Optimizing Alya simulations of multi-physics problems with Dakota

Rogeli Grima

December 16, 2014

Contents

1 Introduction

Dakota is a toolkit that provides a flexible, extensible interface between analysis codes and iterative systems analysis methods. It contains algorithms for optimization, uncertainty quantification, parameter estimation and sensitivity/variance analysis.

Alya is a computational method that is used to simulate complex physical systems. On many occasions we want to use these simulations to find an acceptable or optimized solution for a particular system. Dakota will help us to use Alya as a design tool. It will help us to solve very important questions as: What is the best design? How safe is it? How much confidence do I have in my answer?

This user's guide is intended to provide some background information on how to use Dakota to solve optimization problems that involve a simulation with Alya. We assume the user has some familiarity with Alya execution and configuration.

In this guide we will show how to use Dakota in Marenostrum, how to prepare your simulations with Alya, how to process their outputs and we will provide some basic examples.

1.1 Dakota workflow

Before you start working with Dakota and Alya, is important to know the Dakota's workflow. This will give you a better perspective of what is happening.

Dakota receives input and configuration from a variety of text files prepared by the user, and connects to any simulation code by other text files. This simple interface is one of the most important features of Dakota, as it makes it easy to change the iterative method or strategy by just changing a few lines in the Dakota input file.

In Figure 1, we can see the Dakota workflow. The user should provide an input file to Dakota. This input file controls the algorithm that we want to run, the function to be evaluated, the variables of this function and the outputs that we expect.

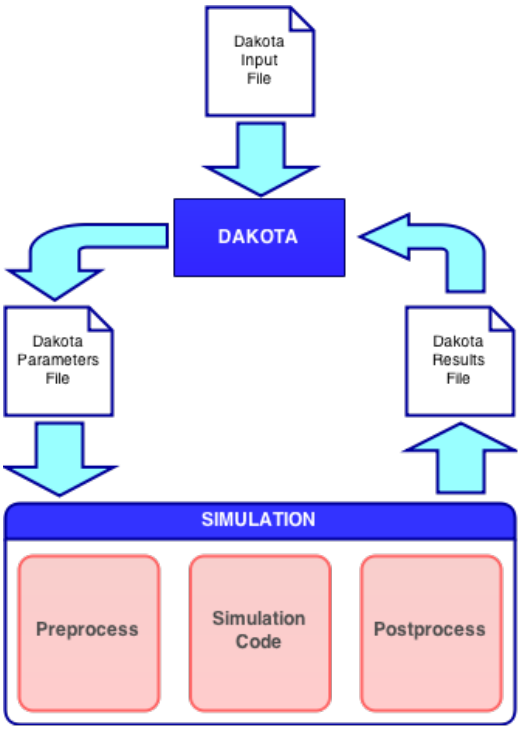

Figure 1: Dakota Workflow

Dakota generates a parameter file for every function evaluation. This file contains a specific value of every variable and specifies the kind of returning values that it needs (function evaluation, derivatives or/and second derivatives).

Dakota treats the simulation code as a black box. Most of the codes, like Alya, don't know what to do with the parameter file, so it is common to create a script to connect the simulation code with Dakota. In this script we merge the Dakota parameters file with the simulation code input files. Once we have some proper input files we can run the simulation code, extract the significant data from its output files and post-process that data in order to generate the appropriate objective function and the Dakota results file.

2 Dakota tutorial

2.1 Set up Dakota

We have installed in Marenostrum the last three versions of Dakota. Althought, most of the experiments that we have ran have been done using version 5.4 we encourage you to use the

last version. From now on, all the information that we will provide you in this guide will refer to version 6.1. You can find the several installed versions of Dakota in:

- */gpfs/projects/bsc21/DAKOTA/dakota-5.4.0*
- */gpfs/projects/bsc21/DAKOTA/dakota-6.0.0*
- */gpfs/projects/bsc21/DAKOTA/dakota-6.1.0*

If you want to install your own version of Dakota in Marenostrum, please check Appendix A.

