entire edit dialog for this object. It also is the result of a bidding procedure.

`taBase_Group* defaults`
: These are pointers to different **TypeDefault** objects for this type. Each **TypeDefault** object is for a different scope where these types can be created (i.e., a different **Project** in the PDP++ software).

`EnumSpace enum_vals`
: Contains the enum objects contained within a given `enum` declaration.

`TypeSpace sub_types`
: These are the sub-types declared with a `typedef`, `enum`, or as part of a template instantiation within a `class` object.

`MemberSpace members`
: These are the members of a `class` object.

`MethodSpace methods`
: These are the methods of a `class` object.

`TypeSpace templ_pars`
: These are the template parameters for template objects. In the `template` itself, they are the formal parameters (i.e., `T`), but in the template instance they point to the actual types with which the template was instantiated.

The most important functions on the **TypeDef** object are as follows:

`bool HasOption(const char* op)`
: Checks to see if the given option (comment directive) (don't include the `#`) is present on this type.

`String OptionAfter(const char* op)`
: Returns the portion of the option (comment directive) after the given part. This is used for things like `#MENU_ON_xxx` to obtain the xxx part. If option is not present, an empty string is returned.

`InheritsFrom(TypeDef* tp)`
: Checks if this type inherits from the given one (versions that take a string and a reference to a **TypeDef** are also defined). Inheritance is defined only for classes, not for a pointer to a given class, for example. Thus, both the argument and the type this is called on must be non-pointer, non-reference types.

`DerivesFrom(TypeDef* tp)`
: Simply checks if the given type appears anywhere in the list of parents for this type. Thus, a pointer to a class derives from that class, but it does not inherit from it.

`String GetValStr(void* base, void* par=NULL, MemberDef* memb_def=NULL)`
: Uses the type-scanned information to obtain a string representation of the value of an instance of this type. `base` is a pointer to the start of a token of this type, and `par` and `member_def` can be passed if it is known that this token is in a parent class at a particular member def. This and the following function are used widely, including for saving and loading of objects, etc.

`SetValStr(const char* val, void* base, void* par=NULL, MemberDef* memb_def=NULL)`
: Takes a string representation of a type instance, and sets the value of the token accordingly (it is the inverse of `GetValStr`).

`CopyFromSameType(void* trg_base, void* src_base, MemberDef* memb_def=NULL)`
: Uses the type-scanned information to copy from one type instance to the next. Any class objects that are members are copied using that object's copy operator if one is defined (this is only known for derivatives of the **taBase** base class).

`Dump_Save(ostream& strm, void* base, void* par=NULL, int indent=0)`
: This will save the given type object to a file. Files are saved in an ASCII format, and are capable of saving pointers to other objects when these objects derive from the **taBase** object. Special code is present for dealing with groups of objects stored in the **taList** or **taGroup** classes. See Section 18.2.5 [prog-typea-dump], page 295 for more details.

`Dump_Load(istream& strm, void* base, void* par=NULL)`
: This will load a file saved by the `Dump_Save` command.

The other **Def** objects are fairly straightforward. Each includes a `name` and `desc` field, and a list of `opts` (comment directives) and `lists`. Also, each contains an `iv` field which represents the item in the GUI edit dialog, and is the result of a bidding process (see the `iv` field in the **TypeDef** object above). They all have the `HasOption` and `OptionAfter` functions plus a number of other useful functions (see the '`ta/typea.h`' for details).

MemberDef objects contain the following additional fields. Note that derived classes contain links (*not copies*) of the members and methods they inherit from their parent, except when the class has multiple parents, in which case copies are made for the derived class because the offset information will no longer be the same for the derived class.

`TypeDef* type`
: The type of the member.

`ta_memb_ptr off`
: The address or offset of this member relative to the start of the memory allocated for the class in which this member was declared.

`int base_off`
: The offset to add to the base address (address of the start of the class object) to obtain the start of the class this member was declared in. This is for members of parents of multiply-inherited derived classes.

`bool is_static`
: True if the member was declared `static`. Thus, it can be accessed without a `this` pointer. The `addr` field contains its absolute address.

`void* addr`
: The absolute address (not relative to the class object) of a static member.

`bool fun_ptr`
: True if the member is actually a pointer to a function.

The **MethodDef** object contains the following additional variables:

`TypeDef* type`
: The type of the method.

`bool is_static`
: True if the method was declared `static`.

`ta_void_fun addr`
: The address of a `static` method. Non-static methods do not have their addresses recorded. Methods are called via the `stubp` function, if the `-css` option was used during scanning.

`int fun_overld`
: The number of times this function was overloaded (i.e., a function of the same name was declared in the class or its parents). TypeAccess does not perform name mangling on functions, so only one instance of a given method is recorded. It is the last one that the scanner encounters that is kept.

