SDSC EBSA

C Library Documentation

San Diego Supercomputer Center

Version 2.1

Overview

Simulated annealing (SA) is an optimization procedure derived from the statistical mechanical theory for annealing real substances. In recent years, SA has become a popular method for attacking problems with rough search spaces. It has a likely permanent niche in the arsenal of global optimization due to its computational simplicity.

SDSC EBSA is an **E**nsemble **B**ased **S**imulated **A**nnealing library for both single and multi-processor systems. Our ensemble-based, constant-speed approach is a true analog of annealing theory and compares favorably with other heuristic optimization techniques. Three types of parallelism are exploited for task distribution. The package includes diagnostic profiling to assist with the determination of annealing parameters. At the user's option some initializations can be performed adaptively. A sub-library for visual performance monitoring has been implemented on SGI workstations.

This is a binary library release with example applications and source code provided.

Contents

1 Preliminaries

SDSC EBSA is for global optimization problems which meet the following criteria:

- an analytic solution is not available
- the cost *domain* has many local minima
- a grand tour is untimely
- the "distance" to extrema from the mean is significantly greater than one standard deviation

A famous example is the traveling salesman problem. Other examples include bin packing problems and inverse problems in remote sensing. **Note:** If the last item is not satisfied—i.e. the domain is a flat pock-marked surface, then random search methods would be more appropriate.

Typically, the problem domain is not one of the standard scalar types such as *double* or *int*. We therefore assume that it is best defined by the user and provide a general mechanism for referencing domain elements. We also realize that many users need to use SDSC EBSA as an integral part of some analytic software package while others will be satisfied with the package as a stand alone application. As a result, we have chosen to implement our methods as an ANSI C library but with an OOP design specification. This of course leaves a fair amount of work to be performed by the user. We have tried to alleviate some of this overhead by providing an automated system for generating prototype applications (see section on *ebsagen*) and a visual performance monitor.

1.1 Obtaining the Software

SDSC EBSA is available via anonymous ftp from

```
ftp.sdsc.edu (198.17.47.33)
```

in directory

```
pub/sdsc/math/Ebsa
```

The file `Ebsa.tar` contains a complete set of binaries, this documentation, sample programs, and more. A breakout of this file is also available in smaller machine-specific tar files; e.g., `Ebsa.sunsparc.tar`.

Un-tar the archive file in either your own development area or a system-wide location such as `/usr/local`. Two directory trees will be extracted, `Ebsa` and `ParTools`. The latter is a set of machine independent libraries for global reduction operations and message passing. Partools can also be obtained independently from `ftp.sdsc.edu` in directory `pub/sdsc/parallel/ParTools`.

Future releases of SDSC EBSA will very likely use MPI[1]—eliminating the need for SDSC ParTools.

1.2 Contacting the Author(s)

For information on how to use ftp, tar, or uncompress please contact your system administrator.

For questions, comments, and bug reports pertaining to SDSC EBSA software please contact

Richard Frost
San Diego Supercomputer Center
P.O. Box 85608
San Diego CA 92186-9784
frost@sdsc.edu

[1] a Message-Passing Interface standard under development by a large consortium of academic researchers, engineers in government laboratories, and commercial vendors

2 Getting Started

This section is intended for all readers, especially those who wish to get a quick start on interfacing with the SDSC EBSA library. It also includes reference material on environment variables and architecture dependencies.

2.1 Supported Architectures

SDSC EBSA is currently supported on the following platforms:

- Convex multiprocessor systems (O/S version 10)
- Cray Research YMP and C90 systems (Unicos v.7)
- Digital Equipment Corporation "Decstations" (MIPS, Alpha)
- Hewlett Packard 700/9000 workstations (HP9000 CPU, HPUXA O/S)
- Intel SSD Paragon (Paragon OSF/1 version 1.1+)
- Silicon Graphics workstations (IRIX versions 4+ and 5+)
- Sun Microsystems "Sparcstations" (BSD/Solaris 4.1+)

If your workstation configuration or computational server is not included in this list, please contact the author(s).

2.2 Environment Variables

All of the SDSC EBSA software development tools require two UNIX(tm) environment variables, `EBSA_HOME` and `PARTOOLS_HOME`. These should be set to the directory paths where SDSC EBSA and SDSC ParTools were installed (extracted from the tar file). At SDSC, these are set by the C shell (csh) command:

```
% setenv EBSA_HOME /usr/local/apps/Ebsa
% setenv PARTOOLS_HOME /usr/local/apps/ParTools
```

The directory `$EBSA_HOME/bin` contains Bourne shell scripts used in the SDSC EBSA software development cycle. Add this to your Unix path. The C shell command

```
% set path = ( $path $EBSA_HOME/bin )
```

is one way to do so.

An important script in `$EBSA_HOME/bin` is *machid*. This script was developed at SDSC for runtime determination of machine dependencies. We recommend setting the shell variable MACHID to ‘machid‘. This can be accomplished in the C shell by

```
% setenv MACHID `machid`
```

provided `$EBSA_HOME/bin` has been added to your path.

It might happen that *machid* does not recognize your platform. Please perform the following sanity check (in the C or Bourne shell):

```
% machid
```

If machid returns “unknown” or (worse) a machine type different from the platform it was executed from, please contact the SDSC EBSA author(s).

Some software developers like to add *include* and *lib* directory paths to environment variables such as `LD_LIBRARY_PATH`. SDSC EBSA has a single include file,

```
$EBSA_HOME/include/Ebsa.h.
```

However, it includes ParTools.h found in `$PARTOOLS_HOME/include`.

The SDSC EBSA library object files are placed in

```
$EBSA_HOME/lib/$MACHID/$MACHOS/libEbsa.a
```

or

```
$EBSA_HOME/lib/$MACHID/$MACHOS/GL/libEbsa.a
```

where “`MACHOS`” identifies the machine’s operating system and version. This is set in *remake* by the assignment

```
MACHOS=`machid -n -v | awk -F'.' '{ print $1 }'`
```

Note that GL libraries are only available for SGI systems in this release.

The SDSC ParTools library objects are stored in the same manner. **Note:** the *ebsagen* and *remake* scripts provided with SDSC EBSA use $EBSA_HOME, etc. to describe full path names. Hence, the use of LD_LIBRARY_PATH is not required.

2.3 SDSC EBSA directory structure

The main directory in the SDSC EBSA distribution is organized as follows:

```
$EBSA_HOME/
        README bin/ doc/ example/ include/ lib/
```

As mentioned in the previous section, $EBSA_HOME/bin contains software development tools, including *machid* and *ebsagen*.

$EBSA_HOME/doc contains postscript format documentation (this file, EbsaDoc.ps) and a man subdirectory for Unix(tm) manpages.

$EBSA_HOME/example contains example source code in 5 subdirectories. Four of these (testmain, testio, testscan, testvis) are main programs generated by *ebsagen*. The 5th, liblj, is a sample library of user functions.

$EBSA_HOME/include contains Ebsa.h.

$EBSA_HOME/lib contains subdirectories for each currently supported platform (e.g., $EBSA_HOME/lib/sunsparc). Each of these contains a suitable copy of libEbsa.a. In some cases, additional subdirectories are included to differentiate between incompatible O/S releases (e.g., $EBSA_HOME/lib/sgi4d/IRIX5).

2.4 Application Program Interface

SDSC EBSA's application program interface (API) has 5 types of user-callable subroutines. An overview is presented here. Detailed descriptions appear in later sections of the manual.

2.4.1 Data Allocation & Construction

These routines build C structures from (pointers to) user data. The structures are then used by SDSC EBSA to perform the actual annealing task(s). They are order dependent.

Most applications call these routines only once. However, some need to repeatedly call SDSC EBSA for the same cost function and move-class but different data (i.e., material properties). These applications might only call EbsaArgCtor and EbsaFuncCtor once, but repeatedly call EbsaDataCtor for each new condition.

EbsaArgCtor – Creates a structure of run-time parameters from keywords specified on the command line. Includes built-in help facility.

EbsaFuncCtor – Creates a structure of pointers to user functions. These include constructors and destructors for states in the user's cost domain; a cost function and move class; copy, print, and scan functions for user state elements.

EbsaDataCtor – Creates a structure containing (pointers to) an ensemble of user-defined states, history tables, and other data necessary for the annealing computation.

EbsaVisCtor – This constructor is only necessary when the visual performance monitor is needed. It does little when visualization is not requested.

2.4.2 Annealing

There are two forms of the computational annealing subroutine. The first is for runs on generic hardware, while the second causes hardware dependent graphics libraries to be loaded.

Ebsa – Perform EBSA using static and runtime parameters initialized by the constructors.

EbsaVis – Also display a visual performance monitor on the users console. EbsaVis calls Ebsa.

2.4.3 Device and File I/O

All of SDSC EBSA's I/O routines currently use *fprintf* and *fscanf* to read and write data. Although this can sometimes result in large data files, we prefer this choice to the evils of inter-machine binary translations. Additional routines for binary I/O are planned for a future version of SDSC EBSA.

EbsaArgPrint – Prints a formatted file of EbsaArg parameters to a file.

EbsaDataPrint – Prints an EbsaDataStruct to a formatted file.

EbsaArgScan – Scans (inputs) a formatted file of EbsaArg parameters.

EbsaDataScan – Scans in an EbsaDataStruct from a disk file.

xMinEcho – Echoes data about the very best minimum found so far to stderr.

xMinPrint – Prints data about the very best minimum found so far to a user file.

TschedPrint – Prints out the temperature history to an ascii stream or file for reuse.

2.4.4 Data Deallocation & Destruction

For each constructor there is a destructor (e.g., EbsaArgDtor) that deallocates memory associated with a C structure. They all have the form Ebsa*Dtor. The pointer argument will be `NULL` after invocation.

2.4.5 Miscellaneous Services

SDSC EBSA also provides a few miscellaneous data services for users who might not have suitable tools readily available.

EbsaRand – A robust uniform pseudo random number generator.

EbsaRandReset – Resets EbsaRand to a user-selectable value. Allows re-generation of a pseudo random sequence.

2.5 Example main program

SDSC EBSA contains several example programs in the directory `EBSA_HOME/example/`. Program `example1` is a brief demonstration program of the SDSC EBSA libraries. To examine this software, first copy the directory into your personal development area. In the C shell, this could be accomplished by

```
% cd $MY_DIR            <--- wherever your development area is
% cp -r $EBSA_HOME/example/example1 .
```

Alternately, if your O/S does not support `cp -r`, try

```
% cd $EBSA_HOME/example
% tar cvf $MY_DIR/example1.tar example1
% cd $MY_DIR
% tar xvf example1.tar
% rm example1.tar
```

The environment variable `$EBSA_HOME` is discussed in section 2.2 of this document.

