Work Information

Logan Brown

February 5, 2015

Contents

1 PSUADE

1.1 Installing PSUADE

1.1.1 General Steps

1. tar xzvf PSUADE.tar.gz
2. cd PSUADE_1.7.2
3. mkdir build; cd build
4. cmake .. &> cmake.output &
   (This step will take a minute or so.)
5. make &> make.output &
   (This step will take some time.)

If the make is successful, build/bin should contain the psuade executable. Set $PATH and $LD_LIBRARY_PATH, based on your installation location. To test your build, change to your build directory and type "make test".

1.1.2 Installing on Darter

1. *tar xzvf PSUADE.tar.gz*
2. *cd PSUADE_1.7.2*
3. *mkdir build; cd build*
4. *cmake .. &> cmake.output &*
5. **Go to cmake_install.cmake and change CMAKE_INSTALL_PREFIX from "/usr/local/" to a directory where you have permission.**
   I used "PSUADE_v1.7.2/inst/"
6. *make &> make.output &*

1.1.3 Installing on star1

Note: Requires cmake, you may have to install a local copy.

1. *tar xzvf PSUADE.tar.gz*
2. *cd PSUADE_1.7.2*
3. *mkdir build; cd build*

4. **export LD_LIBRARY_PATH=**

   /home/kwong/LAPACK/lib-shared/:$LD_LIBRARY_PATH

   Alternatively, use another libblas.so, if you have one.

5. *cmake .. &> cmake.output &*

6. **Go to cmake_install.cmake and change CMAKE_INSTALL_PREFIX from "/usr/local/" to a directory where you have permission.**

7. **Go to CMakeCache.txt and change LAPACK_lapack_LIBRARY:FILEPATH to**

   /home/kwong/LAPACK/lib-shared/liblapack.so

   OR

   /home/lbrown/iel-2.0/EXTLIB/LAPACK/lapack-3.4.0-shared/liblapack.so

   or to whatever liblapack.so you prefer

8. *make &> make.output &*

1.2 Compiling PSUADE as a Module

1.2.1 main in Psuade.cpp as a module

I recommend first copying over Psuade.cpp to PsuadeIEL.cpp, and making all of the relevant changes in that file. That way, you can still compile Psuade as a standalone software piece for data analysis and debugging, in addition to compliling the Psuade module library.

If you choose to just modify Psuade.cpp, replace all instances of 'PsuadeIEL.cpp' below with 'Psuade.cpp'. The resulting library will be incapable of running PSUADE without the DIEL, because it will lack a main function.

1. Add include files to PsuadeIEL.cpp

```
#include "../../../../iel-2.0/EXECUTIVE/HYBRID/IEL\_comm/IEL.h"
#include "../../../../iel-2.0/EXECUTIVE/HYBRID/executive/IEL\_exec\_info.h"
#include "../../../../iel-2.0/EXECUTIVE/HYBRID/IEL\_comm/tuple\_comm.h"
#include "../../../../iel-2.0/EXECUTIVE/HYBRID/IEL\_comm/arrayList.h"
```

2. Change main(int argc, char** argv) to PSUADEmain(IEL_exec_info_t *exec_info)

3. Add the following lines to CMakeLists.txt

```
SET(CMAKE_CXX_FLAGS "-DMPICH_SKIP_MPICXX")

include_directories("../../../EXECUTIVE/HYBRID/executive/")
```

```
include_directories("../../../EXECUTIVE/HYBRID/IEL_comm/")
include_directories("../../MPICH/mpich-shared/include/")
add_library (PsuadeModule ${LIBRARY_TYPE} "Src/Main/PsuadeIEL.cpp" ${psuade_SRC}
    ${psuade_HDRS} ${PDF_SRC})
```

(The add_library command should be one line, but text wrapping makes it look like two)

1.2.2 Passing command line arguments to PSUADE

Command line arguments should be passed through the argument by the cfg file.

I added these lines to the PsuadeIEL.cpp file to get the arguments. PSUADE assumes that the first argument in argv is "/path/to/psuade/psuade", so we put a filler argument in argv[0].

```
int argc = exec_info->modules[exec_info->module_num].mod_argc + 1;

char* argv[argc];
char* temporary = "PsuadeModule";
argv[0] = temporary;
int i=0;
for(i=0; i<(argc-1); i++)
    {    argv[i+1] = exec_info->modules[exec_info->module_num].mod_argv[i];    }
```

PSUADE has four major functions - Sensitivity Analysis, Uncertainty Quantification, Optimization, and Data Analysis.

In general, PSUADE provides inputs to a simulation and compares it to the outputs from the simulation. It stores all of these results in a file, by default called psuadeData, which has all of the information about the inputs and the outputs. Generally, PSUADE will also output relevant information to the console that was requested by the user (for example, sensitivity analysis information).