`int fun_argc`
: The number of arguments for this function.

`int fun_argd`
: The index where the arguments start having default values. Thus, the function can be called with a variable number of arguments from `fun_argd` to `fun_argc`.

`TypeSpace arg_types`
: These are the types of the arguments.

`String_PArray arg_names`
: These are the names of the arguements (in one-to-one correspondence with the types).

`css_fun_stub_ptr stubp`
: A pointer to a "stub" function which calls this method using **cssEl** objects as arguments. This function is defined in the '`xxx_TA.cc`' file if the `-css` argument is given to '`maketa`'. The **cssEl** objects have conversion functions for most types of arguments, so that the function is called by casting the arguments into the types expected by the function. Pointers to class objects are handled by `cssTA` objects which have a pointer and a corresponding `TypeDef` pointer, so they know what kind of object they point to, making conversion type-safe. These stubs return a cssEl object. They also take a `void*` for the `this` object. These stubs are used both by CSS and to call methods from the edit dialogs from menus and buttons.

18.2.4 The Type-Aware Base Class taBase

There is a basic class type called **taBase** that uses the TypeAccess type information to perform a number of special functions automatically. This object is aware of its own type information, and can thus save and load itself, etc. Special code has been written in both the TypeAccess system and in CSS that takes advantage of the interface provided by the **taBase** type. Thus, it is recommended that user's derive all of their types from this base type, and use special macros to provide derived types with the hooks necessary to get their own type information and use it effectively. The type **TAPtr** is a `typedef` for a pointer to a **taBase** object. The definition of a **taBase** object and the macros that are used with it are all in '`ta/ta_base.h`'.

All **taBase** objects have only one member, which is a reference counter. This provides a mechanism for determining when it is safe to delete an object when the object is being referenced or pointed to in various different places. **taBase** provides a set of referencing and pointer-management functions that simplify the use of a reference-count based memory management system. `Ref` increments the reference count, `unRef` decrements it, `Done` checks if the refcount is zero, and deletes the object if it is, and `unRefDone` does both. `Own` both Ref's an object and sets its owner. For pointers, `SetPointer` unrefs any existing object that the pointer points to, and sets it to point to the new object. `DelPointer` does an `unRefDone` on the object pointed to, and sets the pointer to NULL. `OwnPointer` is like SetPointer except it also sets the owner of the pointed-to object to the one given by the argument. See Section 18.4.2 [coding-funs], page 304 and '`ta/ta_base.h`' for more details.

The one essential function that **taBase** provides is `GetTypeDef()`, which is a virtual function that returns a pointer to the `TypeDef` type descriptor for this object. This function is defined as part of the basic macro `TA_BASEFUNS`, which must be included in all classes derived from **taBase**. This function makes it possible for a generic pointer to a taBase object to find out what type of object is really being pointed to.

There are a number of functions defined on the taBase type that simply call the corresponding function on the **TypeDef** pointer. These can be found in the '`ta/ta_base.h`' header file. They just make it easier to call these commonly-used functions, instead of requiring the user to put in a `GetTypeDef` function in between.

taBase also provides a simplified way of managing the construction, deletion, and copying of an object. Basically, construction is broken down into a set of functions that `Initialize` the member variables, `Register` the new token with the type if it is keeping track of tokens, and it sets the default name of the object based on its type name using `SetDefaultName`. The `TA_BASEFUNS` macro defines a default constructor that calls these three functions in that order. The user thus needs to provide a `Initialize` function for every class defined, which does the appropriate member initialization. Note that if this function is not defined, the one on the parent class will be called twice, so its more efficient to include a blank Initialize function when there are no members that need to be initialized.

The destructor function is similar to the constructor. A default destructor is defined in `TA_BASEFUNS`, which simply calls `unRegister`, and `Destroy`. Thus, the user needs to provide a `Destroy` function which frees any additional resources allocated by the object, etc. Like `Initialize`, a blank `Destroy` should be defined when there is nothing that needs to be done to prevent the parent function from being called twice.

Copying, cloning, and making a new token of the given type are also supported in the **taBase** class. The `Copy` function performs the basic copy operations for both the copy constructor and the `=` operator. This should replace the values of this class and any of its existing sub-objects with those of the object passed to it, as it is intended for assignment between two existing objects. In general, the `=` operator should be used for copying all members, except for the case of `LINK_GROUP` groups and lists, which should use the `BorrowUnique` function (since they do not own the items in the list, just link them). `Copy` must call the parent's `Copy` function as well. As a minor simplification of calling the parent (and to provide a copy function for just the items in a given class), it is conventional to define a `Copy_` function, which does everything except for calling the parent copy function. The macro `COPY_FUNS` can be used to define a `Copy` function which calls the parent function and then `Copy_`. The macro `SIMPLE_COPY` defines a `Copy_` function which uses the type-scanned information to do the copying. It is slower than hand-coding things, so it probably shouldn't be used on types which will have a lot of tokens or be copied often.