Once you have successfully copied the example, change to the top level directory and make sure the executable is current.

```
% cd example1
% remake
```

Remake (section 7.2) will either recompile the file, or report that it is up to date. Now execute the file, using the `bin/` directory path shown in the remake output. Use the following flags:

```
% bin/sgi4d/IRIX4/example1 -N 200 -MaxSteps 10
```

(Here we suppose that you are running IRIX4 on an SGI platform.) The program will run an extremely short time on a Leonard-Jones problem. Near the program end, data about the very best so far minimum will be echoed to the terminal. It will look something like:

```
xMinEcho listing:

Emin:                    -3.93961
EminID:                  0
EminSteps:               9

xMin:
3.60735    3.43507    -0.234499
0.259667   2.52367    -3.07594
2.22009    2.79617    -0.939162
...


...
```

For a brief online description of the fields echoed and SDSC EBSA's command line flags, enter:

```
% bin/sgi4d/IRIX4/example1 -help
```

using whatever path is appropriate. A more detailed description of command line flags is given in sections 3.1 and 4.1.1.

For veteran C programmers, the source file `src/example1.c` (2.5.2) should be a good starting point on how to interface with the SDSC EBSA libraries.

2.5.1 More about `example1.c`

This program performs only 5 functions:

- parses SDSC EBSA flags on command line
- receives description of users problem space via constants and function pointers
- allocates space for optimization procedure
- performs ensemble-based simulated annealing
- frees allocated memory

These tasks are performed by:

- **EbsaArgCtor**
- **EbsaFuncCtor** and user defined values
- **EbsaDataCtor**
- **Ebsa**
- **EbsaDataDtor, EbsaFuncDtor, EbsaArgDtor**

Only minimal output of user data is performed by this example, so it is not of much computational interest. However, the programmer might take interest in the include file `lj.h` and library `liblj.a` which are given in section 6.3.

The remainder of this document discusses the SDSC EBSA library and implementation specifics in detail. Output of user data is discussed in section 5.1. EbsaFuncCtor() is introduced in section 4.1.2. Details on the construction of user modules are given in section 6.

2.5.2 `example1.c` source listing

```
/**
 **  FILE
 **       example1.c                generic version
 **
 **  PROJECT
 **       SDSC EBSA
 **
 **  DESCRIPTION
 **       Sample program demonstrating SDSC EBSA library
 **
 **/

#include "lj.h"
#include "Ebsa.h"


/*
 *  FUNCTION
 *       main
 */

int                                     /* return type for main */
main(
        int argc,                       /* arg count from shell */
        char *argv[]                    /* command line args from shell */
        )
{
        EbsaArgStruct *argStruct;       /* defined in Ebsa.h */
        EbsaFuncStruct *funcStruct;
        EbsaDataStruct *dataStruct;

        long steps;                     /* steps returned by Ebsa() */


        /* Parses command line */

        argStruct = EbsaArgCtor( argc, argv );

        if ( argStruct == ((EbsaArgStruct *) NULL) )
        {
```

```
    fprintf(stderr, "example1: bad pointer from EbsaArgCtor\n");
    exit(1);
}


/* incorporates user-defined functions (in this case lj) */

funcStruct = EbsaFuncCtor (
        argStruct,
        (double (*) (void*)) (LjPot),
        LJ_PARTICLES,
        (void * (*) (void)) (LjCtor),
        (void (*) (void *)) (LjDtor),
        (void (*) (void *, void *)) (LjCopy),
        (void (*) (void *, void *)) (LjMove),
        (void (*) (FILE *, void *)) (LjPrint),
        (void (*) (FILE *, void *)) (LjScan),
        (void **) NULL,
        (double *) NULL
        );

if ( funcStruct == ((EbsaFuncStruct *) NULL) )
{
    fprintf(stderr, "example1: bad pointer from EbsaFuncCtor\n");
    exit(1);
}

dataStruct = EbsaDataCtor ( argStruct, funcStruct );

if ( dataStruct == ((EbsaDataStruct *) NULL) )
{
    fprintf(stderr, "example1: bad pointer from EbsaDataCtor\n");
    exit(1);
}


/* Do annealing */

steps = Ebsa (dataStruct);


/* Echo result */
```

```
        xMinEcho(dataStruct);


        /* Clean up ! */

        EbsaDataDtor (dataStruct );
        EbsaFuncDtor (funcStruct );
        EbsaArgDtor ( argStruct );

        exit(0);
}
```

3 Using SDSC EBSA

In this section we discuss the use of programs written with the SDSC EBSA libraries. We begin by explaining the use of command line flags—which are common to both batch and interactive execution. This is followed by an explanation of fields output from EBSA's "Echo" routines. An introduction to the visual diagnostic system is also given, along with a complete example program.

3.1 Command line flags

There are 15 possible command line flags in the current version. They are primarily used to supply run-time parameters to EBSA. In most cases, researchers will want to experiment with some, if not all of these values. Note that the first flag, `-MaxSteps`, is required:

`-MaxSteps` *steps*
: The maximum number of thermodynamic steps to take, including warmup. For test runs, this is on the order of 1000. Productions runs might require over 100K, especially if a noise temperature or other limit criteria are sought. The warmup phase usually requires less that 100 steps. *Steps* must be a positive integer. This flag is *required* because there is currently no other way to guarantee job completion.

`-N` *total-ensemble-size*
: The total number of states in the ensemble. The argument must be an integer greater than 1. For production runs, this value must be large enough to insure a reasonable sample of the density of states. The default value is currently 128. In future versions, it could be adjusted adaptively. On parallel platforms the states will be efficiently distributed across all processors. The decomposition algorithm choses a processors to have m states each, and b processors to have $m+1$ states each so that the total $N = a * m \ + \ b * (m+1)$.

`-T0` *temperature*
: With experience, some researchers will be able to specify an initial temperature—rather than having SDSC EBSA compute one adaptively. This value must be positive.

`-v` *thermo-speed*
: At the core of EBSA, the temperature is updated according to the relationship between current statistics and a target mean temperature $\overline{E_{i+1}}^*$. This value is usually calculated to be v standard deviations from the current mean. The parameter v is referred to as the thermo-dynamic speed. The default value is currently 0.01.

`-Method` *type*
: Used to specify the temperature update method. *type* must be one of:

"capacity" Use heat capacity method (default). This is sometimes referred to as the "Florida" implementation. Target energy changes are assumed to be a direct function of heat capacity, therefore yielding explicit target temperatures.

"gibbs" Use Gibbs free energy method. Sometimes referred to as "Copenhagen" implementation. A bit slower than the heat capacity method, but yields more accurate temperature profiles.

"fixed" Keep the temperature constant

`-WarmUpCriteria` *type*
: Specifies the type of warmup procedure. *type* must be one of:

"Tfrac" (default) warmup stops when

$$Tfrac \equiv \frac{\sigma(E)}{T} \leq EbsaWarmUpLimit$$

for $2 \cdot neighbors$ steps[34].

"activity" warmup stops when

$$Eactivity \equiv \frac{\overline{\delta E_i} - \overline{\delta E_{i-1}}}{\overline{\delta E_{i-1}}} \leq EbsaWarmUpLimit$$

for $2 \cdot neighbors$ steps.

`-WarmUpLimit` *EbsaWarmUpLimit*
: Limit (if any) for warmup criteria. The default value is currently 0.05, corresponding with the default WarmUpCriteria of *Tfrac*. A trial value of 0.0003 might be more suitable when WarmUpCriteria = *activity*.

`-HaltCriteria` *type*
: Used to specify the type of exit criteria used; i.e., the criteria for halting the annealing procedure. Note however that EBSA will always halt when steps = MaxSteps. The options here therefore specify conditions under which Ebsa can halt earlier. *type* must be one of:

"steps" (default) stop at MaxSteps

"trapped" stop when

$$Eactivity \equiv \frac{\overline{\delta E_i} - \overline{\delta E_{i-1}}}{\overline{\delta E_{i-1}}} \leq EbsaHaltLimit$$

for $2 \cdot neighbors$ steps

"temperature" stop when

$$T \leq EbsaHaltLimit$$

for $2 \cdot neighbors$ steps

“noise” stop when

$$dEnorm \equiv \max\{\mid \delta E_i \mid\} \leq EbsaHaltLimit$$

for $2 \cdot neighbors$ steps

“value” stop when

$$Emin \equiv \min\{E_i\} \leq EbsaHaltLimit$$

“uphill” stop when

$$uphill \equiv \%\ of\ uphill\ moves \leq EbsaHaltLimit$$

`-HaltLimit` *EbsaHaltLimit*

Limit (if any) for exit criteria. The default value is currently 0, corresponding with the default HaltCriteria of *steps*.

`-visFlag` *type*

Specify type of visualization to employ. Note that the user code must invoke EbsaVisCtor() and EbsaVis() on a supported architecture. Only “off” and “gamut” are supported at this time. *type* must be one of:

“off” (default) turn visualization off

“gamut” display all vis windows

“states” display state diagram only

“Ehist” display E distribution only

“track” perform 2D color contour track

“trace” perform 2D color contour trace

“volume” compute volume vis data

`-Ebins` *Ebins*

\# of bins to use in visualization energy histograms

`-xBins` *xBins*

\# of bins in visualization state space axis

`-ioLabel` *string*

I/O device or file label (31 chars max) This option provides an easy way for the user to pass a string at the command line into their program. The characters are stored in the *ioLabel* record of the structure created by EbsaArgCtor().

`-profiling`

Turn on optimization profiling. This causes EBSA to store historical data (e.g., T_i) and statistics related to the annealing run. Visualization implies profiling. Note that profiling must be turned on when data written by EbsaDataPrint() is to be recaptured by a user program calling EbsaVis().

`-verbose`

Turn on verbose comments. Causes statistics to be echoed to stderr.

`-help`

Echoes help message to stderr.

3.2 Ascii Debug

During the course of development, it is sometimes useful to examine the contents of structures, etc. to verify correct initialization and program behavior. SDSC EBSA's echo routines EbsaArgEcho(), EbsaFuncEcho(), EbsaDataEcho(), and EbsaVisEcho() provide a mechanism for outputing the non-function valued members of EBSA data structures. These routines are automatically invoked by EBSA's `-verbose` flag, so the user rarely needs to insert them directly in their code. Use of `-verbose` will cause output at the close of each *Ctor() routine and the close of either Ebsa() or EbsaVis().