Additionally, in general, if you're looking for advice about PSUADE and how to run it, launch the psuade executable and type help, then help (topic).

```
[lbrown@star1 ~]$ iel-2.0/EXTLIB/PSUADE/psuade_v1.7.2/build/bin/psuade
**************************************************************************
*      Welcome to PSUADE (version 1.7.2)
**************************************************************************
PSUADE - A Problem Solving environment for
         Uncertainty Analysis and Design Exploration (1.7.2)
(for help, enter <help>)
==========================================================================
psuade> help
Help topics:
```

```
info          (information about the use of PSUADE)
io            (file read/write commands)
stat          (basic statistics)
screen        (parameter screening commands)
rs            (response surface analysis commands)
qsa           (quantitative sensitivity analysis commands)
calibration   (Bayesian calibration/optimization commands)
plot          (commands for generating visualization plots)
setup         (commands to set up PSUADE work flow)
edit          (commands to edit sample data in loal memory)
misc          (miscellaneous commands)
advanced      (advanced analysis and control commands)
<command -h>  (help for a specific command)
```

1.3 Sensitivity Analysis / Uncertainty Quantification (SA / UQ)

For Sensitivity Analysis and Uncertainty Quantification, the code runs by generating all of the inputs at once, then running all of them in sequence. For these modes in particular, the code can actually generate all fo the inputs up front if you add "gen_inputfile_only" to the application section of the PSUADE input file.

These sorts of algorithms work by making small, measured changes in the inputs to a function/simulation, then analyzing the resulting output. The exact nature of these algorithms is unnecessary to discuss here. In general, sensitivity analysis algorithms work by making small changes in various inputs for a simulation to judge the relative strength that each input has on the output of the simulation. Uncertainty analysis tends to revolve around making small changes in the input to see how much the output is successfully determined by the inputs.

1.4 Optimization

Optimization runs slightly different. The inputs of the next iteration are determined by the output of the previous iteration, in an effort to optimize some quality of the run. For example, if you measure the error or convergence of the results, by choosing to minimize this quality, the optimization algorithm will attempt to generate inputs that minimize the error or convergence.

One thing we're interested in doing is using the optimization functions to minimize the runtimes of our codes. This has a clear purpose when running codes with a genetic algorithm or MCMC that run to convergence.

1.5 Data Analytics

PSUADE can also be used to analyze data. You can load psuadeData files in psuade by going into the PSUADE API (launching the PSUADE binary) and typing load (see

“psuade¿ help io” details). This allows you to generate plots (psuade¿ help plot), or do other forms of analysis (psuade¿ help rs or psuade¿ help qsa).

We haven't done much with this, but its inclusion as a feature of psuade seems necessary to me.

We have done several examples with PSUADE.

1.6 make test

The most notable one, in my book, has been the ”make test” command. PSUADE has 14 tests included with the package. They can be accessed by going into the build directory and typing

```
make test
```

Along with any wrapper things you want to do like making it a background task or capturing the output.

Here's how it looks when I run it. Your results may vary, but should probably only vary in runtimes, not in what succeeds and what doesn't.

```
Running tests...
Test project /home/lbrown/iel-2.0/EXTLIB/PSUADE/psuade_v1.7.2/build
      Start  1: pdf1
 1/14 Test  #1: pdf1 ............................   Passed   24.91 sec
      Start  2: pdf2
 2/14 Test  #2: pdf2 ............................   Passed   24.88 sec
      Start  3: pdf3
 3/14 Test  #3: pdf3 ............................   Passed   24.94 sec
      Start  4: pdf4
 4/14 Test  #4: pdf4 ............................   Passed   25.33 sec
      Start  5: pdf5
 5/14 Test  #5: pdf5 ............................   Passed   24.87 sec
      Start  6: pdf6
 6/14 Test  #6: pdf6 ............................   Passed   25.75 sec
      Start  7: pdf7
 7/14 Test  #7: pdf7 ............................   Passed   24.79 sec
      Start  8: ARSM1
 8/14 Test  #8: ARSM1 ...........................   Passed    0.04 sec
      Start  9: Bungee
 9/14 Test  #9: Bungee ..........................   Passed    0.76 sec
      Start 10: Morris20MOAT
10/14 Test #10: Morris20MOAT ....................   Passed    0.90 sec
      Start 11: Morris20LH
11/14 Test #11: Morris20LH ......................   Passed  144.31 sec
      Start 12: MCMCTest
```

```
12/14 Test #12: MCMCTest .......................   Passed    0.26 sec
      Start 13: OptRosenbrockM
13/14 Test #13: OptRosenbrockM ..................   Passed    0.89 sec
      Start 14: MOO
14/14 Test #14: MOO .............................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 14

Total Test time (real) = 322.64 sec
```

1.7 Rosenbrock function

We also did analysis of the rosenbrock function,

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

.