A `Clone` function which returns a `TAPtr` to a new duplicate of this object is defined in `TA_BASEFUNS`, as well as an "unsafe" version of `Copy` (`UnSafeCopy`), which takes a generic `TAPtr` argument and casts it into their type. The argument's type should thus be tested before calling this function. A safe interface to this function is provided by the `CopyFrom` function, which does the type checking. Finally, the `MakeToken` function will create a new token of the type.

The **taBase** class also contains functions for creating and manipulating a structure hierarchy of objects. This is where certain objects contain groups of other objects, which contain other objects, etc. For example, the PDP++ software has a structure hierarchy built around a root object, which contains projects, which contain lots of other objects like networks, projects, environments, etc. Special container objects like **taList** and **taGroup** play an important role in representing and manipulating this structure (note that it is possible to write other types of container objects which could play the same role simply by overloading the same functions that these objects do).

When an object is "linked" into the object hierarchy, a function called `InitLinks` is called. This function should perform any kind of initialization that depends on the object being situated in the hierarchy, like being able to know what object "owns" this one. **taBase**

has functions for getting and setting the owner of an object. For example, when a group (**taList** or **taGroup** creates a new object and links it into its list of objects, it calls the `SetOwner` function with a pointer to itself on this new object, and then it calls `InitLinks`. Similarly, when the object is removed from the group, the `CutLinks` function is called, which should cut any links that the object has with other objects.

An object's location in the object hierarchy can be represented by a *path* to that object from a global root object. A given application is assumed to have a root object, which contains all other objects. A pointer to that object is kept in **tabMisc::root**, which is used to anchor the path to any given object. An object can find its path with the `GetPath` function, and an object can be found from a path with the `FindFromPath` function.

Finally, a function for allowing an object to set the values of certain members based on changes that might have been made in other members after a user edits the object, called `UpdateAfterEdit`, is provided. This function is called on most objects after they are loaded from a save file (except those with the `#NO_UPDATE_AFTER` comment directive), and on all objects after the user hits *Apply* or *Ok* in an edit dialog, and after any member is set through a CSS assign statement. While the object does not know which members were changed when `UpdateAfterEdit` is called, the object can buffer previous state on its own to figure this out if it is needed.

For a step-by-step guide to making a new class that derives from **taBase**, see Section 18.4 [prog-coding], page 302.

18.2.5 The Dump-file Format for Saving/Loading

The format used for dumping objects to files an loading them back in involves two passes. The first pass lists all of the objects to be saved (i.e., the object that the `Save` function was called on, and any sub-objects it owns. This is done so that during loading, all objects will have been created before pointers to these objects attempt to be cashed out. The second pass then saves all of the values associated with the members in the object. The format is a name-value based one, so that files can be loaded back into objects whose definition has changed. It skips member names it can't find, etc, so you can continue to modify your software and still load old data.

Paths (i.e., the `GetPath` function) figure heavily into the saving of objects, especially pointers. Pointers are saved by giving the path to the object. These saved paths are automatically corrected if the objects are loaded into a different location than the one they were saved in. All pointers that are saved are assumed to be reference-counter based. Thus, the `SetPointer` function is used to set the pointer. Also note that it is impossible to save a pointer to a non-taBase derived object, since there is no way to get the path of such an object.

18.3 Standard TypeAccess Comment Directives

The following sections document comment directives that are recognized by the standard TypeAccess GUI and script-language interfaces. These must be placed in comments immediately following the definition of that which the apply to. Thus, an object directive should appear as

```
class whatever : public something {  // #IGNORE comment goes here
```

for members and methods, it should be as follows:

```
class whatever : public something {  // #IGNORE comment goes here
  int         member_1;       // #HIDDEN comment goes here
  float       member_2;
  // #READ_ONLY or here
  float       get_real();     /* #USE_RVAL note that multi-line
     old-fashioned c-style comments are legal too! */
```

18.3.1 Object Directives

If you add an extra "#" to the beginning of the comment directive, it will automatically be inherited by any sub-classes of the given object. Otherwise, it only applies to the object on which it was given.

`#IGNORE` Do not register this object in the list of types.

`#NO_TOKENS`
: Do not keep a record of the tokens of this object type. Types can keep pointers to all instances or tokens of themselves. This can be expensive in terms of memory, but the interface uses "token menus" for arguements or methods which are pointers to objects of the given type.

`#NO_INSTANCE`
: Do not create a global instance (`TAI_xxx`) of this object. This will prevent tokens of this object from being made.

`#INSTANCE`
: If default is not to create instances, then create one anyway for this object.