Some of EBSA's structures contain flag variables whose settings are stored as integers instead of strings. A listing of flags and their values is given in Table 1. Undefined states are usually assigned the value -1.

3.3 EbsaVis interface

If `-visflag gamut` is specified and the program is executing on a supported graphics architecture, EbsaVis() will launch up to 12 graphic objects on the user's workstation. A simple dialog bar will also be drawn at the bottom of the screen alerting the user to available commands.

The default display contains 12 borderless plot windows. They are spatially arranged as:

$$\begin{array}{cccc} T(t) & E(w) & s(t)/T(t) & \overline{E}(T) \\ c(t) & move\%(T) & xNorm & s(T) \\ E(t) & Boltz(T) & xGeom & c(T) \end{array}$$

By pressing the space bar followed by a function key during execution, the user can select any one of these plots for an enlarged solo display. Pressing "g" while in command mode (space bar) will return to the default *gamut* display. Function keys are mapped to the plots as follows:

F1	F2	F3	F4
F5	F6	F7	F8
F9	F10	F11	F12

FLAG	VALUE	LABEL
`Ebsa_Method`		
	0	capacity
	1	gibbs
	2	fixed
`Ebsa_WarmUpCriteria`		
	0	Tfrac
	1	activity
`Ebsa_HaltCriteria`		
	0	trapped
	1	steps
	2	temperature
	3	noise
	4	value
	5	uphill
`Ebsa_visFlag`		
	0	off
	1	gamut
	2	states
	3	Ehist
	4	track
	5	trace
	6	volume
`misc.`		
	0	`Ebsa_FALSE`
	1	`Ebsa_TRUE`

Table 1: Table of SDSC EBSA flag values

The visualization diagonosic display is particularly useful for analyzing the efficiency of move class designs and for aiding the selection of run-time parameters. A brief description of each plot type follows—an extended discusion can be found in the References; e.g. [34, 22, 26, 12, 24, 36, 30].

Description of EbsaVis Plots

$T(t)$	Temperature as a function of time step
$c(t)$	Heat capacity as a function of time step
$E(t)$	Mean Energy, Target Mean Energy, and Best-So-Far Energy (cost evaluation) as a function of time step
$E(w)$	Energy histogram as a function of temperature bins
$move\%(T)$	Percentage of uphill and downhill moves attempted as a function of temperature
$Boltz(T)$	Boltzmann move probability as a function of temperature
$s(t)/T(t)$	The temperature fraction ($Tfrac$) as a function of time step
$xGeom$	The user's `xGeom` graphic
$xNorm$	The normed state space display from the user's `xNorm` function
$\overline{E}(T)$	Mean energy as a function of temperature [34]
$s(T)$	standard deviation as a function of temperature
$c(T)$	Heat capacity as a function of temperature

3.4 Example multi-purpose program

The following program uses the library of section 6 to demonstrate data I/O and optional visualization diagnostics in either batch or interactive environments.

3.4.1 example2.c

```
/**
 **  FILE
 **     example2.c
 **
 **  PROJECT
 **     SDSC EBSA
 **
 **  DESCRIPTION
 **     Sample program demonstrating SDSC EBSA library
 **
```

```
 **     General purpose program for profiling in batch or
 **     interactive mode, or for making production runs.
 **
 **     Flags supported by example2:
 **
 **         -ioLabel "string"   prefix label for input/output files
 **                             Default is name of executable.
 **
 **         -profiling          save EbsaArg data in ioLabel.arg
 **                             and EbsaData statistics in ioLabel.data
 **
 **         -scan               scan EbsaArg data from ioLabel.arg
 **                             and EbsaData from ioLabel.data
 **
 **         -visFlag            Enable EbsaVis routine
 **
 **         -xMinOut            output global minimum to ioLabel.xMin
 **
 **     All EbsaArgCtor flags also supported by Ebsa.
 **
 **
 **     Example usage:
 **
 **         generate a data set in production run for later viewing:
 **
 **           example2 -N 4200 -MaxSteps 10000 -profiling -ioLabel "bigrun"
 **
 **         scan and visualize data set from production run:
 **
 **           example2 -scan -ioLabel "bigrun" -visflag gamut
 **
 **         perform a small run and output global min to example2.xMin
 **
 **           example2 -N 64 -MaxSteps 128 -xMinOut
 **
 **/

#include "Ebsa.h"
#include "lj.h"


/*
```

```
 *  FUNCTION
 *
 *     main
 *
 */

int                     /* return type for main */
main(
        int argc,       /* arg count from shell */
        char *argv[]    /* command line args from shell */
        )
{
        EbsaArgStruct *argStruct;            /* defined in Ebsa.h */
        EbsaFuncStruct *funcStruct;
        EbsaDataStruct *dataStruct;
#ifdef LINKLIB_GL
        EbsaVisStruct *visStruct;
#endif
        long steps;                          /* steps taken by Ebsa */

        long i;                              /* iterate */
        long ioLabelFlag;                    /* flag -ioLabel */
        long profilingFlag;                  /* flag -profiling */
        long scanFlag;                       /* flag -scan */
        long visFlag;                        /* flag -visFlag */
        long xMinOutFlag;                    /* flag -xMinOut */

        char ioLabel[1000];                  /* ioLabel string */
        char argFile[1024];                  /* file names */
        char dataFile[1024];
        char xMinFile[1024];
        char *charPtr;                       /* index */

        /* parse command line for flags */

        ioLabel[0] = '\0';
        profilingFlag = Ebsa_FALSE;
        scanFlag = Ebsa_FALSE;
        visFlag = Ebsa_FALSE;
        xMinOutFlag = Ebsa_FALSE;
        i = 1;
        while ( (i < argc) && (
```

```
                (! scanFlag) ||
                (! visFlag) ||
                (! profilingFlag) ||
                (! xMinOutFlag)
                ))
{
        scanFlag = (scanFlag || (strcmp("-scan",argv[i]) == 0));
        visFlag = (visFlag || (strcmp("-visFlag",argv[i]) == 0));
        profilingFlag = (profilingFlag ||
                        (strcmp("-profiling",argv[i]) == 0));
        xMinOutFlag = (xMinOutFlag ||
                        (strcmp("-xMinOut",argv[i]) == 0));
        i++;
}


if ( scanFlag && profilingFlag )       /* mutually exclusive */
{
fprintf(stderr,
        "example2: -scan and -profiling are mutually exclusive\n");
exit(1);
}


/* if performing I/O, initialize file names */

if (scanFlag || profilingFlag || xMinOutFlag)
{
        ioLabelFlag = Ebsa_FALSE;
        i = 1;
        while ( (i < argc) && (! ioLabelFlag) )
        {
                if (strcmp("-ioLabel",argv[i])==0)
                {
                        ioLabelFlag = Ebsa_TRUE;
                        strncpy((char *) ioLabel, argv[i+1], 998);
                        ioLabel[999] = '\0';
                }
                i++;
        }
        if (! ioLabelFlag)
        {
```

```
                charPtr = argv[0] + strlen(argv[0]);
                while ((*(charPtr - 1) != '/') &&
                                (charPtr > argv[0] ))
                        charPtr--;
                strcpy((char *) ioLabel, charPtr);
                fprintf(stderr,
                        "example2: ioLabel = '%s'\n", ioLabel);
        }

        strcpy(argFile, ioLabel);
        strcat(argFile, ".arg");

        strcpy(dataFile, ioLabel);
        strcat(dataFile, ".data");

        strcpy(xMinFile, ioLabel);
        strcat(xMinFile, ".xMin");
}


/* initialization */

if (scanFlag)
        argStruct = EbsaArgScan( argFile );
else
        argStruct = EbsaArgCtor( argc, argv );

if ( argStruct == ((EbsaArgStruct *) NULL) )
        {
        fprintf(stderr,
                "example2: bad pointer from EbsaArgCtor\n");
        exit(1);
        }

funcStruct = EbsaFuncCtor (
        argStruct,
        (double (*) (void*)) (LjPot),
        LJ_PARTICLES,
        (void * (*) (void)) (LjCtor),
        (void (*) (void *)) (LjDtor),
        (void (*) (void *, void *)) (LjCopy),
        (void (*) (void *, void *)) (LjMove),
```

```
                (void (*) (FILE *, void *)) (LjPrint),
                (void (*) (FILE *, void *)) (LjScan),
                (void **) NULL,
                (double *) NULL
                );

        if ( funcStruct == ((EbsaFuncStruct *) NULL) )
        {
                fprintf(stderr,
                        "example2: bad pointer from EbsaFuncCtor\n");
                exit(1);
        }

        if (scanFlag)
                dataStruct =
                        EbsaDataScan( dataFile, argStruct, funcStruct );
        else
                dataStruct = EbsaDataCtor( argStruct, funcStruct );

        if ( dataStruct == ((EbsaDataStruct *) NULL) )
        {
                fprintf(stderr,
                        "example2: bad pointer from EbsaDataCtor\n");
                exit(1);
        }


        /* Annealing */

#ifndef LINKLIB_GL

        steps = Ebsa(dataStruct);

#else
        if (visFlag)
        {
                visStruct = EbsaVisCtor (
                        argStruct,                /* from EbsaArg* */
                        (long) LJ_NORM1DIM,       /* xNormDim */
                        (double *) LjNorm1Bounds, /* xNormBounds */
                        (double * (*) (void *)) (LjNorm1), /* xNorm */
                        (long ) LJ_GEOMDIM,                /* xGeomDim */
```

```
                        (void (*) (void *)) (LjGeom)       /* xGeom */
                        );

                if ( visStruct == ((EbsaVisStruct *) NULL) )
                {
                        fprintf(stderr,
                            "example2: bad pointer from EbsaVisCtor\n");
                        exit(1);
                }

                steps = EbsaVis(dataStruct, visStruct);
        }
        else
                steps = Ebsa(dataStruct);
#endif


        if (profilingFlag)
        {
                EbsaArgPrint( argFile, argStruct );
                EbsaDataPrint( dataFile, dataStruct );
        }

        if (xMinOutFlag)
                xMinPrint( xMinFile , dataStruct );

        EbsaArgDtor ( argStruct );
        EbsaFuncDtor (funcStruct );
        EbsaDataDtor (dataStruct );

#ifdef LINKLIB_GL
        EbsaVisDtor (visStruct );
#endif

        exit(0);
}
```

4 Basic API modules

SDSC EBSA's application program interface (API) has 5 types of user-callable subroutines. An overview is presented in section 2.4. Here we examine the 3 main types which are common to all applications.

4.1 Data Allocation & Construction

These routines build C structures from (pointers to) user data. The structures are then used by SDSC EBSA to perform the actual annealing task(s). They are order dependent.