In order to run Dakota you will need to set the path of the executables and the directory of the shared libraries. You can add these lines to your .barshrc file:

```
export DAK_INSTALL=/gpfs/projects/bsc21/DAKOTA/dakota-6.1.0
export PATH=${DAK_INSTALL}/bin/:${PATH}
export LD_LIBRARY_PATH=${DAK_INSTALL}/bin:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH=${DAK_INSTALL}/lib:${LD_LIBRARY_PATH}
```

Check that everything is working properly by running:

```
dakota -v
```

2.2 Runing Dakota with a simple input file

This section is intended for users who are new to Dakota, to demonstrate the basics of running a simple example.

1. Create a working directory.
2. From path */gpfs/projects/bsc21/DAKOTA/Examples/power*, copy files *power_multidim.in* and *power.py* to the working directory.
3. From the working directory, run: *dakota -i power_multidim.in -o power_multidim.out > power_multidim.stdout*

Dakota outputs a large amount of information to help users track progress. Four files should have been created:

1. The screen output has been redirected to the file power_multidim.stdout. The contents are messages from Dakota and notes about the progress of the iterator (i.e. method/algorithm).
2. The output file *power_multidim.out* contains information about the function evaluations.
3. *power_multidim.dat* is created due to an specification of the input file. This summarizes the variables and responses for each function evaluation.
4. *dakota.rst* is a restart file. If a Dakota analysis is interrupted, it can be often be restarted without losing all progress.

This example used a parameter study method and the *power.py* test problem. The Python script *power.py* reads a Dakota parameters file, computes the function $F(x, y) = (X - 0.5)^2 + (Y + 0.5)^2$ and writes the result in a Dakota results file.

As we said, this is a parametric study. Let's try to execute an optimization problem. Now, copy file *power_mas.in* to your working directory and run:

```
dakota -i  power_mas.in -o power_mas.out > power_mas.stdout
```

This executes a mesh adaptive direct search algorithm. If you look at the output file you will see that the method converges in 160 iterations and propose as a best objective function a value of $F(X,Y) = 0.0$ evaluated for $X = 0.5$ and $Y = -0.5$. That is the minimum of the evaluated function.

You can see a graphic representation of the results executing the script *visualize.sh* that you can find in the same directory. This scripts reads the results from the parametric study and from the mesh adaptive direct search algorithm.

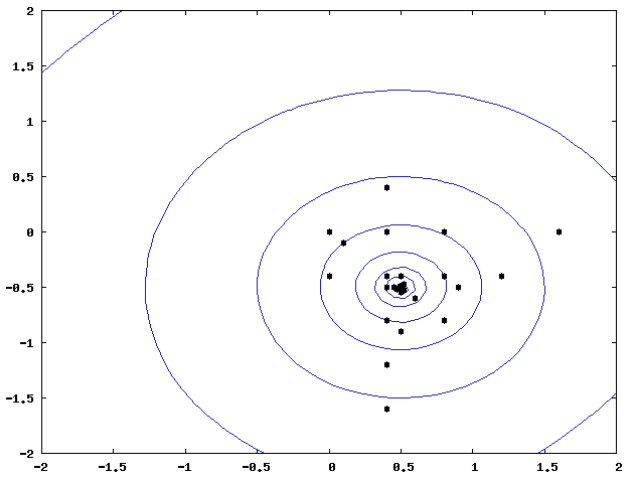

Figure 2: Mesh adaptive direct search evaluation points

3 Dakota files

3.1 Dakota input file format

There are six sections in every Dakota input file. These sections are identified with the following keywords: *variables*, *interface*, *responses*, *model*, *method*, and *environment*. At least one *variables*, *interface*, *responses*, and *method* must appear, and no more than one *environment* should appear.

Figure 3 shows the relationships between the six keyword blocks. The environment specifies high level Dakota settings, and identifies the top level method. A method runs a model. A model block defines the connections between variables, the interface, and responses. It shows the most common relationships between blocks but others are possible. Most Dakota analyses just needs to define a single method which runs a single model.

For a more concrete example, a simple Dakota input file, *power_multidim.in*, for a two-dimensional parameter study on $F(x,y) = (X - 0.5)^2 + (Y + 0.5)^2$ function is shown in Figure 4. This input file will be used to describe the basic format and syntax used in all Dakota input files.