`#NO_MEMBERS`
: Do not store the members (including member functions) of this class. Only the type name will be registered.

`#NO_CSS` Do not create CSS stub functions for the member functions on this object.

`#INLINE` Causes this item to be edited in a single line in a dialog box (e.g. for geometry x,y,z) and affects saving/loading, etc.

`#EDIT_INLINE`
: Only causes this item to be edited in a single line in a dialog box, but in all other respects it is treated as a normal included class. This is useful for certain complex objects such as arrays and lists that do not otherwise save/load well as INLINES.

`#BUTROWS_x`
: Set the number of button rows to be x, useful if default allocation of number of rows of buttons for edit dialog is not correct

`#EXT_xxx` Sets the default extension for saving/loading this type to xxx.

`#COMPRESS`
: store dump file's of this object compressed. Since the save files are text, they can be large, so it is a good idea to auto-compress dump files for large objects.

`#MEMB_IN_GPMENU`
: This indicates that there is a group object as a member of this one who's objects should appear in menus where this object appears.

`#VIRT_BASE`
: This is a "virtual" base class: don't show in token menus for this object, etc.

`#NO_UPDATE_AFTER`
: Don't call UpdateAfterEdit when loading this object (and other places it might automatically get called). Since a list of objects which should be updated after loading is made, small or numerous objects should not be added to this list if not necessary.

`#IMMEDIATE_UPDATE`
: Perform an immediate UpdateAfterEdit on this object after loading (i.e., it creates other objects..). Normally, updating happens after all of the other objects have been loaded.

`#SCOPE_xxx`
: Type of object to use as a scope for this object. The scope restricts token menus and other things to only those things that share a common parent token of the given scope type.

`#ARRAY_ALLOC`
: Specific to taList_impl derivatives: this list or group should have saved items created all together during loading (ie., like an array). If actually using array-based memory allocation, this is essential, but otherwise it can only speed things up a little bit.

`#LINK_SAVE`
: Save the actual contents of this object even when it appears as a link in a list. Usually just the link pointer is saved, and the object is saved later in the group that actually owns it. This overrides this and saves the information in both places – can be useful if info from the linked object is needed during loading.

`#NO_OK` Do not present an OK button on the edit dialog for this object.

`#NO_CANCEL`
: Do not present a CANCEL button on the edit dialog for this object.

18.3.2 Member Directives

`#HIDDEN` Hides member from user's view in edit dialogs and CSS type information printouts.

`#HIDDEN_INLINE`
: Hides member when inlined in another object, but not when edited itself. This only applies to members of #INLINE objects.

`#SHOW` Always show this member in the edit dialog (i.e., even if it was marked #READ_ONLY).

`#IGNORE` Does not register this member in the type information for this class.

`#DETAIL` Flags this member as a level of detail that the user usually does not need to deal with — can be viewed by changing the Show setting in the edit dialog.

`#NO_SAVE` This member is not saved when dumping to a file.

`#NO_SAVE_PATH_R`
Don't create these objects in the 1st pass of the dump file (e.g., they will be created automatically by something else, usually an #IMMEDIATE_UPDATE UpdateAfterEdit function on a parent object). This can be used to speed up saving and loading of large numbers of repetitive objects which can be created instead.

`#READ_ONLY`
Allows the user to see but not edit this item. By default the gui edit dialog will not show these items. This prevents the member from being changed in CSS as well.

`#IV_READ_ONLY`
Like READ_ONLY, but user can modify the value via CSS (which is prevented by READ_ONLY).

`#LIST_xxx`
Sets the Lookup List for this element. This is used mainly for pointers to functions, where one wants the gui to show a list of top-level functions that have been scanned by maketa (see Section 18.3.4 [comdir-funs], page 301).

`#TYPE_xxx`
Sets the default type for members which are pointers to TypeDef objects. This also works for MemberDef pointers. If xxx is 'this', then the type of the current object is used.

`#TYPE_ON_xxx`
For object, TypeDef, or MemberDef pointers: use member xxx of this object to anchor the listing of possible types, tokens, or members.

`#FROM_GROUP_xxx`
For token pointers, use given member xxx as the group from which to select token options (xxx can be a pointer to a group).

`#GROUP_OPT_OK`
For FROM_GROUP_xxx mbrs, allows group itself as an option (else not allowed).

`#SUBTYPE_xxx`
Sets this token pointer member to be only subitems (objects owned by this one) of type xxx. A recursive scan of members on this object is performed to search for objects of the given type as possible values for this field.

`#NO_SUBTYPE`
Don't search this ptr for possible subitems (use if this ptr might point "up", causing a endless loop of searching for subitems).

`#NO_FIND` Don't search this member for the recursive FindMembeR function which searches recursively through objects (use if this ptr might point up in the hierarchy, which might cause an endless loop).