Most applications call these routines only once. However, some need to repeatedly call SDSC EBSA for the same cost function and move-class but different data (i.e., material properties). These applications might only call EbsaArgCtor and EbsaFuncCtor once, but repeatedly call EbsaDataCtor for each new condition.

4.1.1 EbsaArgCtor

EbsaArgCtor is used to parse the user's command line for SDSC EBSA run-time parameters. This is the preferred way to initialize these values since most user's will want to perform many experiments. As an alternative, the user can define a string of command line arguments in their program; e.g.,

```
char *myArgs = "-MaxSteps 40000 -N 2300 -verbose";
long myArgCount = 5;
```

then pass these arguments to EbsaArgCtor().

In order to capture the command line, the user needs to declare their main program as follows:

```
int                                     /* return type for main */
main(
        int argc,                       /* arg count from shell */
        char *argv[]                    /* command line args from shell */
        )
```

As shown in `example1.c` (section 2.5.2), an EbsaArgStruct pointer must also be declared prior to calling EbsaArgCtor. Here is the function prototype:

```
EbsaArgStruct *         /* return type for EbsaArgCtor */
EbsaArgCtor(
        int argc,       /* arg count from shell */
        char *argv[]    /* command line args from shell */
        );
```

When called by the user's program, EbsaArgCtor will respond to Ebsa flags, but leave any other user flags untouched. The flags recognized by EbsaArgCtor are:

```
        -MaxSteps           the maximum number of steps (required)
        -N                  desired number of random walkers
        -T0                 initial temperature
        -v                  constant thermodynamic speed
        -Method             type of temperature update method

        -WarmUpCriteria     type of warmup criteria
        -WarmUpLimit        limit (if any) for warmup criteria
        -HaltCriteria       type of exit criteria
        -HaltLimit          limit (if any) for exit criteria

        -visFlag            type of visualization (if any)
        -Ebins              # of bins in vis energy histograms
        -xBins              # of bins in vis state space axis

        -ioLabel            I/O device or file label (31 chars max)

        -profiling          turn on optimization profiling
        -verbose            turn on verbose comments
        -help               echo help message to stderr
```

For information on using SDSC EBSA flags, see section 3.1.

4.1.2 EbsaFuncCtor

This is function is used to describe the optimization problem to Ebsa. For input, it requires an EbsaArgStruct pointer plus several user defined values and function pointers. For more information on how to construct these functions, see section 6.1. The function prototype is:

```
EbsaFuncStruct *                /* return type for EbsaFuncCtor */
EbsaFuncCtor(
```

```
/*
 * These arguments are required.
 */
EbsaArgStruct *argStruct, /* constructed by EbsaArgCtor */
double (*f)(void *x),     /* Objective function to be minimized.
                             Must return type double */
long neighbors,           /* # of neigbors per state--or at
                                least the order or maximum
                                number of neighbors */
void * (*xCtor)( void ),  /* Allocates memory for an element of
                                domain(f), initializes the element
                                to a random state, and returns a
                                pointer to this data.  Should
                                return NULL on failure. */
/* Note: use the supplied function EbsaRand to generate
           psuedo-random values          */
void (*xDtor)(void *x),         /* De-allocates all memory
                                   associated with x */
void (*xCopy)(void *x,void *y), /* Domain copy function:
                                   copies the contents of *x into
                                   previously allocated *y. */
void (*xMove)(void *x,void *y), /* Domain permutation function:
                                   User passes pointer to element
                                   *x in domain of f to
                                   (*xMove)(x,y) and a psuedo-
                                   random neighbor of *x is
                                   returned in *y.  Note: storage
                                   for *y must be pre-allocated */
void (*xPrint)(FILE *fp, void *x),
                                /* Prints user *x and a trailing
                                   white-space character to an
                                   ascii stream (e.g. \\t, \\n). */
void (*xScan)(FILE *fp, void *x),
                                /* When positioned at beginning of
                                   string generated by xPrint,
                                   xScan scans in a user *x value
                                   (incl. trailing white space). */
/*
 * These may be NULL or user supplied values.  If NULL,
 * they will be automatically initialized.  If non-null,
 * they must be valid pointers.
 */
```

```
        void **x0,   /* M states (elements of domain) i.e., x0[i] points
                        to the domain element assigned to walker i. */
        double *E0   /* energy of M states */
        );
```

4.1.3 EbsaDataCtor

EbsaDataCtor creates a structure containing (pointers to) an ensemble of user-defined states, history tables, and other data necessary for the annealing computation. It requires the structures created by EbsaArgCtor and EbsaFuncCtor for input:

```
EbsaDataStruct *                    /* return type for EbsaDataCtor */
EbsaDataCtor(
        EbsaArgStruct *argStruct,    /* constructed by EbsaArgCtor */
        EbsaFuncStruct *funcStruct   /* constructed by EbsaFuncCtor */
        );
```

4.1.4 EbsaVisCtor

This constructor creates and initializes data for EbsaVis, the visual performance monitor. It does little when visualization is not requested on the command line.

```
EbsaVisStruct *                     /* return type for EbsaVisCtor */
EbsaVisCtor(
        EbsaArgStruct *argStruct,    /* constructed by EbsaArgCtor */

        long xNormDim,               /* Dimension of xNorm.  Can be 1, 2,
                                        or NULL to disable */
        double *xNormBounds,         /* xNormBounds[] is NULL or contains
                                        pairs of type double describing
                                        the min and max value for each
                                        dimension of xNorm */
        double * (*xNorm)(void *x),  /* Converts user x into double
                                        vector of length xNormDim
                                        for plot of E(x).  The user
                                        should supply a valid function
                                        pointer or NULL */

        /*
```

```
 * Function required for state geometry display.
 * For a given element of the domain (void pointer), the
 * user supplies a graphics function which draws either
 * a 2 or 3 dimensional (xGeomDim) model in a window
 * controlled by EbsaVis.  The user must shift the actual
 * model coordinates within a unit square (2d) or cube (3d)
 * anchored on the origin and return the actual coordinates
 * (#xGeomDim min/max pairs) in a type double array.  EbsaVis
 * will draw an axis based upon the returned values and
 * permit rotation and zooming on the model.
 *
 * On SGI systems, the function should be written in GL.
 *
 * If either are NULL, then this plot type will be disabled.
 */

long xGeomDim,               /* Dimension of xGeom.  Can be 2, 3,
                                or NULL to disable */
void (*xGeom)(void *x)       /* Draws a model of in unit
                               coordinates anchored on the
                               origin. */

);
```

4.2 Annealing

There are two forms of computational annealing subroutine. The first is for production runs, while the second invokes the visual performance monitor.

4.2.1 Ebsa

For production runs, Ebsa() is the most efficient algorithm to call. Although EbsaVis has little overhead when visualization is not requested, runs which require 10^4 or more steps could experience serious latencies.

```
long                                /* return type for Ebsa */
Ebsa(
        EbsaDataStruct *dataStruct  /* annealing parameters */
        );
```

4.2.2 EbsaVis

EbsaVis causes hardware dependent graphics libraries to be loaded for the display of a visual performance monitor on the users console. EbsaVis calls Ebsa.

```
long                                /* return type for EbsaVis */
EbsaVis(
        EbsaDataStruct *dataStruct,  /* annealing parameters */
        EbsaVisStruct *visStruct     /* vis structure */
        );
```

4.3 Data Deallocation & Destruction

For each constructor there is a destructor (e.g., EbsaArgDtor) that deallocates memory associated with a C structure. They all have the form Ebsa*Dtor. The pointer argument will be `NULL` after invocation.

```
void                              /* return type for EbsaArgDtor */
EbsaArgDtor(
        EbsaArgStruct *argStruct   /* struct to deallocate */
        );
```

```
void                                /* return type for EbsaFuncDtor */
EbsaFuncDtor(
        EbsaFuncStruct *funcStruct   /* struct to deallocate */
        );
```

```
void                                /* return type for EbsaDataDtor */
EbsaDataDtor(
        EbsaDataStruct *dataStruct   /* structure to be freed */
        );
```

```
void                               /* return type for EbsaVisDtor */
EbsaVisDtor(
        EbsaVisStruct *visStruct    /* structure to be freed */
        );
```

5 Additional modules

Most users want to examine more than just the very best minimum, in fact they want to examine the entire spectrum of computed data. Investigators using computational servers without visualization capabilities not only want to output their data, but read it back in on a graphics workstation for examination. The I/O routines listed here are designed to assist with these matters. All SDSC EBSA I/O routines are architecture independent.

5.1 Data output

There are currently two output methods: Echo and Print. Both translate all or part of an SDSC EBSA data structure to an ascii stream or file.

The Echo routines send their data to stderr. They are primarly for interactive use. For example, the `-verbose` flag (3.1) enables Echo routines in various SDSC EBSA library calls.

The Print routines send output to a user-specified file. They provide the best data for analysis of SDSC EBSA computational runs. Also, the output format of each Print call is designed for re-input by SDSC EBSA's "Scan" routines.

5.1.1 EbsaDataPrint

EbsaDataPrint sends the data content of an EbsaDataStruct to a user file. If you plan on re-scanning this data, EbsaArgPrint must also be used to save the command line parameters that created this structure, and EbsaFuncCtor must be called to initialize the correct user data functions (see example2, section 3.4.1).

```
void                                /* return type for EbsaDataPrint */
EbsaDataPrint(
        char *filename,             /* ascii stream for output */
        EbsaDataStruct *dataStruct  /* structure to print */
        );
```

5.1.2 EbsaArgPrint

EbsaArgPrint prints the entire contents of an EbsaArgStruct to a user file.

```
void                                /* return type for EbsaArgPrint */
```

```
EbsaArgPrint(
        char *filename,             /* ascii stream for output */
        EbsaArgStruct *argStruct    /* structure to print */
        );
```

5.1.3 EbsaDataEcho

EbsaDataEcho echoes most entries of an EbsaArgStruct to stderr.

```
void                                /* return type for EbsaDataEcho */
EbsaDataEcho(
        EbsaDataStruct *dataStruct  /* structure to print */
        );
```

5.1.4 EbsaArgEcho

EbsaArgEcho echoes the entire contents of an EbsaArgStruct to stderr.

```
void                              /* no return type for EbsaArgEcho */
EbsaArgEcho(
        EbsaArgStruct *argStruct  /* structure to print */
        );
```

5.1.5 xMinEcho

Echoes data about the very best so far minimum to stderr.

```
void                               /* return type for xMinEcho */
xMinEcho(
        EbsaDataStruct *dataStruct /* methods to use */
        );
```

5.1.6 xMinPrint

Prints data about the very best so far minimum to a user file.

```
void                              /* return type for xMinPrint */
xMinPrint(
        char *filename,           /* ascii stream for output */
        EbsaDataStruct *dataStruct  /* methods to use */
        );
```

5.1.7 TschedPrint

TschedPrint prints the actual temperature time-series "vector" (through the current time step) to a user file. It is anticipated that this routine will be used to create pre-conditioned cooling schedules for problems with similar potential surfaces.

```
void                                    /* return type for TschedPrint */
TschedPrint(
        char *filename,                 /* ascii stream for output */
        EbsaDataStruct *dataStruct      /* methods to use */
        );
```

5.2 Data input

There is currently only one kind of file input, based on the C routine fscanf. Both of these routines (see example2, section 3.4.1).

