



LAWRENCE  
LIVERMORE  
NATIONAL  
LABORATORY

# Verification and Validation using DAKOTA via the DakTools scripts

Scott Brandon, Peter Tipton

December 14, 2004

Nuclear Explosives Code Developers' Conference 2004  
Livermore, CA, United States  
October 4, 2004 through October 7, 2004

## **Disclaimer**

---

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

## **Verification and Validation using DAKOTA via the DakTools scripts**

**Scott Brandon\* and Peter Tipton\*\***

\*Lawrence Livermore National Laboratory , \*\*University of Southern California

*Several of the intermediate capabilities which are being developed by the AX V&V program may be helpful in other ways. This paper describes a new PYTHON interface to one such tool, DAKOTA (a parallel optimizing controller from Sandia National Laboratory) and the subsequent simpler set of operations required to run and analyze sets of calculations using any LCC computational platform.*

### **Introduction**

A goal for the AX V&V Program is to develop a methodology and the corresponding computational tools which allow the quantification of the uncertainty in the results of calculational models to be determined. Several of the intermediate capabilities which are being developed to pursue this goal may be helpful in other ways. This paper describes a new interface to one such tool, DAKOTA (a parallel optimizing controller from Sandia National Laboratory) and the subsequent simpler set of operations required to run and analyze sets of calculations using any LCC computational platform.

### **DAKOTA (*Eldred et al., 2001 and Wojkiewicz et al., 2001*)**

DAKOTA is a massively parallel software tool kit which provides optimization, uncertainty quantification, and statistical analysis tools. DAKOTA is available on all of our computational platforms (both on clusters with just a few processors per node and on massively parallel machines). The interface between DAKOTA and the design code (often just a UNIX script file) requires no change in the application code and only small changes in the code input files. Once DAKOTA is linked to the application code, the full power of the available computational resources can be used to simultaneously run and analyze problems of interest. Thus, studies which require tens of thousands of hours of computer time can be completed in a time scale measured in days to weeks.

## **Running DAKOTA/Application Code Problems on Multiple Platforms**

When run on a single platform using a single execution model (for example each application code instance runs as a separate batch task), the UNIX program which links DAKOTA and the application code approaches the simplicity of a “typical” 3-line UNIX shell script. However, if one wishes to run on all LCC platforms, support multiple execution models (for example running each application code instance on a separate MPI task from a single batch job), control complicated post-processing procedures, and incorporate error detection and correction, the simple UNIX shell script coupling DAKOTA and the application code becomes complicated. Due to limitations of the shell language (in this case CSH), the resulting script will contain information specific to the application code and the user’s problem which is being executed. Thus, this CSH script can not be placed in a public location and maintained for a large group of users.

The above problems can be corrected by rewriting the DAKOTA/application code interface scripts in a programmable language, such as PYTHON. The PYTHON conversion of the our DAKOTA interface scripts is now nearly complete. A “beta” version of the new PYTHON based interface scripts is available in /usr/apps/dakota/test/daktools on most LCC platforms. These scripts greatly simplify the procedures needed by the user to run hundreds to thousands of calculations using the DAKOTA controller running on any or all LCC platforms.

## **The DakTools DAKOTA Interface**

The DakTools package consists of a set of tools that fit into the DAKOTA workflow environment. These tools maximize the user’s efficiency in studying code behavior by both providing machine independence and extending the DAKOTA variable definition section. This simple structure allows users to easily set up and run large sets of calculations, provides a mechanism to collect code and analysis tools created by the user community, and allows multiple options to analyze data; all with concurrent or parallel calculation capability.

Fig. 1 shows the components required to setup and analyze a set of calculations. The calculations are controlled by DAKOTA via the user supplied DAKOTA input file. With the exception of the variables section, which is omitted (it will be automatically created by the DakTools interface), this is any standard single\_method DAKOTA input file. The user supplied options script, the main interface for the DakTools package, specifies run options for both DAKOTA and the application code. Also supplied via the options file is an optional setup function and the required output function. The setup function takes care of any initialization that may be required by the application code. The output function defines and

returns values of the objective functions to DAKOTA. The remaining user supplied file is the input deck for the application code.