`#NO_SCOPE`
: Don't use scope for tokens for a token pointer member. See SCOPE object directive

`#LABEL_xxx`
: Set the label for item (or menu or button) to be xxx.

`#OWN_POINTER`
: For a pointer to an object, when loading, set the owner of the obj to be this object. Thus, this pointer is always created and owned by this object.

`#NULL_OK` A null value is ok as an option for the user (else not) for pointer to a type, and SUBTYPE tokens.

`#NO_NULL` A null value is not ok (for tokens) (else ok).

`#NO_EDIT` Don't include Edit as an option on a token pointer menu (else ok).

`#POS_ONLY`
: Only positive (non-negative) integers, this controls behavior of the stepper for integer types.

`#LINK_GROUP`
: This group member only has linked items (doesn't allow user to create new tokens in this group).

`#IN_GPMENU`
: This members' items should appear in the group menu. The member must be a **taGroup_impl** descendent type, and the class must have a MEMB_IN_GPMENU option set.

`#CONDEDIT_xxx`
: This makes editing a member conditional on the value of another member. For example: #CONDEDIT_OFF_type:NONE,LT_MAX specifies that this member is to be not editable (OFF) when the type enum variable is either NONE or LT_MAX. One alternatively specify ON for conditions when it should be editable. The comparison is based on the string representation of the member value – sub-paths to members within contained objects can also be used.

`#DEF_xxx` Specifies a default value for the member. If the field is set to a value other than this default value, it will be highlighted in yellow to indicate that the value is different from default. This should only be used where there are clear default values that are typically not changed.

`#AKA_xxx` This allows old project files etc to be loaded correctly after changing the name of a field or enum by matching xxx to the new field/enum.

18.3.3 Method Directives

`#MENU` Creates a Menu for this function item in an Edit dialog.

`#MENU_SEP_BEFORE`
: Create a separator before this item in the menu.

`#MENU_SEP_AFTER`
: Create a separator after this item in the menu.

`#MENU_ON_xxx`
: Puts this function on given menu. Creates menu if not already there. This does not replace the #MENU directive. Everything on the File and Edit menus will be on the edit button for this class in an edit dialog.

`#BUTTON` Creates a button for this function in the edit dialog.

`#LABEL_xxx`
: Sets the label for item (or menu or button) to be xxx.

`#USE_RVAL`
: Use (display) return value from this function. Otherwise return values are ignored.

`#USE_RVAL_RMB`
: Use (display) return value from this function only if the right mouse button was pressed on the Ok button. Otherwise return values are ignored.

`#NO_APPLY_BEFORE`
: Do not apply any changes to dialog before calling this function. The default is to apply the changes first.

`#NO_REVERT_AFTER`
: Do not update (revert) dialog after calling this function (and do not call the UpdateAfterEdit function either). The default is to do both.

`#UPDATE_MENUS`
: Update the global menus after calling this function (e.g., because altered the structure reflected by those menus).

`#ARGC_x` How many args to present to the user (if default args are available).

`#ARG_ON_OBJ`
: An argument to this function is an object within the base object (e.g., a member of the group).

`#TYPE_xxx`
: For TypeDef pointer args: use given type to anchor the listing of possible types. if xxx == 'this', then the type of the current object is used.

`#TYPE_ON_xxx`
: For a function with (any) TypeDef or Token args, uses the member xxx of this to anchor type selection or type of tokens to present.

`#FROM_GROUP_xxx`
: Performs selection of tokens for args from given group member xxx, which is a member of this object (like ARG_ON_OBJ). Can also specify which arg(s) this applies to by doing #FROM_GROUP_1_gp: 1 = this arg or below uses from_group, so put your from_group args first and specify the highest index as this.

`#NO_GROUP_OPT`
: For FROM_GROUP_xxx args, disallows group itself as an option.

`#NO_SCOPE`
: Don't scope the argument to this function. See SCOPE object directive

`#NO_SCRIPT`
: Do not generate script code to call this function, if script code recording is currently active. Section 18.3.1 [comdir-objs], page 296.

`#GHOST_ON_xxx`
: For BUTTON meths, ghosts the button based on the value of boolean member xxx of this class. If member == true, button is ghosted.

`#GHOST_OFF_xxx`
: Like above, except if member == false, button is ghosted.

`#CONFIRM` For functions with no args, put up a dialog for confirmation (shows function description too).

`#NEW_FUN` Give user the option to call this (void) function during New (in the "new" dialog).

`#NULL_OK` A null value is ok as an option for the user (else not). for all pointers as args.

`#EDIT_OK` Include Edit as an option on the token pointer menu (else not)

`#FILE_ARG_EDIT`
: For functions with one ostream arg, use the normal arg edit dialog, instead of a shortcut directly to the file chooser (arg edit allows user to choose open mode for saving).