First, some syntax background:

- Blocks can follow any order.
- Comments starts with symbol #.
- Use of single or double quotes for string inputs.

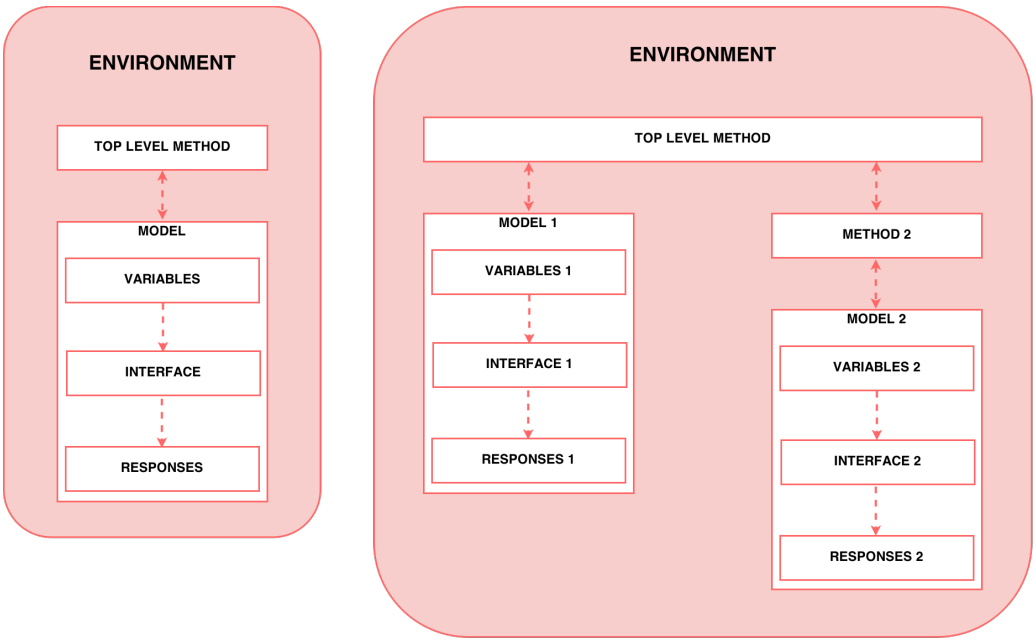

Figure 3: Relationship between the six blocks

```
environment
  tabular_graphics_data
    tabular_graphics_file = 'power_multidim.dat'

method
  multidim_parameter_study
    partitions = 8 8

model
  single

variables
  continuous_design = 2
    lower_bounds      -2.0     -2.0
    upper_bounds       2.0      2.0
    descriptors        "X"      "Y"

interface
  fork
    analysis_driver = './power.py'
    parameters_file = 'params'
    results_file    = 'results'
    file_save

responses
  response_functions = 1
  no_gradients
  no_hessians
```

Figure 4: File power_multidim.in

- Use of commas and/or white spaces for separation of specifications.
- The optional use of symbol = to indicate supplied data.

The first block that we can find in the file is *environment*. This keyword is used to specify the general Dakota settings such as Dakota's graphical output and the tabular data output (via the *tabular_graphics_data* keyword).

The *method* block of the input file specifies which iterative method Dakota will employ, such as a parameter study, optimization method, data sampling technique, etc. The keyword *multidim_parameter_study* calls for a multidimensional parameter study, while the keyword *partitions* specifies the number of intervals per variable. In this case, there will be eight intervals (nine data points) evaluated between the lower and upper bounds of both variables (bounds provided subsequently in the variables section), for a total of 81 response function evaluations.

The *model* block of the input file specifies the model that Dakota will use. It provides the logical unit for determining how a set of variables is mapped into a set of responses in support of an iterative method. The model allows one to specify a single interface, or to manage more sophisticated mappings involving surrogates or nested iteration. Most of the time we are going to use the *single* model, that is the default value for *model* and its definition can be omitted.

The *variables* block of the input file specifies the characteristics of the parameters that will be used in the problem formulation. The variables can be continuous or discrete, and can be classified as design variables, uncertain variables, or state variables. In figure 4 you can see as we have defined two continuous design variables, with their upper and lower bounds and a name: X and Y.