Dakota Workflow Diagram

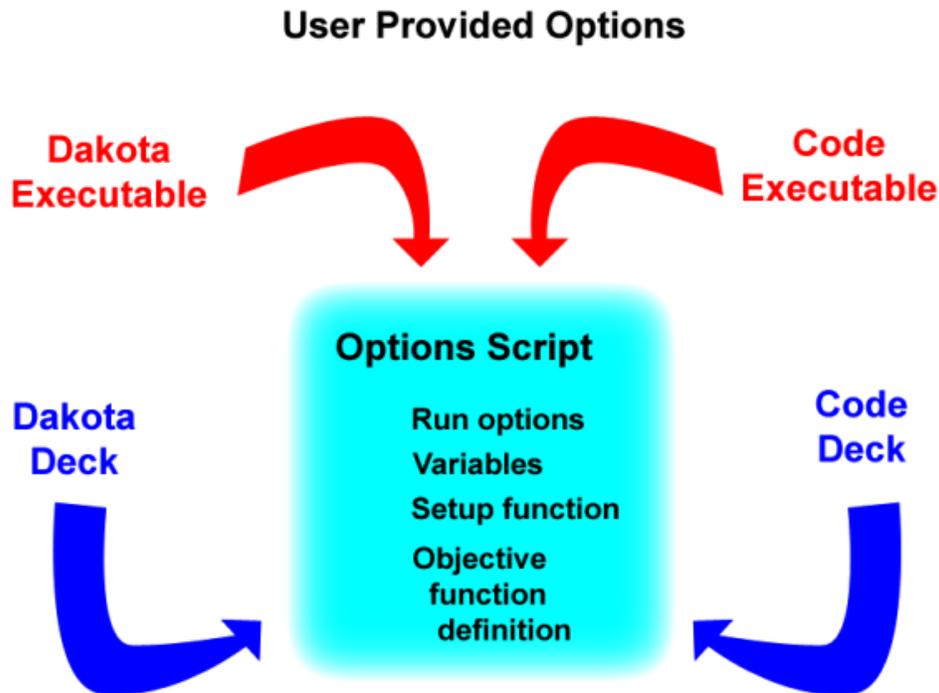


Figure 1: Dakota/DakTools workflow

The DAKOTA input file strategy (`single_method`) and method selection sections select which DAKOTA module will be used to control the application code. Classic optimization procedures often require knowledge of previous results to determine the choice of the next calculation. Hence most optimization procedures run sequentially (or with low order parallelism). These methods are poorly matched to today's massively parallel computing platforms. DAKOTA also offers a variety of sampling methods. These methods allow information to be collected via sets of calculations spanning a large portion of the relevant parameter space of the application code. Since each member of a parameter survey is independent of all the others, parameter studies are embarrassingly parallel. By using tens to thousands of CPU's, parameter studies can often be completed in the real time required for just one optimization cycle. Once the data set is available, traditional optimization amounts to post processing the precalculated data set. A second parameter study, with adapted parameter ranges, can be used to further refine the resulting set of calculations which now may all satisfy the given constraints. Examination of the parameter values which satisfy the constraint(s) may reveal degeneracies.

(Optimization strategies are likely to fail in the presence of degeneracies.) Calculations which came close to satisfying the constraints provide information about the behavior of the application code near the desired point. (Information which is seldom obtained via standard optimization approaches.)

## **Introduction to the DakTools Interface to DAKOTA**

The DakTools interface consists of nine PYTHON scripts augmented by three user supplied files. A short description of the DakTools Interface files follows:

- daktools.py: main entry point for the DakTools/CODE interface tools
- cleardir: automates removal of DakTools generated files
- coderun.py: executes CODE with DAKOTA variables, returns objective functions (can be user supplied)
- codetools.py: utility functions to help handle application codes (can be user supplied)
- datatools.py: utility functions to help process CODE results
- interface.py: provides the interface between DAKOTA and CODE
- killjob: automates deletion of running DAKOTA/CODE jobs
- rundict.py: expands %%key%% constructs via a dictionary (general purpose utility)
- runexe.py: controls interactive, batch, or parallel execution of it's argument (general purpose utility)