`#QUICK_SAVE`
: For functions with one ostream arg, use existing file name if possible (default is to prompt).

`#APPEND_FILE`
: For functions with one ostream arg, use append as the file opening mode.

18.3.4 Top-Level Function Directives

In addition to class objects and typedef's, it is possible to scan information about certain top-level functions. These functions must be preceded by a `#REG_FUN` comment, and the comments that apply to the function must precede the trailing `;` that ends the function declaration.

```
// #REG_FUN
void Cs_Simple_WtDecay(CsConSpec* spec, CsCon* cn, Unit* ru, Unit* su)
// #LIST_CsConSpec_WtDecay Simple weight decay (subtract decay*wt)
    ; // term here so scanner picks up comment
```

These functions get registered as `static` functions of a mythical object with a **TypeDef** of `TA_taRegFun`. The purpose of registering functions in this way is to make them available for members of classes that are pointers to functions. These registered functions are shown in a menu in the edit dialog if the `#LIST_xxx` directive matches on the registered function and the pointer to a function.

18.3.5 PDP++ Specific Directives

`#NO_VIEW` For real-valued unit members, do not display this item in the net view display.

`#AGGOP_xxx`
(Process object only) sets the default aggregate operator for this process's statistics.

`#FINAL_STAT`
(Stat object only) indicates if this should be created as a final stat.

`#LOOP_STAT`
(Stat object only) indicates if this should be created as a loop stat.

`#COMPUTE_IN_xxx`
(Stat object only) level at which this stat should be computed, xxx is a process type.

`#NO_INHERIT`
In specs, makes member not-inherit from higher-ups.

18.4 Coding Conventions and Standards

This section describes the steps that need to be taken to define a new class. Every class based on the **taBase** type (i.e., all classes in the PDP++ software) needs to have a set of standard methods (member functions) that allow it to interface with the rest of the software. Also, many commonly occurring data types and tasks that a class needs to perform have been dealt with in a standardized way. This chapter familiarizes the programmer with these standards and interfaces.

Defining a new class is typically the first step a user will take in programming with the PDP++ software. This is because the software is designed to be extended, not revised, by the user. Fortunately, most everything that is done by the PDP++ library code can be overwritten by defining a new class that does something differently, or simply by adding on to what the existing code does. Both of these approaches require the definition of a new class.

The first step in defining a new class is figuring out which existing class to base the new one on. This requires a knowledge of the existing class structure, which is covered in this manual. Once this has been decided, the guidelines in this section should be followed as closely as possible. It is assumed that the reader knows C++ and the basic ideas about constructors, destructors, virtual vs. non-virtual functions, etc.

18.4.1 Naming Conventions

The basic tension in naming something is the length vs. descriptiveness tradeoff. In general, we try to avoid abbreviations, but really long words like "environment" inevitably get shortened to "Env", etc.

The bulk of the conventions we have established have to do with distinguishing between different categories of names: class type names, member function names, and member names

being the major categories. In addition, the way in which names composed of multiple words are formed is discussed.

Object Class Names

Class types are first-letter-capitalized with words separated by capitalization: e.g. `MyClass`. There are certain exceptions, where an underbar '_' is used to attach a high-frequency suffix, usually from a template, to the name:

Common Suffixes:

`_List` taList derivative.

`_Group` taGroup derivative.

`_MGroup` MenuGroup derivative.

`_Array` taArray derivative.

`_SPtr` Smart Spec object pointer (PDP++).

Also if the class name contains multiple words, words which are actually acronyms ending with a capital letter are separated from the following word by an '_', e.g., (CE_Stat).

Classes in lower-level libraries also have the name-space identifier prefixed to the name, which is lower case: e.g., `ta`, `taiv`, `css`.

Enums

`enum` type names follow the same naming convention as class types. `enum` members are all upper-case, words are separated by '_', e.g., `INIT_STATE`.

Member Names

Members are lower-case, words are separated by a '_', e.g., member_name. One exception is for names ending in spec or specs in which case there is no separation (e.g., viewspecs).

Method Names

Methods are first-letter-capitalized, words are separated by capitalization (e.g., `RunThis()`). However, there some special prefixes and suffixes that are exceptions to this rule, because they are "high frequency" and denote a whole class of methods:

Prefixes:

`Dump_` Saving and loading functions.

`Compute_` Network computation functions (PDP++).

`Send_` Network communication functions (PDP++).

`C_` C code versions of process functions (PDP++).

`Init_` Special initialize function for processes (PDP++).

Suffixes:

`_impl` Implementation (guts) of some other function which is the one that should be called by the user.

`_xxxx` Other _impl type functions that do specific aspects of the implementation (xxx is lower case). Examples in PDP++ include _flag, _force.

`_` (just a trailing underbar) This is a short version of _impl, which is used extensively in InterViews, and sparingly in TA/PDP++.