5.2.1 EbsaArgScan

```
EbsaArgStruct *                    /* return type for EbsaArgScan */
EbsaArgScan(
        char *filename             /* ascii stream for input */
        );
```

5.2.2 EbsaDataScan

```
EbsaDataStruct *                   /* return type for EbsaDataScan */
EbsaDataScan(
        char *filename,            /* ascii stream for input */
        EbsaArgStruct *argStruct,  /* parameters to use */
        EbsaFuncStruct *funcStruct /* methods to use */
```

```
        );
```

5.3 Miscellaneous routines

These are special purpose routines which don't seem to fit any other catagory well.

5.3.1 EbsaRand, EbsaRandReset

Use these routines to generate psuedo random numbers, or to reset the seed used by Ebsa. See sections 6.1.5 and 6.3 for more details.

```
double              /* return type for EbsaRand */
EbsaRand(
        void        /* takes no argument */
        );
```

```
void                /* return type for EbsaRandReset */
EbsaRandReset(
        long seed   /* seed argument */
        );
```

6 User-defined modules

The most essential component of the SA process is the user's description of their problem space. This includes user routines for the creation of state space elements, a function for performing state transistions (move class), and the objective (cost) function. The user's problem description is passed to SDSC EBSA in a call to EbsaFuncCtor().

Optionally, the user can supply metrics to EbsaVisCtor for the graphical representation of their state elements. These metrics are not required to use SDSC EBSA's visual performance monitor, but might be of some aid in understanding peculiar move class behavior. They are discussed in section 6.2.

Examples of function prototypes are givin throughout this section, along with a discussion of their relation to specificaitions in either EbsaFuncCtor() or EbsaVisCtor(). Full source code for these examples is provided in section 6.3.

In the case of parallel computing platforms, there is no restriction on the nature of the user supplied functions. For example, any or all of the user's routines may be parallel programs. SDSC EBSA is implemented as SPMD processes.

6.1 The user's problem space

This section is concerned with the preparation of user functions and parameters for EbsaFuncCtor(). This data describes static attributes of the user's problem. The full prototype for EbsaFuncCtor() is given in section 4.1.2. Individual segments of the prototype are given here in sequence, with example implementation from the full library given in section 6.3.

6.1.1 Cost function

We begin with the 1st user defined argument to EbsaFuncCtor(), the cost function. (The very first argument is an EbsaArgStruct, created by either EbsaArgScan() or EbsaArgCtor().) The prototype for the user's cost function is:

```
double (*f)(void *x),     /* Objective function to be minimized.
                             Must return type double */
```

Note that this requests a pointer to a function that returns a double, and takes an arbitrary pointer as an argument. For efficiency, Ebsa() requires that the cost function always return a floating point (*C type* double) value. However, the user is free to design any kind of state variable they desire. In fact, Ebsa() does not care about most details of the user's domain elements. Here is the prototype for a user cost function whose true input type is an array of doubles:

```
double              /* required return type for LjPot */
LjPot(
        double *x     /* domain element to compute cost */
        );
```

In some optimization problems, the change in the cost function (δf) can be computed an order of magnitude faster than the cost function (f) itself. In this case, the user can compute δf in their *move class* subroutine (section 6.1.5) and add this value to a sub-element in the state element structure. In pseudo-code:

y->f = x->f + δf(x, y)

Note that considerable round-off error can develop after many iterations. To compensate, the user should also implement an error control mechanism. One option is to increment a counter in the state element structure each time the *move* routine is called. The cost function (routine) would then check the stored counter and either return the stored f or recompute it based upon a predetermined number of iterations. Other error control methods can be found in standard numerical analysis references.

6.1.2 # neighbors

During the warm-up phase (a pre-annealing stage), Ebsa() needs to obtain an estimate of the cost domain density of states. An estimate to the number of local neighbors per state is necessary for this computation. The prototype for this parameter is:

```
        long neighbors,           /* # of neigbors per state--or at
                                        least the order or maximum
                                        number of neighbors */
```

6.1.3 State constructor, destructor

There are usually many state space representations available to the user (e.g., choices of coordinate systems). This choice can effect the shape of the cost function and the efficiency of the move class. We provide some general guidelines here, with the view that experimentation is worthwhile.

It is our view the user should take a constructionist approach to their state space. That is, the user should design (on paper) means of generating elements of the cost domain. From this exercise it should become clear what constitutes a state—and more importantly, how to generate the neighbors of any given state. This will greatly ease the task of designing and refining a move class.

SDSC EBSA requires that you supply a routine to generate and initialize elements of your cost domain. These initial states might be randomly distributed or identical. Here is the EbsaFuncCtor() prototype for the user's *constructor* (Ctor) function:

```
        void * (*xCtor)( void ),  /* Allocates memory for an element of
                                        domain(f), initializes the element
                                        to a random state, and returns a
                                        pointer to this data.  Should
                                        return NULL on failure. */
```

This function takes no argument and returns a valid pointer to a single member of the user's cost domain. The actual implementation of this in our example library is:

```
double *              /* return type for LjCtor */
LjCtor(
        void          /* return pseudo-random element */
        );
```

The user is also required to supply a routine to de-allocate (*destruct*) all memory associated with a state element. In general, this function appears to EbsaFuncCtor() as:

```
        void (*xDtor)(void *x),         /* De-allocates all memory
                                           associated with x */
```

Our prototype implementation looks like:

```
void                  /* return type for LjDtor */
LjDtor(
        double *x     /* destroy all memory associated with *x */
        );
```

6.1.4 Copy

Ebsa() occasionally needs to shuffle some of the user states around in it's own internal storage areas. For example, when a new global minimum is detected, this state is saved in a storage area called xMin. In order for Ebsa() to perform such operations, the user needs to supply EbsaFuncCtor() with a function of the form:

```
        void (*xCopy)(void *x,void *y), /* Domain copy function:
                                           copies the contents of *x into
                                           previously allocated *y. */
```

Assume that both `*x` and `*y` were previously allocated by `xCtor`. In the example library, our copy function prototype is:

```
void                  /* return type for LjCopy */
LjCopy(
        double *x,    /* element from function domain */
        double *y     /* return duplicate in *y */
        );
```

6.1.5 Move class

The user's move function performs state transitions from a given domain element to a random (pseudo-uniform) neighbor. We note that the design of the *move class* is intricately related to the structure of a state (section 6.1.3). Some researchers have proposed that moves not be restricted to nearest-neighbors but instead be made to pseudo-random elements of the domain[2]. This kind of move class converts the optimization process from simulated annealing to random search with Boltzmann move probability. Although this approach might be recommended for a certain class of problems, we generally do not recommend it.

Note also that no hysterisis is permitted: every move must be reversible within hardware numerical tolerances. The move class must not change the cost function domain.

Here is how EbsaFuncCtor() expects the move function to be passed:

```
        void (*xMove)(void *x,void *y), /* Domain permutation function:
                                           User passes pointer to element
                                           *x in domain of f to
                                           (*xMove)(x,y) and a psuedo-
```

```
                                        random neighbor of *x is
                                        returned in *y.  Note: storage
                                        for *y must be pre-allocated */
```

Our example prototype is very similar:

```
void                /* return type for LjMove */
LjMove(
        double *x,    /* element from function domain */
        double *y     /* random neighbor state */
        );
```

6.1.6 Print, Scan

Ebsa() also requires that the user supply methods of printing (a la `fprintf`) and scanning individual state elements. This permits output functions such as xMinEcho() and EbsaDataPrint() to write out arbitrary domain elements to stderr or disk; and the input function EbsaDataScan() to re-scan files created by EbsaDataPrint().

There are two constraints on the design of these function. The output must be ascii so that it can read (scanned) on any hardware platform. The output must contain a trailing white space character so that there will be some separation between consecutive domain elements and other output data.

The user's output function should be of the form

```
        void (*xPrint)(FILE *fp, void *x),
                                      /* Prints user *x and a trailing
                                         white-space character to an
                                         ascii stream (e.g. \\t, \\n). */
```

which our example prototype follows:

```
void                /* return type for LjPrint */
LjPrint(
        FILE *fp,
        double *x     /* domain element to print */
        );
```

The user's input function is required to fit:

```
        void (*xScan)(FILE *fp, void *x),
                                        /* When positioned at beginning of
                                           string generated by xPrint,
                                           xScan scans in a user *x value
                                           (incl. trailing white space). */
```

Here is the prototype for our scan function:

```
void                /* return type for LjScan */
LjScan(
        FILE *fp,
        double *x     /* scan domain element */
        );
```

6.2 Visualization metrics and graphics

The user may optionally supply two functions to supplement the visual diagonostic system. The first is a simple norm which EbsaVis() will use to create a contour plot of the current ensemble. The second is a computer-graphics routine to draw a 2D or 3D cartoon of individual domain elements. Both of these routines require a scalar to indicate the dimension of plot being used.

6.2.1 Norm for contour space plot

To enable the contour plot, the user needs to supply

- a function which returns a scalar or normative 2D point representation of cost domain elements
- number of dimensions (1 or 2) in norm *range*
- a min and max bound for each dimension

If any of these items are `NULL`, this plot type will be disabled.

The supplied bounds are regretably fixed. The trade off would be to for EbsaVis() to search all walkers for the min and max each iteration, or for the user to supply a min/max function. Since the user is designing the norm function, they will typically know these values apriori.