The three user supplied files are as follows:

- options.py: global options[arguments and variables] for daktools.py
- dakota.input: DAKOTA input [no variables section]
- code.deck: input deck template for CODE

In addition, there is an example options.py file and several complete working example DAKOTA/CODE job files stored in the Examples subdirectory. Most of the PYTHON files are at least partially self documenting [from PYTHON, import the desired module and print module.\_\_doc\_\_]. Executing daktools.py with no arguments causes daktools to print a short list of available arguments

## *Proceedings from the NECDC 2004*

The first step to use the DakTools interface is to set up the problem you wish to investigate using the CODE you wish to use. Choose the parameters you wish DAKOTA to vary and put together the DAKOTA input file. The DakTools interface supports all DAKOTA single strategy methods. The interface portion of the DAKOTA input is nearly standardized and the variables portion is created via the DakTools interface. Hence, the user primarily modifies only the method and responses portion of the DAKOTA input file. A typical LHS parameter study DAKOTA input deck is shown in Fig. 2.

```
# totalb.input - DAKOTA input for totalb LHS parameter study

strategy,                                     \
    single_method                             \

method,                                       \
    nond_sampling, all_variables              \
    samples = 1000                            \
    seed = 5                                  \
    sample_type lhs                            \

interface,                                    \
    application system,                       \
    analysis_driver = 'interface.py'         \
    parameters_file = 'info.in'              \
    results_file    = 'info.out'             \
    aprepro                                                 \
    file_tag                                                 \
    file_save                                                 \

responses,                                    \
    num_objective_functions = 3                \
    no_gradients                                                 \
    no_hessians
```

Figure 2: A DAKOTA input file for TOTALB calculations using the DAKOTA Latin Hypercube Sampling (LHS) method.

The next step is to modify the code.deck input file to accept the variables to be controlled via DAKOTA. This is accomplished by replacing occurrences of “variable\_one” wherever it occurs in the CODE input file with the “%%variable\_one%%” flag. DakTools will convert each occurrence of the flagged variables with the DAKOTA generated values. More complete initializations can be accomplished via the “setup” procedure defined within the “options.py” file. An example TOTALB (*Adams et al., 2002*) input file is shown in Fig. 3.

## *Proceedings from the NECDC 2004*

```
Infile  tot001001_n5
Outfile a10.out
Profile tot.xy
Atomw   1.0079
Zrad    0.0
Zp      1.0
B       0.0
Ne      %%Ne%%
Te      %%Te%%
Range   10.195 10.215
Nmc     50
Grid    4096
Reduce  all onlyn
Runtype real
Angc    -1
Pol     2
Doppler
end
```

Figure 3: A DAKOTA input file for TOTALB calculations.

Finally, prepare the “options.py” file. This is the main interface to the DakTools package and provides argument options and input variables to the daktools.py script. Many of the available options have default values. However, you must always at least specify the code\_deck and code\_path arguments. It is in this file that you choose how DAKOTA and the application CODE is to be run [interactive/batch, parallel/serial]. The “setup” function is optional. But you must specify the output function which is used to return objective function values to DAKOTA. The arguments and variables portion of a “options” file corresponding the the TOTALB LHS parameter study is shown in Fig. 4. The setup and output functions for TOTALB LHS parameter study are shown in Fig. 5. DakTools is being told to execute DAKOTA using 16 processors on the current HOST (Note: HOST must be a parallel machine!).

All required input files are now setup and ready to be tested. Use the following command to execute daktools.py and test much of the file syntax:

```
./.../daktools.py -norun options.py
```

This will cause most of the files required to run your problem to be created. The “-norun” option prevents any jobs from being actually run. You are free to examine various files and determine if your problem will do what you expect.