`_post` A function which is to be called after another one of the same name (for two-step processes).

`_Copy` A function called after the Copy function (e.g., to clean up pointers).

`_gui` A special GUI version of function call.

`_mc` A special menu callback version of function call.

18.4.2 Basic Functions

These are the functions that must be either specified or considered in any new instance of a **taBase** class:

`void Initialize()`

- It is called in every constructor.
- Do not call `Parent::Initialize()`, as this is a constructor function and the parent's will be called for you by C++ as the object is constructed.
- Set the initial/default values of each member in the class.
- Set the default type for groups that you own (`SetBaseType()`).
- Call `taBase::InitPointer(ptr)` on every taBase object pointer in class, or just set all pointers to NULL.
- EVEN IF NOTHING NEEDS INITIALIZING: use '`void Initialize() { };`' to avoid multiple calls to the parent's Initialize.

`void Destroy()`

- It is called in every destructor.
- Do not call the parent, as C++ will automatically call the parent's destructor for you.
- Free any resources you might have allocated.
- Call `CutLinks()`, if defined, to sever links with other objects.
- EVEN IF NOTHING TO DESTROY: use '`void Destroy() { };`' to avoid multiple calls to parents Destroy.

`void InitLinks()`

- Called when an object is linked into some kind of ownership structure.
- Call the `Parent::InitLinks()`, since this is not a constructor function and the parent's links will not otherwise be set.
- Own any classes contained as members: '`taBase::Own(recv, this);`'

- Set any pointers to objects with default values (e.g., '`spec->SetDefaultSpec(this));`', etc.
- Be sure to use '`taBase::SetPointer(ptr, new_val);`' for setting pointers.
- Or use '`taBase::OwnPointer(ptr, new_val);`' for those you own.
- If you do not need to do any of these InitLinks actions, then you do not need to define an InitLinks function.

`void CutLinks()`

- Called when an object is removed from its owner, or as part of the `Destroy` function when an object is actually deleted, or explicitly by the user when the object is a member of another object.
- At end of `CutLinks()`, call `Parent::CutLinks()`, since this is not always used as a destructor function, and parent's might not be called. Note, however, that when it is called in the destructor, it will be repeatedly called, so it should be robust to this (i.e., SET ANY POINTERS YOU DELETE TO NULL SO YOU DON'T DELETE THEM AGAIN!).
- Should sever all links to other objects, allowing them to be freed too.
- Call `CutLinks()` on any owned members, especially groups!
- Use `taBase::DelPointer()` on any pointers.
- If you have a spec, call `CutLinks()` on it.
- If you have group members, call `CutLinks()` on those groups.

`void Copy_(const T& cp), Copy(const T& cp)`

- Used to duplicate the class, Copy is the = oper and copy constructor
- Call `Parent::Copy` since this will not be called otherwise.
- `Copy_(const T& cp)` is an "implementation" that does the copying for just this class, and does not call the parent `Copy`.
- Use `COPY_FUNS(T, P);` (type and parent-type) to define the default macros for doing this:

```
void Copy(const T& cp)      { P::Copy(cp); Copy_(cp); }
```

- Use `SIMPLE_COPY(T);` to define a `Copy_` function that automatically copies the members unique to this class in a member-by-member (using TypeDef) way. This is less optimal, but easy when members are just simple floats and ints, etc.
- Be sure to use `taBase::SetPointer(&ptr, cp.ptr)` for copying pointers.

`TA_BASEFUNS(T);`

This defines the actual "basic" functions like constructors, destructors, `GetTypeDef()` etc. for taBase classes. These default constructors and destructors call the other functions like `Initialize()` and `Destroy()`.

`TA_CONST_BASEFUNS(T);`

This defines the actual "basic" functions like constructors, etc. for **taBase** classes which have `const` members defined in the class. These need to be initialized as part of the constructor, so this macro leaves out the default constructors and destructor, which should contain the following code:

```
MyClass() { Register(); Initialize(); SetDefaultName(); }
MyClass(const MyClass& cp)
 { Register(); Initialize(); Copy(cp); }
~MyClass() { unRegister(); Destroy(); }
```

`TA_TMPLT_BASEFUNS(y,T);`

Defines the actual "basic" functions like constructors, etc. for **taBase** classes which are also templates. `y` is the template class, `T` is the template class parameter.

`void UpdateAfterEdit()`

- Called after class members change via edit dialogs, loading from a file, or and assign operator in CSS.
- Maintain consistency of member values.
- Update links, etc.