Here are the function declarations required by EbsaVisCtor:

```
        double * (*xNorm)(void *x),  /* Converts user x into double
                                        vector of length xNormDim
                                        for plot of E(x).  The user
                                        should supply a valid function
                                        pointer or NULL */


        long xNormDim,               /* Dimension of xNorm.  Can be 1, 2,
                                        or NULL to disable */


        double *xNormBounds,         /* xNormBounds[] is NULL or contains
                                        pairs of type double describing
                                        the min and max value for each
                                        dimension of xNorm */
```

Our example library contains the following prototype for its 1D norm function:

```
double *               /* return type for LjNorm1 */
LjNorm1(
        double *x      /* compute Norm of this element */
        );
```

We have used a `#define` macro and static array to supply the norm dimension and bounds:

```
#define LJ_NORM1DIM ( 1 )        /* dimension of normative value
                                    returned by LjNorm1 */


static double LjNorm1Bounds[2] = { LJ_EST_NORM1_MIN, LJ_EST_NORM1_MAX };
                                /* the min, max for LjNorm1 */
```

6.2.2 Graphics routine for rendering domain element

Some researchers may also find it of use to supply a rendering routine which draws a cartoon of an input domain element. EbsaVis() will supply the window, etc. for this graphic—the user need only provide the actual drawing. The graphic can be either 2D or 3D, specified in the xGeomDim parameter to EbsaVisCtor():

```
        void (*xGeom)(void *x)          /* Draws a model of in unit
                                          coordinates anchored on the
                                          origin. */
```

In order for EbsaVis() to manipulate the graphic correctly, all drawing must occur in a unit cube (or 2D square) and be centered on the origin. EbsaVis() maintains internal scaling and rotation factors to allow interactive zooming and rotation of the object. EbsaVisCtor() expects the rendering function to meet the following prototype:

```
        long xGeomDim,                  /* Dimension of xGeom.  Can be 2, 3,
                                           or NULL to disable */
```

In our example library, the dimension parameter and rendering function are defined as:

```
#define LJ_GEOMDIM ( 3 )        /* dimension of Geometry
                                   drawn by LjGeom */


void                /* return type for LjGeom */
LjGeom(
        double *x       /* element to draw */
        );
```

6.3 Example library

The following source code resides in `$EBSA_HOME/example/lj`. We include it here for reference. The file `include/lj.h` contains all of the prototypes given above plus a few other definitions; `src/lj.c` contains the actual implementations.

This library implements a simplistic implementation of a Leonard-Jones potential problem for 19 point particles. The cost function is simply the sum of all inter-particle potentials. The cost domain is all configurations of 19 point particles that fit within a finite volume of space normalized about the origin. The representation of each configuration is given by a vector of $3 * 19$ (3 dimensions $*$ 19 particles) floating point (type double) values. The move class displaces a random particle of each 19-point configuration a small, discrete distance (defined by `LJ_MOVE`). For visualization, a scalar norm was chosen that represents the sum of particle distances from the origin. The visualization geometry function draws a 3D perspective view of a given configuration.

6.3.1 lj.h

```
/**
 **  FILE
 **        lj.h                generic version, works across all platforms
 **
 **  PROJECT
 **        SDSC EBSA
 **
 **  DESCRIPTION
 **        Include file for lj.c
 **
 **  PUBLIC CONTENTS
 **                            d =defined constant
 **                            f =function
 **                            m =defined macro
 **                            t =typedef/struct/union
 **                            v =variable
 **                            ? =other
 **
 **                            m various #define'd constants
 **
 **  PRIVATE CONTENTS
 **        none
 **
 **  HISTORY
 **        $Log$
 **/

#ifndef LJ_H
#define LJ_H

#include "Ebsa.h"

#include <stdio.h>

#ifdef LINKLIB_GL
#include <gl/gl.h>
#endif

#include <math.h>
```

```
#define LJ_PARTICLES (19)       /* Mark Fletcher configuration */
#define LJ_ELEMENTS 4           /* coodinate dimensions */
#define LJ_MAX_PENALTY ( LJ_PARTICLES * 10.0 )
#define LJ_MIN_POS (-LJ_PARTICLES / 4.0)
#define LJ_MAX_POS (LJ_PARTICLES / 4.0)
#define LJ_DOMAIN ( LJ_MAX_POS - LJ_MIN_POS )
#define LJ_EST_NORM1_MIN ( 0.6667 * LJ_MAX_POS )
#define LJ_EST_NORM1_MAX ( 1.2248 * LJ_MAX_POS )
#define LJ_EXTENT ( 1.732 * LJ_MAX_POS )
#define LJ_SCALE ( 1.0 / LJ_MAX_POS )


#define LJ_NORM1DIM ( 1 )        /* dimension of normative value
                                   returned by LjNorm1 */

static double LjNorm1Bounds[2] = { LJ_EST_NORM1_MIN, LJ_EST_NORM1_MAX };
                                /* the min, max for LjNorm1 */

#define LJ_NORM2DIM ( 2 )        /* dimension of normative value
                                   returned by LjNorm2 */

static double LjNorm2Bounds[4] = { 0, LJ_EXTENT/2, LJ_EXTENT/2, LJ_EXTENT };
                                /* the min, max for LjNorm2 */

#define LJ_GEOMDIM ( 3 )        /* dimension of Geometry
                                   drawn by LjGeom */

/* prototypes */

double                  /* required return type for LjPot */
LjPot(
        double *x       /* domain element to compute cost */
        );


void                    /* return type for LjMove */
LjMove(
        double *x,      /* element from function domain */
        double *y       /* random neighbor state */
        );
```

```
void                  /* return type for LjPrint */
LjPrint(
        FILE *fp,
        double *x     /* domain element to print */
        );

void                  /* return type for LjScan */
LjScan(
        FILE *fp,
        double *x     /* scan domain element */
        );

double *              /* return type for LjCtor */
LjCtor(
        void          /* return pseudo-random element */
        );


void                  /* return type for LjDtor */
LjDtor(
        double *x     /* destroy all memory associated with *x */
        );

void                  /* return type for LjCopy */
LjCopy(
        double *x,    /* element from function domain */
        double *y     /* return duplicate in *y */
        );

void                  /* return type for LjCoG */
LjCoG(
        double *x     /* Given a stream of coordinates created
                        by LjCtor, shift their values so the center
                        of gravity is the origin */
        );

double *              /* return type for LjNorm1 */
LjNorm1(
        double *x     /* compute Norm of this element */
        );
```

```
double *                /* return type for LjNorm2 */
LjNorm2(
        double *x       /* compute Norm of this element */
        );

void                    /* return type for LjGeom */
LjGeom(
        double *x       /* element to draw */
        );

#endif
```