Your problem is executed by entering the following command:

## *Proceedings from the NECDC 2004*

```
#####
#----- global options provided by client
#####
import commands, os, sys

### OPTIONS ###
#### most have defaults and all can be set via command line arguments
arguments = {

'problem_name' : 'totalbLHS',

##### USER PROVIDED FILES
'dakota_deck'      : 'totalbLHS.input',
'code_deck'       : 'tot.inp',

##### DAKOTA
'dakota_path'     : '/g/g14/brandon/Dakota/Dakota_3.1.2/Dakota/bin/dakota',
'dakota_mode'     : 'batch', #interactive | batch
'dakota_procs'   : '6',

##### CODE
'code_path'       : '/nfs/tmp3/brandon/totalb/totalb',

##### MISC
'timeout'         : '600' # timeout for batch submission

}

### Simulation Variables ###
variables = r"""
Ne          1.0e+12 1.0e+14 1.0e+17   = log
Te          1.0    2.0    5.0        = lin 0 1 2
"""
```

Figure 4: The “args” and “variables” definition sections from the TOTALB DakTools options file.

```
./.../daktools.py options.py
```

If you wish you may add other arguments such as:

```
./.../daktools.py -code_mode="interactive" options.py
```

The execute line options over ride the values (if any) set within the options.py user supplied file.

*Proceedings from the NECDC 2004*

```
def setup():
    os.system('ln -fs ../tot001001_n5 .')

    return

def output():
    import codetools, glob

    #####writout output file for dakota

    # ..... replace the following with py_tabnormu
    tabnormu = '/g/g14/brandon/HDG_east/hodag/tools/'+os.getenv('SYS_TYPE')+'/bin/tabnormu'
    os.system('ln -s ../tot.tas .')
    os.system('%s tot > taboutput' % tabnormu)

    tot_dif = glob.glob("*tot.dif")
    if len(tot_dif) > 0:
        # ..... tabnormu norm data exists
        line = commands.getoutput('cat tot.dif | grep ONE')
        f1 = float(line.split()[2])
        line = commands.getoutput('cat tot.dif | grep TWO')
        f2 = float(line.split()[2])
        line = commands.getoutput('cat tot.dif | grep MAX')
        f3 = float(line.split()[2])
    else:
        f1 = 1.0e+99
        f2 = 1.0e+99
        f3 = 1.0e+99

    f = [f1,f2,f3]

    return f
```

Figure 5: The “setup” and “output” function definitions from the TOTALB DakTools options file.

The DakTools interface runs each application code instance in its corresponding run directory. The run directories are named “run0001-runNNNN”. The setup and output functions can take advantage of this organization by linking files from the parent directory into the run directory. For example the command, “ln -s ../\*.py .”, will cause all PYTHON files in the parent directory to be soft linked to the run directory. The output function, which is always executed after the application code completes, can clean up files left behind by the application code. If the application

code data files are left intact, the `daktools.py` option, “-pp” is available to run the output function over the complete data set. This option can run the post processor (controlled by the output function) concurrently over the entire data set; reducing the time and effort required to analyze large DAKOTA/application CODE data sets.

The `daktools.py` script is designed to be executed using the complete file path specification of the `daktools.py` file [usually somewhere in `/usr/gapps/dakota`] on the host you wish to run DAKOTA and the application code. If you alter your “path” variable to access `daktools.py`, you must insure that the directory “.” occurs earlier in your “path” variable than the `daktools.py` directory. If this is not the case, you will get PYTHON errors complaining about “%%key%%” constructs! Having DAKOTA or the application code run (batch only) on a different host than the one you execute `daktools.py` is possible; and sometimes very handy. Occasionally there are machines available to run the application CODE, but with no DAKOTA version available. In this case, run DAKOTA on an alternate machine and use the “code\_host” (set to the desired host) and “code\_env” (set to ‘batch’) commands to run the application CODE on the new machine. Note: running the application CODE in the batch mode requires some changes to be made in the DAKOTA input file. In the “interface” section change the line “application system” to “application asynchronous system” and add an additional line to determine the desired evaluation concurrency (“evaluation\_concurrency = 6”). DAKOTA will then monitor the progress of the application CODE batch tasks and try to keep evaluation\_concurrency of them running.