The *interface* block of the input file specifies what approach will be used to map variables into responses as well as details on how Dakota will pass data to and from a simulation code. In this example, the keyword *fork* is used to indicate the use of a user-supplied program. Then we find four keywords:

- *analysis_driver:* indicates the name of the program to execute. In this case, it is going to use the python program *power.py*.
- *parameters_file:* name of the dakota parameters file that the driver would receive from Dakota.
- *results_file:* name of the results file that the driver should return to Dakota once it has finished.
- *file_save:* It's telling Dakota to not delete parameters and results files once the driver has finished.

Dakota will execute the external program like this:

```
./power.py params.$i results.$i
```

Where $i is the iteration number of the method.

The *responses* block of the input file specifies the types of data that the interface will return to Dakota. For our example, the assignment *num_objective_functions = 1* indicates that there is only one objective function. Since there are no constraints associated with Rosenbrock's function, the keywords for constraint specifications are omitted. The keywords *no_gradients* and *no_hessians* indicate that no derivatives will be provided to the method; none are needed for a parameter study.

3.2 Dakota parameters file format

Prior to invoking a simulation, Dakota creates a parameters file which contains the current parameter values and a set of function requests. Let's execute the same example that we ran in section 2.2, but first, uncomment the line that contains *file_save*. This will prevent Dakota

to remove the temporal files that it uses to communicate with the simulation program. If you list your working directory you will see 81 parameters files (params.*) and 81 results files (results.*). This is how it looks like the content of file *params.1*:

```
                                    2 variables
 -2.000000000000000e+00 X
 -2.000000000000000e+00 Y
                                    1 functions
                                    1 ASV_1:response_fn_1
                                    2 derivative_variables
                                    1 DVV_1:X
                                    2 DVV_2:Y
                                    0 analysis_components
                                    1 eval_id
```

This file can be interpreted as a sequence blocks, where every block contains an array.

- First we find information about variables: the number of variables (2) followed by the variable values and names ($X = -2.0$ and $Y = -2.0$).
- The second block contains information of the functions: number of functions (1) followed by kind of response. This information is coded in ASV_i. It can be value, gradient and/or Hessian.
- The third block contains information about active variables.
- Then information about analysis components.
- Finally we have an id of the current iteration

3.3 Dakota results file format

Once the simulation code has finished it should return the needed information through a results file. This is the file *results.1*:

```
      8.5 f
```

It returns the function evaluation value. For a general case we will find a line for every requested response, a line for every requested gradient and a line for every requested Hessian.

After a simulation, Dakota expects to read a file containing responses reflecting the current parameters and corresponding to the function requests in the active set vector. The response data must be in proper format. If the amount of data in the file does not match the function request vector, Dakota will abort execution with an error message.

The format of the numeric fields may be floating point or scientific notation. In the latter case, acceptable exponent characters are E or e.

A Dakota Source code modification

Original code from Dakota has an incompatibility with the queue system of Marenostrum. The queue system kills all the asynchronous task that Dakota uses to run concurrent jobs. We have modified Dakota code to avoid this problem.

In the Linux Bash terminal, an orphan process can be created by attaching an ampersand at the end of the command line. This is an abstract of the original file CommandShell.cpp where we can see how Dakota runs asynchronous jobs:

```
CommandShell& CommandShell::flush()
{
  if (asynchFlag)
    sysCommand += " &";
  std::system(sysCommand.c_str());
}
```

As we can see, Dakota makes a system call using the ampersand when the asynchronous flag is true. In this case, the new process is orphan and this can became a problem. If we want to avoid this, we can not use the ampersand.

```
CommandShell& CommandShell::flush()
{
  if (asynchFlag)
  {
    if ( fork() == 0 )
    {
      std::system(sysCommand.c_str());
      exit( 0 );
    }
  }
  else
    std::system(sysCommand.c_str());
}
```

In the new version, we call fork to create a new Dakota process for each new simulation. We can identify the original process because the returning value of fork is different from zero. This process exits the function without doing anything. Meanwhile, the new process (its returning value from fork is equal to zero) makes the system call to execute the task, but without detaching it. The task is never an orphan process. Once the task is finished the process is terminated.