6.3.2 lj.c

```
/**
 ** FILE
 **      lj.c                generic version
 **
 ** PROJECT
 **      SDSC EBSA
 **
 ** DESCRIPTION
 **      Sample program demonstrating SDSC EBSA library
 **
 ** PUBLIC CONTENTS
 **                         d =defined constant
 **                         f =function
 **                         m =defined macro
 **                         t =typedef/struct/union
 **                         v =variable
 **                         ? =other
 **
 **                         f LjPot
 **                         f LjMove
 **                         f LjCtor
 **                         f LjDtor
 **                         f LjScan
 **                         f LjPrint
 **                         f LjCopy
 **                         f LjNorm1
```

```
 **                        f LjNorm2
 **                        f LjCoG
 **
 **  PRIVATE CONTENTS
 **       none
 **
 **  HISTORY
 **       $Log$
 **/

#include <stdio.h>
#include "lj.h"
/*
 *  FUNCTION
 *       LjPot
 *
 *  DESCRIPTION
 *       LJ Potential
 */

double                  /* required return type for LjPot */
LjPot(
        double *x       /* domain element to compute cost */
        )
{
        extern double sqrt( double );
        extern double floor( double );

        double rSum;
        double kSum[LJ_PARTICLES];
        double dx[LJ_PARTICLES], dy[LJ_PARTICLES], dz[LJ_PARTICLES];
        double dx2[LJ_PARTICLES], dy2[LJ_PARTICLES], dz2[LJ_PARTICLES];
        double r2[LJ_PARTICLES], r6i[LJ_PARTICLES], r12ih[LJ_PARTICLES];
        double ux[LJ_PARTICLES], vx[LJ_PARTICLES];
        double uy[LJ_PARTICLES], vy[LJ_PARTICLES];
        double uz[LJ_PARTICLES], vz[LJ_PARTICLES];

        long i, k, p;


        rSum = 0.0;
        for ( k=0; k<LJ_PARTICLES; k++ )
```

```
        {
                p = k * LJ_ELEMENTS;
                ux[k] = x[p];
                uy[k] = x[p+1];
                uz[k] = x[p+2];

#pragma _CRI ivdep
                for ( i=0; i<k; i++ )
                {
                        vx[i] = x[(i*LJ_ELEMENTS)];
                        vy[i] = x[(i*LJ_ELEMENTS)+1];
                        vz[i] = x[(i*LJ_ELEMENTS)+2];
                        dx[i] = ux[k] - vx[i];
                        dy[i] = uy[k] - vy[i];
                        dz[i] = uz[k] - vz[i];
                        dx2[i] = dx[i] * dx[i];
                        dy2[i] = dy[i] * dy[i];
                        dz2[i] = dz[i] * dz[i];
                        r2[i] = dx2[i] + dy2[i] + dz2[i];
                        r6i[i] = 1.0 / (r2[i] * r2[i] * r2[i] );
                        r12ih[i] = ( r6i[i] * r6i[i] ) * 0.5;
                }

                r6i[k] = 0;
                r12ih[k] = 0;

#pragma _CRI ivdep
                for ( i=k+1; i<LJ_PARTICLES; i++ )
                {
                        vx[i] = x[(i*LJ_ELEMENTS)];
                        vy[i] = x[(i*LJ_ELEMENTS)+1];
                        vz[i] = x[(i*LJ_ELEMENTS)+2];
                        dx[i] = ux[k] - vx[i];
                        dy[i] = uy[k] - vy[i];
                        dz[i] = uz[k] - vz[i];
                        dx2[i] = dx[i] * dx[i];
                        dy2[i] = dy[i] * dy[i];
                        dz2[i] = dz[i] * dz[i];
                        r2[i] = dx2[i] + dy2[i] + dz2[i];
                        r6i[i] = 1.0 / (r2[i] * r2[i] * r2[i] );
                        r12ih[i] = ( r6i[i] * r6i[i] ) * 0.5;
                }
```

```
                kSum[k] = 0;
                for ( i=0; i<LJ_PARTICLES; i++ )
                        kSum[k] += ( r12ih[i] - r6i[i] );

        }

        rSum = 0;
        for ( k=0; k<LJ_PARTICLES; k++ )
                rSum += kSum[k];
        if ( rSum < LJ_MAX_PENALTY )
                return rSum;
        else
                return LJ_MAX_PENALTY;
}


/*
 *  FUNCTION
 *        LjCopy
 *
 *  DESCRIPTION
 *        Given an instance of a state variable, copy to a
 *        previously allocated pointer.
 *
 *        Note: the pointer is allocated for argSize bytes.
 *        If the variable contains internal pointers, they
 *        should be allocated here.
 */

void                    /* return type for LjCopy */
LjCopy(
        double *x,      /* element from function domain */
        double *y       /* return duplicate in *y */
        )
{
        long i;

        for ( i=0; i<LJ_ELEMENTS*LJ_PARTICLES; i++ )
                y[i] = x[i];
}
```

```
/*
 *  FUNCTION
 *        LjMove
 *
 *  DESCRIPTION
 *        Given an instance of a state variable, generate a random
 *        neighbor move.
 */

void                    /* return type for LjMove */
LjMove(
        double *x,      /* element from function domain */
        double *y       /* return neighbor state */
        )
{
        static double LJ_MOVE = ( 0.57735 * 0.05 );
                              /* 1/sqrt(3) * 5% */
        double dx, dy, dz;    /* incremental changes */
        long k, k1, k2;       /* indices */

        /* copy x into y */
        LjCopy( x, y );

        /* choose a random element of y */
        k = (long) floor( LJ_PARTICLES * EbsaRand() );
        k *= LJ_ELEMENTS;
        k1 = k + 1;
        k2 = k + 2;

        /* select random direction */
        dx = 1 - (2 * floor(EbsaRand() + 0.5));
        dy = 1 - (2 * floor(EbsaRand() + 0.5));
        dz = 1 - (2 * floor(EbsaRand() + 0.5));


        /* move it */
        y[k] = x[k] + (dx * LJ_MOVE);
        y[k1] = x[k1] + (dy * LJ_MOVE);
        y[k2] = x[k2] + (dz * LJ_MOVE);

        /* Update distance for this coordinate */
```

```
        y[k+3] = sqrt((y[k]*y[k]) +  (y[k1]*y[k1]) + (y[k2]*y[k2]));

        /* normalize center of gravity */
        LjCoG (y);
}


/*
 *  FUNCTION
 *        LjCtor
 *
 *  DESCRIPTION
 *        Generate a pseudo-random element of the Lj domain.
 */

double *        /* return type for LjCtor */
LjCtor(
        void    /* return pseudo-random element */
        )
{
        double * x;    /* double stream of pseudo random elements in Lj */
        long i;        /* temp. counter */
        long j;        /* temp. counter */
        long k;        /* temp. counter */
        long d;        /* temp. counter */
        double cost;        /* cost of generated state */
        double minmax;        /* see code */
        long w;        /* temp. counter */

        x = (double *)
                malloc (sizeof (double) * LJ_PARTICLES * LJ_ELEMENTS);
        if (x == (double *) NULL)
        {
                fprintf ( stderr,"LjCtor: malloc for x failed\n" );
                return x;
        }

        i = 0;
        while ( i < (LJ_ELEMENTS * LJ_PARTICLES) )
        {
                j = i + 1;
                k = i + 2;
```

```
                d = i + 3;
                x[i] = LJ_MIN_POS + ( EbsaRand() * LJ_DOMAIN );
                x[j] = LJ_MIN_POS + ( EbsaRand() * LJ_DOMAIN );
                x[k] = LJ_MIN_POS + ( EbsaRand() * LJ_DOMAIN );
                x[d] = sqrt((x[i] * x[i]) + (x[j] * x[j]) + (x[k] * x[k]));
        i += 4;
        }

        LjCoG (x);

        return x;
}


/*
 *  FUNCTION
 *        LjDtor
 *
 *  DESCRIPTION
 *        Deallocate an element of the domain.
 */

void                    /* return type for LjDtor */
LjDtor(
        double *x       /* destroy all memory associated with *x */
        )
{
        free (x);
}


/*
 *  FUNCTION
 *        LjPrint
 *
 *  DESCRIPTION
 *        Prints an element of x and places a trailing tab afterwards.
 *        The trailing space is helpful when scanning.
 */

void                   /* return type for LjPrint */
LjPrint(
```

```
        FILE *fp,
        double *x     /* domain element to print */
        )
{
        long i;        /* temp counter */


        for (i=0; i < LJ_PARTICLES; i++)
        {
           fprintf (fp, "%g\t%g\t%g\t%g\n",
                x[(LJ_ELEMENTS*i)], x[(LJ_ELEMENTS*i)+1],
                x[(LJ_ELEMENTS*i)+2], x[(LJ_ELEMENTS*i)+3]
                );
        }
}


/*
 *  FUNCTION
 *        LjScan
 *
 *  DESCRIPTION
 *        Scans one element from the stream pointed to by fp. It will get
 *        that one element plus its trailing white space.
 */


void                  /* return type for LjScan */
LjScan(
        FILE *fp,
        double *x     /* scan domain element */
        )
{
        long i;        /* temp counter */

        for (i=0; i < LJ_PARTICLES; i++)
                fscanf( fp, "%lg%lg%lg%lg",
                        &(x[(LJ_ELEMENTS*i)]), &(x[(LJ_ELEMENTS*i)+1]),
                        &(x[(LJ_ELEMENTS*i)+2]), &(x[(LJ_ELEMENTS*i)+3])
                        );
}
```

```
/*
 *  FUNCTION
 *        LjNorm1
 *
 *  DESCRIPTION
 *        Given an instance of a state variable, generate a scalar
 *        value for a state space plot.  In this case, it is simply
 *        the average particle distance to the origin.
 */

double *                 /* return type for LjNorm1 */
LjNorm1(
        double *x        /* compute Norm of this element */
        )
{
        long i;

        static double rSum;

        rSum = 0;

        for (i=3; i<( LJ_ELEMENTS * LJ_PARTICLES ); i+=LJ_ELEMENTS )
        {
                rSum += x[i];
        }

        rSum /= LJ_PARTICLES;

        return ( &rSum );
}


/*
 *  FUNCTION
 *        LjNorm2
 *
 *  DESCRIPTION
 *        Given an instance of a state variable, generate a 2D norm
 *        value for a state space plot.  Here we calculate the min
 *        and max particle distance from the origin.
 */
```

```
double *                /* return type for LjNorm2 */
LjNorm2(
        double *x       /* compute Norm of this element */
        )
{
        long i;

        static double vNorm[2];

        vNorm[1] = vNorm[0] = x[3];

        for ( i = ( 3 + LJ_ELEMENTS );
                i<( LJ_ELEMENTS * LJ_PARTICLES ); i+=LJ_ELEMENTS )
        {
                if ( vNorm[0] > x[i] )
                        vNorm[0] = x[i];
                if ( vNorm[1] < x[i] )
                        vNorm[1] = x[i];
        }

        return ( vNorm );
}


/*
 *  FUNCTION
 *        LjGeom
 *
 *  DESCRIPTION
 *        Given an instance of a state variable, draw a cartoon
 *        of the state using SGI GL.
 */

void                    /* return type for LjGeom */
LjGeom(
        double *x       /* element to draw */
        )
{

#ifndef LINKLIB_GL
        return;
```

```
#else

        long i, k;              /* iterates */

        double dMin, dMax;      /* min, max distance from origin */
        float gScale;           /* graphics scale */
        float v3[3];            /* 3d coordinate value */


        dMin = dMax = x[3];
        for (i=(3 + LJ_ELEMENTS);
                i<( LJ_ELEMENTS * LJ_PARTICLES ); i+=LJ_ELEMENTS )
        {
                if ( dMin > x[i] )
                        dMin = x[i];
                if ( dMax < x[i] )
                        dMax = x[i];
        }

        pushmatrix();

        gScale = (float) 0.7 / dMax;
        scale( gScale, gScale, gScale );

        k = 0;
        for (i=0; i<( LJ_ELEMENTS * LJ_PARTICLES ); i+=LJ_ELEMENTS )
        {
                cpack( EbsaHue(
                        (double) k,
                        (double) 0,
                        (double) LJ_PARTICLES,
                        30,
                        330 )
                        );

                bgnline();
                        v3f( Ebsa3dFltPt(
                                0.0, 0.0, 0.0, v3 )
                                );
                        v3f( Ebsa3dFltPt(
                                x[i], x[i+1], x[i+2], v3 )
                                );
```

```
                endline();
                k++;
        }

        popmatrix();

#endif
}


/*
 *  FUNCTION
 *      LjCoG
 *
 *  DESCRIPTION
 *      Given a pointer to a stream of doubles which gives the three
 *      coordinates to a particle and their distance from the origin,
 *      this will return the coordinates of the center of gravity, and
 *      an average distance.
 *      WEAKNESS: Assumes ALL paritcles have same mass.
 */

void                /* return type for LjCoG */
LjCoG (
        double *x   /* Given a stream of coordinates created
                       by LjCtor, shift their values so the center
                       of gravity is the origin */
        )
{
        long i;         /* temporary counter */
        long j;         /* temporary counter */
        long k;         /* temporary counter */
        long d;         /* temporary counter */
        long p;         /* temporary counter */

        double CoG[LJ_ELEMENTS];        /* center of gravity */

        for (p=0; p<LJ_ELEMENTS-1; p++)
        {
                CoG[p] = 0;

                for ( i=p; i<(LJ_ELEMENTS * LJ_PARTICLES); i+=LJ_ELEMENTS )
```

```
		{
			CoG[p] += x[i];
		}

		CoG[p] /= LJ_PARTICLES;

		for ( i=p; i<(LJ_ELEMENTS * LJ_PARTICLES); i+=LJ_ELEMENTS )
		{
			x[i] -= CoG[p];
		}
	}

	i = 0;
	while ( i < (LJ_ELEMENTS * LJ_PARTICLES) )
	{
		j = i + 1;
		k = i + 2;
		d = i + 3;
		x[d] = sqrt((x[i] * x[i]) + (x[j] * x[j]) + (x[k] * x[k]));

	i += 4;
	}
}
```

7 Development Tools

To ease the development cycle, two software implementation tools are supplied with Ebsa. The first is for generation of example programs and (re)make files, while the second is for the (nearly) transparent porting of your code to other platforms.

7.1 ebsagen

Ebsagen is a Bourne shell script which generates source code and directory structures for Ebsa applications. The main intent of the script is to provicde a significant head start to those users who must develop their applications from scratch. Users who will add SDSC EBSA library calls to existing code might also derive some benefit from the script.

Both ebsagen and remake have a "project" mentality. They assume that software packages—whether they be executables or libraries—are organized in a directory that bears the project name. They also assume the project directory is where the (R)Makefile resides, along with template files that have names of the form *project.template-name*.