## **A DakTools Example**

We wish to run the Sedov verification problem in 2-D using CALE (*Tipton, 1998*) in both the Lagrange and ALE modes and for multiple resolutions. The CALE input syntax allows the mesh resolution to be specified via a parameter in the input file. We choose to allow DakTools to directly manipulate this mesh resolution variable. We will choose the DAKOTA variable “nzones” to represent the desired mesh resolution. Specifying the use or absence of ALE operations presents more of a challenge. There are multiple ALE commands which naturally occur in several places within the CALE input deck. We choose to include all of the ALE commands, prefixed with a comment (“\*Ale ”), in the Lagrange input file. By default CALE will then run Lagrangian. We will give our “setup” function to task to remove the “\*Ale “ comments when an ALE result is desired. We choose the DAKOTA variable “nale” to represent Lagrangian (nale=0) or ALE (nale=1) calculation modes. We will use an arbitrary set of mesh resolutions; suggesting the DAKOTA “list\_parameter\_study” method. This is sufficient information to complete the DAKOTA input file which we will call “Sedovc\_2d.input”. This file is shown in Fig. 6.

*Proceedings from the NECDC 2004*

```
# Sedovc_2d.input - DAKOTA script for CALE 2-D Sedov verification problems
```

```
strategy, \
    single_method \

method, \
    list_parameter_study \
        list_of_points = \
            16 0 \
            16 1 \
            23 0 \
            23 1 \
            45 0 \
            45 1 \
            90 0 \
            90 1 \
            180 0 \
            180 1 \
            360 0 \
            360 1 \
            #720 0 \
            #720 1 \

interface, \
    #application system, \
    application asynchronous system, \
    evaluation_concurrency = 6 \
    analysis_driver = 'interface.py' \
    parameters_file = 'info.in' \
    results_file = 'info.out' \
    aprepro \
    file_tag \
    file_save \

responses, \
    num_objective_functions = 1 \
    no_gradients \
    no_hessians \
```

Figure 6: A DAKOTA input file for CALE Sedov problems using the “list” method.

We also have enough information to complete the CALE input file, Sedovc\_2d. This file is shown in Fig. 7. Note that the ifadvec variable is now initialized by “%%nale%%” which will be expanded automatically by DakTools. Likewise, the NZ

*Proceedings from the NECDC 2004*

variable is now initialized by “%%nzones%%” which will also be expanded automatically by DakTools.

```
*****
*
* Sedovc_2d (blast wave)
*
* Two dimensional model of the Sedov Blast Wave test problem
*
* Models the blast on full plane in cylindrical geometry
*
*****

*
* set parms
*
* .....
* ..... hydro controls
  set ifadvec  %%nale%%          * Lagrange/ALE mode (0 is Lagrangian)
* .....
  def      NZ  %%nzones%%          * nzones
* .....

*
*Ale gridmove relax 1 1 KX LX-          * relax all nodes
*Ale ale      kntrp2 1 1 2 {max( 3, (5 * NZ ) / 45)} 1.0
*
* .....
run

showcpu

pampfile Sedovc_2d.u
tpamp
* .....
```

Figure 7: A portion of the CALE input file for 2-D Sedov test problems.

We are now ready to complete the user supplied options file which we will name “Sedovc\_2d\_options.py”. We must overcome several challenges to complete this task. First, we choose to run CALE on the LCC ILX cluster. ILX is a cluster of servers, hence the normal DAKOTA MPI mode is not available. We will have to run the CALE calculations as a separate batch task. Second, ILX was recently converted to the CHAOS II operating system and a ILX version of DAKOTA is not yet available. However, the RedHat version of DAKOTA will still run, as long as we

don't try to run it as an MPI parallel task! Third, a substantial amount of preprocessing is required to control file names and the ALE option. These tasks will all be handled via our "setup" function. Finally, a significant amount of post processing is also required. We must insure that all files are uniquely named, calculate the Ln norms between the CALE calculation and the analytical solution, and move all files from the "run" directories to the more permanent "problem" directories. The options and variables section of the "Sedovc\_2d\_options.py" file is shown in Fig. 8, the setup function is shown in Fig. 9 and the output function is shown in Fig. 10.