When you add/remove/change any class members:

Check and add/remove/change initialization, copying, of this member in:

`Initialize()`

`Copy_()`

`Copy()`

For classes with Specs:

A pointer to a spec is encapsulated in a SpecPtr template class, which is declared once immediately after a new class of spec types is defined as follows (this will not typically done by the user):

```
SpecPtr_of(UnitSpec);  // this defines the template class
  (only for base spec type)
```

This pointer is then included in the class with the following:

```
UnitSpec_SPtr  spec;  // this puts a spec pointer in the class
```

Also, InitLinks() should have:

```
spec.SetDefaultSpec(this);
```

So that the spec pointer will set its pointer to a default instance of the correct spec type (the `this` pointer is because this also "owns" the spec pointer object.

For classes with taBase members:

All **taBase** members which appear as members of another class should be owned by the parent class. This increments their ref counter, so that if they are ever pointed to by something else (e.g., during loading this happens), and then unref'd, they won't then be deleted.

`InitLinks()` should own the object member as follows:

```
ta_Base::Own(obj_memb, this);
```

For members that derive from **taList** or **taGroup**, `Initialize()` should set the default type of object that goes in the group:

```
gp_obj.SetDefaultType(&TA_typename);
```

Referring to other objects via pointers:

If a class contains a pointer to another object, it should typically refer to that object whenever the pointer is set. The interface assumes that this is the case, and any pointer member that it sets will use the `SetPointer` function described below, which does the referencing of the new value and the dereferencing of the current one.

HOWEVER, when the pointer is to a physical PARENT of the object (or just higher in the deletion hierarchy) then it should not be referenced, as this will prevent the parent from being deleted, which will then prevent the child from being deleted.

In this case, and in general when the pointer is just for "internal use" of the class, and is not to be set by the user, the following comment directives should always be used: `#READ_ONLY #NO_SAVE` as this will prevent the user from overwriting the pointer, and the loading code automatically does a reference when setting a pointer, so these should not be saved. DO NOT COPY SUCH POINTERS, since they typically are set by the `InitLinks` based on the owner, which is usually different for different tokens.

When managing a pointer that the user can set, there are a set of convenient functions in taBase that manage this process (note that the argument is a *pointer* to the pointer):

`taBase::InitPointer(TAPtr* ptr)`
: initializes the pointer (or just set the ptr to NULL yourself) in `Initialize()`

`taBase::SetPointer(TAPtr* ptr, TAPtr new_val)`
: unRef's *ptr obj if non-null, refs the new one.

`taBase::OwnPointer(TAPtr* ptr, TAPtr new_val, TAPtr ownr)`
: like set, but owns the pointer too with the given owner.

`taBase::DelPointer(ptr)`
: unRefDone the object pointed to, sets pointer to NULL.

Using these functions will ensure correct refcounts on objects pointed to, etc.

If you `Own` the object at the pointer, then you should either mark the member as `#NO_SAVE` if it is automatically created, or `#OWN_POINTER` if it is not. This is because saving and loading, being generic, use `SetPointer` unless this comment is present, in which case they use `OwnPointer`.

Using Group Iterators:

There are special iterator functions which iterate through the members of a group. One method is to iterate through those sub-groups (including the 'this' group) which contain actual terminal elements ("leaves"). This is leaf-group iteration. Then, the elements of each group can be traversed simply using the `El` or `FastEl` functions.

- for leaf-group iteration, using macros (preferred method):

```
Con_Group* recv_gp; // the current group
int g;
FOR_ITR_GP(Con_Group, recv_gp, u->recv., g)
  recv_gp->UpdateWeights();
```

- for leaf-group iteration without macros:

```
Con_Group* recv_gp; // the current group
int g;
for(recv_gp = (Con_Group*)u->recv.FirstGp(g); recv_gp;
    recv_gp = (Con_Group*)u->recv.NextGp(g))
  recv_gp->UpdateWeights();
```

When all you care about are the leaf elements themselves, you can iterate over them directly using leaf-iteration:

- for leaf-iteration, using macros (preferred method):

```
Connection* con; // the current leaf
taLeafItr i; // the iterator data
FOR_ITR_EL(Connection, con, u->recv., i)
  con->UpdateWeights();
```

- for leaf-iteration without macros:

```
Connection* con; // the current leaf
taLeafItr i; // the iterator data
for(con = (Connection*)u->recv.FirstEl(i); con;
    con = (Connection*)u->recv.NextEl(i))
  con->UpdateWeights();
```

Appendix A Copyright Information

We would like to acknowledge the contributions in writing the software of David Fogel, Gautam Vallabhajosyula, Alex Holcombe, and the contributions in testing, debugging, and improving the software of Yuko Munakata, Craig Stark, Todd Braver, Jonathan Cohen, Shaun Vecera, Deanna Barch, and the rest of the PDP Lab at CMU. Finally, we would like to thank David Rumelhart for his early involvement in designing the overall shape of the simulator.

Concept Index

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Class Type Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W

X

Z

Variable Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Function Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Short Contents

Table of Contents