The results themselves were not particularly relevant, but it worked as a proof-of-concept.

```
PSUADE
INPUT
   dimension = 2
   variable   1 x1 =   -4.00000000e+00   4.00000000e+00
   variable   2 x2 =   -4.00000000e+00   4.00000000e+00
#####    Inputs for the example given by PSUADE    #####
##   variable   1 x1 =   4.00000000e+01   6.00000000e+01
##   variable   2 x2 =   6.70000000e+01   7.40000000e+01
##   variable   3 S =   2.00000000e+01   4.00000000e+01
END
OUTPUT
   dimension = 1
   variable   1 H
END
METHOD
   sampling = MC
   num_samples = 1000
#  for other options, consult manual
END
APPLICATION
   driver = ./simulator
#   gen_inputfile_only
#  driver = ./simulator.py
END
```

```
ANALYSIS
#  for analyzer and optimization options, consult manual
#  analyzer method = Moment
#  analyzer method = MainEffect
#  analyzer rstype = MARS
   diagnostics 1
END
END

/*************************************************
 * Main program
 *************************************************/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

main(int argc, char **argv)
{
   int    ii, nInputs;
   double *X, Y=1.0e35;
   FILE   *infile, *outfile;

   infile  = fopen(argv[1], "r");
   if (infile == NULL)
   {
      printf("ERROR - cannot open input file.\n");
      exit(1);
   }
   fscanf(infile, "%d", &nInputs);
   if (nInputs <= 0)
   {
      printf("ERROR - nInputs <= 0.\n");
      exit(1);
   }
   X = (double *) malloc(nInputs*sizeof(double));
   for (ii = 0; ii < nInputs; ii++)
      fscanf(infile, "%lg", &X[ii]);
   fclose(infile);

   Y = 100*(X[0] - X[1]*X[1])*(X[0] - X[1]*X[1]) - (1 - X[1])*(1 - X[1]);
    //Example generated by PSUADE
//Y = X[0] - 19.6 * X[1] / (1.5 * X[2]);
   outfile  = fopen(argv[2], "w");
```

```
   if (outfile == NULL)
   {
      printf("ERROR-cannot write to outfile.\n");
      exit(1);
   }
   fprintf(outfile, " %24.16e\n", Y);
   fclose(outfile);
}
```

The next step is to implement PSUADE so that it can work in parallel with other simulations and modules. We also want PSUADE to use tuple communications instead of file I/O for its work.

Fortunately, it appears we can do both at once.

```
I don't know whether this will useful : there is a psuade option called
'driver = psLocal' which will call a resident function inside psuade called
psLocalFunction. If you have a fixed function, you can compile it in psuade.
The psLocalFunction is in Src/DataIO/FunctionInterface.cpp.

If you have a fixed function that you call, you can put it
inside the psLocalFunction subroutine in Src/DataIO/FunctionInterface.cpp (that is,
replace everything inside that function with your code).
If you want to use your code every time a simulation is requested, you will
instantiate FunctionInterface and then call loadFunctionData

loadFunctionData(length, names)

and you set length = 5
and
names is a char**
with names[0] = 'PSUADE_LOCAL';
        names[1] = 'NONE'
        names[2] = 'NONE'
        names[3] = 'NONE'
        names[4] = 'NONE'

Then whenever you call FunctionInterface->evaluate, it will
call your local function.
```

It should be possible to substitute in the IEL_tget function as the 'fixed function' that will be called. Then PSUADE can treat other modules as fitness functions.

The next step will be to replace PSUADE's I/O function with IEL_tput. This shoudln't be too difficult.

```
psuade_v1.7.2/Src/DataIO
```

For the tput and tget changes to PSUADE, it should follow a similar process to the section in this document describing my changes to the main function.

2 R, RC

The RC module is fairly straightforward.

```
extern int R_running_as_main_program;   /* in ../unix/system.c */

//For debugging purposes, use main instead of RC, then run make executable
//int main(int ac, char **av)
int RC(IEL_exec_info_t *exec_info)
{
putenv("R_HOME=../../EXTLIB/R/inst/lib64/R");

char *a0 = "RCcar";
char *a1 = "--slave";
char *a2 = malloc(sizeof(char));

//Option to get the R script name from the command line, instead of just script.r
if(exec_info->modules[exec_info->module_num].mod_argc > 0){
char tmp[40];  strcpy (tmp, "--file=");
strcat (tmp, exec_info->modules[exec_info->module_num].mod_argv[0]);
a2 = tmp;
}

char** arguments = malloc(3*sizeof(char*));
arguments[0] = a0;
arguments[1] = a1;
arguments[2] = a2;

printf("Executing %s\n", arguments[2]);

SEXP aleph, beth;
R_running_as_main_program = 0; //May do nothing?
Rf_initialize_R(3, arguments);  //R --slave --file=(filename)
Rf_mainloop();

return 0;
}
```

Mostly what it does is set up the necessary environment, then calls the function Rf_mainloop(). The IEL configuration file will have to have the information about where the R script is. The first argument in the args section of the configuration file must be the relative path to the R script that you are interested in running.

The RC module currently doesn't use any tuple communication, and does not interact with other modules in any way. Fortunately, this should be easily remedied. R can use

the .C() function to call just about any C function. Converting C datatypes to R datatypes should be somewhat of a challenge, but there shouldn't be many challenges beyond that. It should be a very direct process to compile any C libraries into the R library, or compile them as a separate library to be loaded into R.