Information is communicated from DAKOTA to the application CODE via input files, always called "info.in.nnn" by DakTools. If any of the DAKOTA variables must be used by the "setup" or "output" functions, they must either parse the "info.in.nnn" files for this information or take advantage of hints left behind by the application CODE. Our "setup.py" function uses a grep—awk pipe to determine the values of "nzones" and "nale". A private PYTHON utility, `re_string` (Regular Expression string replacement), is used as a simplified SED to manipulate the CALE input deck. The "output" function obtains the value of "nzones" and "nale" (now called "mod") by parsing the name of the "Sedov\_2d[ab]\_nzones.success" file. This file is only created when the problem successfully completes; providing a simple test on whether or not CALE completed the problem. (In this example, CALE will fail to complete the test problem when run in the Lagrange mode for resolutions greater than `nzones=180`.) Another private PYTHON utility function, `tabnormu`, is used to calculate the Ln norms between the CALE and analytic solutions.

The `codetools.py` and `datatools.py` DakTools scripts are collections of general purpose utilities intended to simplify the data analysis and code setup functions appearing in the DakTools option files. Both of the previous private PYTHON utility functions, `re_string` and `tabnormu`, are candidates for inclusion into the DakTools `codetools.py` or `datatools.py` scripts. Once a utility is added to either of these PYTHON files, it becomes available to all other DakTools users. The ability to easily add user created PYTHON modules to the DakTools collection provides an important mechanism to improve the utility of the DakTools environment.

ILX is fairly heavily loaded. This example restricts the concurrency to 6. Often the batch system will deliver a concurrency less than 6. Normally, a concurrency of between 3-6 is observed. Hence, with the penalty of "wasting" a processor to run DAKOTA, the speedup offered by using this DakTools script is between 3-6 and limited only by the capacity of the machine and the number of desired calculations.

To achieve speedups of 100-1000, you must run on one of the MPI parallel machines, such as MCR. The maximum run time and the availability of resources are both quite restricted. However, obtaining 100 processors for 12 hours at a time is fairly reliable. Dedicated access time (available each weekend) is needed to run at

## *Proceedings from the NECDC 2004*

```
#####
#----- global options provided by client
#####
import commands, os, sys

### OPTIONS ###
#### most have defaults and all can be set via command line arguments
arguments = {

'problem_name' : 'Sedovc_2d',

##### USER PROVIDED FILES
'dakota_deck'      : 'Sedovc_2d.input',
'code_deck'       : 'Sedovc_2d',

##### DAKOTA
'dakota_path'     : '/usr/gapps/dakota/redhat_7a_ia32/dakota',
'dakota_mode'     : 'batch', #interactive | batch
'dakota_procs'   : '0',

##### CODE
'code_path'       : '/usr/gapps/cale/'+os.getenv('SYS_TYPE')+'/cale',
'code_mode'       : 'batch',
'code_tM'        : '1200',

##### MISC
'timeout'         : '600' # timeout for batch submission
}

### Simulation Variables ###
variables = r"""
nzones          45      45      720    # mesh resolution (list method wins)
nale            0       0       1     # ALE cycles (list method wins)
"""
```

Figure 8: The options and variables section of the CALE Sedov DakTools options file.

levels of 1000 (or more) processors. A factor of 100 (or more) speedup is enough to make large scale parameter studies a viable alternate to even highly effective optimization strategies.