Ebsagen looks for template files of the form *project*`.ebsa`. For example, the sample program given in section 3.4.1 was generated from the template file `example.ebsa`. These are simple ascii files that contain information about tokens in the user modules. If no template file is found, ebsagen automatically generates an example.

The command line syntax for ebsagen is:

`% ebsagen [ -f` *project* `| -help ]`

If no argument or `-help` is given, a help file is sent to stderr. Otherwise, ebsagen will proceed as follows:

1. Check if

   (a) the current directory or one of its subdirectories is named *project*;

   (b) a template file named *project*`.ebsa` exists.

   If neither exists, ebsagen tries to create a subdirectory named *project*. The user is asked to confirm this action. On failure, the program exits. If successful, all further action is focused in this directory.

   If no template file exists, an example is written and used for input.

2. The template file is read.

3. If no `src/` sub-directory is found, it is created.

4. If the file `src/`*project*`.c` exists, ebsagen tries to move it to `src/`*project*`.c.ymdhs` — where "ymdhs" is a unix date stamp. The user is asked to confirm this action. On failure, the program exits.

5. `src/`*project*`.c` is created.

6. A *project*`.rmake` file is created in the same fashion.

7. remakegen is called to create `remake`, `RMakefile`, and `makefile` scripts.

8. If the `RMakefile` script is new, it is modified to full functionality for compiling `src/`*project*`.c`.

If all of these steps are successfully completed, the user can enter

```
% remake
```

(instead of `make`) to create an executable. We prefer the former for multi-platform development.

7.1.1 ebsagen template format

An ebsagen template is simply a listing of user parameters and functions to be called by SDSC EBSA's constructor functions (section 6). User tokens are identified by assignment statements of the form `Ebsa_`*token*`=`*USER_TOKEN*. Blank lines, lines beginning with `#`, and all white space characters except `\n` are ignored. In the example automatically produced by ebsagen, default values appear in upper-case.

Here is the file used to generate all source code in the `example2` directory:

```
# ebsagen default ebsa template file
#
Ebsa_project            =       example2
Ebsa_project_home       =       ${EBSA_HOME}/example/example2
#
Ebsa_user_lib           =       lj
Ebsa_user_lib_home      =       ${EBSA_HOME}/example/lj
#
Ebsa_f                  =       LjPot
Ebsa_neighbors          =       LJ_PARTICLES
```

```
Ebsa_xCtor            =       LjCtor
Ebsa_xDtor            =       LjDtor
Ebsa_xCopy            =       LjCopy
Ebsa_xMove            =       LjMove
Ebsa_xPrint           =       LjPrint
Ebsa_xScan            =       LjScan
Ebsa_x0               =       NULL
Ebsa_E0               =       NULL
#
Ebsa_xNormDim         =       LJ_NORM1DIM
Ebsa_xNormBounds      =       LjNorm1Bounds
Ebsa_xNorm            =       LjNorm1
Ebsa_xGeomDim         =       LJ_GEOMDIM
Ebsa_xGeom            =       LjGeom
```

7.2 remakegen and remake

Ebsagen calls remakegen to generate `remake` and `RMakefile` scripts. The user is free to develop with either the sample `RMakefile` or `makefile`. The former is recommended for portability. `Remakegen` is available as a stand-alone package from `ftp.sdsc.edu` in directory `pub/sdsc/system/unix/remake`.

SDSC *remake* and *remakegen* are Bourne shell scripts used in conjunction with *make* for the development of binary applications that have multiple hardware targets. Rather than constructing a `Makefile` for each target architecture, *remake* sets shell variables which are passed to a single, generic Unix *make* script called `RMakefile`. The `RMakefile` variables include hardware specific information required for the compilation, linking, loading, and reliable storage of package binaries. This is especially useful during the development phase, when `Makefile`(s) must be updated on a regular basis.

The *remakegen* script is for the automatic generation of *remake* scripts and `RMakefile`(s) from several (a handfull) of user-defined parameters. These parameters are placed in an ascii template file for reuse. *Remakegen* will generate an example template file when none exists.

For more information about remakegen and remake, see `remakegen.ps`.

7.3 ebsa-qgen

The utility `ebsa-qgen` is located in `$EBSA_HOME/bin`. This Bourne shell script is for generating batch files files suitable for NQS submissions on SDSC's Intel Paragon. Small modifications (in the queue names) might be required for use at other sites. The general syntax is:

```
        % ebsa-qgen PROG Q QNODES QLEN MSTEPS [ MNODES ]

where
        PROG      =        name of Ebsa executable
        Q         =        either q or qf, for regular or fat nodes
        QNODES    =        number of nodes in desired queue
        QLEN      =        s, m, or l for small, medium, or large jobs
        MSTEPS    =        max number of annealing steps
        MNODES    =        actual number of nodes to use, MNODES <= QNODES.
```

The batch file name is echoed to stdout.

References

[1] B. Andresen, K. H. Hoffmann, K. Mosegaard, J. Nulton, J. M. Pedersen, and P. Salamon. On lumped models for thermodynamic properties of simulated annealing problems. *J. Phys. France*, 49:1485, 1988.

[2] C. Belisle. Convergence theorems for a class of simulated annealing algorithms on $\Re^d$. In *SIAM Conference on Simulation and Monte Carlo Methods*. SIAM, 1993.

[3] R.S. Berry. Potential surfaces and dynamics: what clusters tell us. *Chemical Reviews*, in press, 1994.

[4] K. Binder. *Monte Carlo Methods in Statistical Physics*. Springer Verlag, 1979.

[5] R. Frost, N. Priebe, and P. Salamon. An ensemble-based simulated annealing library for multiprocessors. In *ISUG-93 Proc.* Intel SSD, 1993.

[6] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *PAMI*, 6:721–741, 1984.

[7] B. Hajek. Cooling schedules for optimal annealing. *Math. Oper. R.*, 13:311–329, 1988.

[8] J. R. Harland and P. Salamon. Simulated annealing: A review of the thermodynamic approach. *N. Phys. B (Proc. Suppl.)*, 5A:109–115, 1988.

[9] K. H. Hoffmann and P. Salamon. The optimal simulated annealing schedule for a simple model. *Journal of Physics A*, 23:3511, 1990.

[10] K. H. Hoffmann and P. Salamon. Simulated annealing for single minimum optimization problems. *International Journal of Computer Mathematics*, 39:193–204, 1991.

[11] K. H. Hoffmann, P. Sibani, J. M. Pedersen, and P. Salamon. Optimal ensemble size for parallel implementations of simulated annealing. *Appl. Math. Lett.*, 3:53–56, 1990.

[12] K. H. Hoffmann, D. Wuertz, C. de Groot, and M. Hanf. Concepts in optimizing simulated annealing schedules: An adaptive approach for parallel and vector machines. In M. Grauer and D. B. Pressmar, editors, *Parallel and Distributed Optimization*. Springer-Verlag, Heidelberg, 1991.

[13] M. D. Huang, F. Romeo, and A. Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. In *ICCAD 1986*, page 381. IEEE, 1986.

[14] E.T. Jaynes. Information theory and statistical mechanics. *Phys. Rev.*, 106:620, 1957.

[15] E.T. Jaynes. Information theory and statistical mechanics II. *Phys. Rev.*, 108:171, 1957.

[16] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671, 1983.

[17] Z. Koren, K. Mosegaard, E. Landa, P. Thore, and A. Tarantola. Seismic background velocity estimation and model error analysis by simulated annealing. *Journal of Geophysical Research*, 96(B12):20289–20299, 1991.

[18] J. Lam and J. M. Delosme. An adaptive annealing schedule. Technical Report 8608, Department of Electrical Engineering, Yale University, 1989.

[19] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1807, 1953.

[20] I. Morgenstern and D. Wuertz. Simulated annealing for spin-glass-like optimization problems. *Z. Phys. B.*, 67:1351–1356, 1987.

[21] H. Muehlenbein and D. Schleirkamp-Voosen. Predictive models for the breeder genetic algorithm: I. continuous function optimization. *Evolutionary Computation*, 1:25–49, 1993.

[22] J. D. Nulton and P. Salamon. Statistical mechanics of combinatorial optimization. *Phys. Rev. A*, 37:1351–1356, 1988.

[23] G. Ruppeiner. Implementation of an adaptive, constant thermodynamic speed simulated annealing schedule. *N. Phys. B*, 5A:116–121, 1988.

[24] G. Ruppeiner, J. M. Pedersen, and P. Salamon. Ensemble approach to simulated annealing. *J. Phys. I*, 1:455–470, 1991.

[25] P. Salamon, K. H. Hoffmann, J. Ro Harland, and J. D. Nulton. An information theoretic bound on the performance of simulated annealing algorithms. Technical Report IRC.89.2, Interdisciplinary Research Center, San Diego State University, 1989.

[26] P. Salamon, J. D. Nulton, J. R. Harland, J. Pedersen, G. Ruppeiner, and L. Liao. Simulated annealing with constant thermodynamic speed. *Comp. Phys. Comm.*, 49:423–428, 1988.

[27] P. Sibani and K. H. Hoffmann. Random walks on Cayley trees. *Europhysics Letters*, 4:967–972, 1987.

[28] P. Sibani and K. H. Hoffmann. Relaxation in complex systems: Local minima and their exponents. *Europhysics Letters*, 16:423–428, 1991.

[29] P. Sibani, J. M. Pedersen, K. H. Hoffmann, and P. Salamon. Monte Carlo dynamics of optimization problems, a scaling description. *Phys. Rev. A*, 42:7080–7086, 1990.

[30] P. Sibani, J. C. Schön, P. Salamon, and J. O. Andersson. Emergent hierarchical structures in complex system dynamics. *Europhysics Letters*, 22:479–485, 1993.

[31] P.F. Stadler and W. Schnabl. The Landscape of the TSP. *Phys. Lett. A*, 161:337, 1992.

[32] P.J.M. van Laarhoven and E.H.L. Aarts. *Simulated annealing: theory and applications*. Kluwer Academic, 1987.

[33] P. D. Vestergaard and K. Mosegaard. Inversion of post-stack seismic data using simulated annealing. *Geophysical Prospecting*, 39:613–624, 1991.

[34] S. R. White. Concepts of scale in simulated annealing. In *ICCAD '84 Proc.*, pages 646–651. IEEE, 1984.

[35] J. Yi, J. Bernholc, and P. Salamon. Simulated annealing strategies for molecular dynamics. *Comp. Phys. Comm.*, 66:177–180, 1991.

[36] T. Zimmermann and P. Salamon. The demon algorithm. *Intern. J. Computer Math.*, 42:21–31, 1992.