## *Proceedings from the NECDC 2004*

```
def setup():
    import codetools, commands, string

    os.system('ln -fs ../re_string.py .')
    os.system('ln -fs ../Sedov.tas .')
    os.system('ln -fs ../py_tabnormu.py .')

    import re_string

    # ..... finish updating/creating CALE input file
    nzones_str = commands.getoutput("grep nzones info.in* | awk '{print $4}'")
    nzones = int(string.atof(nzones_str))
    nale_str = commands.getoutput("grep nale info.in* | awk '{print $4}'")
    nale = int(string.atof(nale_str))

    inDict = {'*end':'end',                # stop when run is complete
              ' tv':'* tv'}               # disable interactive plots
    if nale == 1:
        # ..... this is an ALE problem
        mod = 'b'
        inDict['*Ale'] = ''                # enable ALE commands
    else:
        # ..... this is a LAGRANGE problem
        mod = 'a'
    outName = "Sedovc_2d" + mod + ("_%i" % nzones)
    inDict['Sedovc_2d'] = outName          # set problem name
    PWD = os.getenv('PWD')
    prob_dir = PWD[:PWD.find("/run")] + '/Sedovc_2d' + mod
    inDict['prob_dir'] = prob_dir
    re_string.rpl_file("Sedovc_2d", inDict, outName)

    return 'time %%exe%% ' + outName      # define new 'code_cmd'
```

Figure 9: The “setup” function definition from the CALE Sedov DakTools options file.

## **Summary**

This paper describes DakTools, a new PYTHON interface to DAKOTA (a parallel optimizing controller from Sandia National Laboratory). DakTools frees the user from many of the harsh realities encountered while running on disparate LCC platforms. The subsequent simpler set of operations required to concurrently run and analyze sets of calculations, using any available LCC platform, allows the user to more easily take advantage of today’s large, parallel computational environments.

## **References**

- [1] Adams, M.L., Lee, R.W., Scott, H.A, Chung, H.K., and Klein, L., "Complex atomic spectral line shapes in the presence of an external magnetic field," Phys. Rev. E, 66, Number 6-2, pp. 066413 1-12 (2002).
- [2] Eldred, M.S., Giunta, A.A., van Bloemen Waanders, B.G., Wojkiewicz, S.F. Jr., Hart, W.E., and Alleva, M.P., "DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis," Version 3.0 Reference Manual, Sandia National Laboratories, SAND 2001-3515 (2001).
- [3] Robert Tipton, "CALE Users Manual," LLNL (1998) [export controlled].
- [4] Wojkiewicz, S.F., Eldred, M.S., Field, F.V. Jr., Urbina, A. and Red-Horse, J.R., "Uncertainty Quantification in Large Computational Engineering Models," American Institute of Aeronautics and Astronautics, AIAA-2001-1455 (2001).

## **Acknowledgments**

This work was performed under the auspices of the Department of Energy by the University of California Lawrence Livermore National Laboratory under contract No. W-7405-ENG-36.

## *Proceedings from the NECDC 2004*

```
def output():
    import codetools, commands, glob, os, string, py_tabnormu

    # ..... determine problem name and "mod"
    successFile = glob.glob("*Sedovc_2d*.success")[0]
    if successFile == []:
        # ..... stop if error
        print "          Run "+os.getcwd()+" has failed"
        return [1.0e+99]

    i1 = successFile.find("_2d")
    i2 = successFile.find(".success")
    base = successFile[0:i1+3]
    mod = successFile[i1+3:i1+3+successFile[i1+3:].find("_")]
    prob = successFile[0:i2]
    strNzones = successFile[i1+3+successFile[i1+3:].find("_")+1:i2]
    print " * strNzones = '%s' " % strNzones
    nzones = string.atoi(strNzones)
    print " * base='%s', mod='%s', prob='%s', nzones='%d' " % \
        (base, mod, prob, nzones)

    # ..... rename remaining generic CALE files
    os.rename("cale.cgm00", prob+".cgm00")
    os.rename("calehsp", prob+"hsp")
    os.rename("calez", prob+"z")
    os.system("mv cale*.silo %s_final.silo" % prob)

    # ..... calculate Lnorms
    py_tabnormu.tabnormu(prob, "Sedov", 1.125, nzones, 1)

    if mod <> '':
        # ..... copy output files into standard directories
        os.system('pwd;ls')
        try:
            # ..... create any missing directories
            os.mkdir("../%s" % (base+mod))
        except:
            pass
        os.system('cp %s* ../%s' % (prob, (base+mod)))

    return [1.0]          # stub for now
```

Figure 10: The “output” function definition from the CALE Sedov DakTools options file.