Sandia National Laboratories

Operated for the U.S. Department of Energy by
Sandia Corporation

Albuquerque, New Mexico 87185- 0826

date: October 24, 1997

to: Distribution

from: T. Simmermacher, 9119, MS0557, M. Eldred, 9234, MS0439

subject: REVISED: DAKOTA tutorial for new users (GT-004.1)

keywords: DAKOTA, tutorial, text_book, examples, input files, filter, Dryer Design, Slot Coater

input record None

Introduction

DAKOTA (Design Analysis Kit for OpTimizAtion) is a general purpose iterator toolkit that is under development for the integration of commercial and in-house analysis capabilities with broad classes of systems analysis tools. Written in C++, the DAKOTA toolkit is intended as a flexible, extensible interface between analysis codes and iteration methods. In addition to optimization methods and strategies, the DAKOTA toolkit implements uncertainty quantification with stratified sampling methods, parameter estimation with nonlinear least squares solution methods, and sensitivity analysis with general-purpose parameter study capabilities.

This tutorial is designed to give an experienced GOMA/EXODUS jockey an introduction into tying the DAKOTA iterator toolkit to the GOMA simulation code. In addition to understanding GOMA and the EXODUS file format, the user is assumed to have an understanding of a programing language such as C or FORTRAN. Although the examples will be presented in C, the programs can just as easily be written in FORTRAN.

Standard text_book example

Problem formulation

The problem to be solved in this portion of the tutorial is the text_book example:

$$f = (x_1 - 1)^4 + (x_2 - 1)^4$$

$$g_1 = x_1^2 - \frac{x_2}{2} \leq 0$$

$$g_2 = x_2^2 - 0.5 \leq 0$$

subject to simple bounds on the variables: x_1 and x_2 range between -10 and +10.

dakota_sample.in problem description file

Sections are delimited by newline characters. Therefore, to continue a section onto multiple lines, the back-slash character is needed to escape the newline. Input is order-independent and white-space insensitive. Keywords may be abbreviated so long as the abbreviation is unique. Comments are preceded by #.The definitive resource for input grammar is Dakota/src/dakota.input.spec.

```
# DAKOTA INPUT FILE - dakota_textbook.in

# Interface section specification
# NOTES: Interfaces are 1 of 3 main types: application interfaces are used for interfacing
# with simulation codes, approximation interfaces use inexpensive design space
# approximations in place of expensive simulations, and test interfaces use linked-in test
# functions for algorithm testing purposes (to eliminate system call overhead).
# Application interfaces can be further categorized into system and direct types. The
# system type uses system calls to invoke the simulation, while the direct type uses the
# same constructs as the test interface for linked-in simulation codes. Both application
# interface types use analysis_driver, input_filter, and output_filter specifications. The
# system type additionally uses parameters_file, results_file, analysis_usage, file_tag,
# and file_save specifications. The analysis_driver provides the name of the analysis
# executable, driver script, or linked module; the input_filter and output_filter provide
# pre- and post-processing for the analysis in the procedure of mapping parameters into
# responses (default = NO_FILTER); the parameters_file and results_file are data files
# which Dakota creates and reads, respectively, in the system call case (default = Unix
# temp files); analysis_usage defines nontrivial command syntax (default = standard
# syntax), file_tag controls the unique tagging of data files with function evaluation
# number (default = no tagging), and file_save controls whether or not file cleanup
# operations are performed (default = data files are removed when no longer in use).
# Most settings are optional with meaningful defaults as shown above. Refer to the
# Interface Commands section in the User's Instructions manual for additional
```

```
# information.


interface,\
    application system,\
      input_filter    =       'NO_FILTER'\
      output_filter   =       'NO_FILTER'\
      analysis_driver =       'text_book'\
      parameters_file =       'text_book.in'\
      results_file    =       'text_book.out'\
      analysis_usage  =       'DEFAULT'\
      file_tag\
      file_save

# Variables specification
# NOTES:A variables set can contain design, uncertain, and state variables.Design
# variables are those variables which an optimizer adjusts in order to locate an optimal
# design.  Each of the n design parameters can have an initial point, a lower bound, an
# upper bound, and a descriptive tag. Uncertain variables are those variables which are
# characterized by probability distributions. Each uncertain variable specification can
# contain a distribution type, a mean, a standard deviation, a lower bound, an upper
# bound, a histogram filename and a descriptive tag. State variables are "other" variables
# which are to be mapped through the interface. Each state variable specification can
# have an initial state and a descriptor. State variables provide a convenience mechanism
# for parameterizing additional model inputs, such as mesh density, solver convergence
# tolerance and time step controls,and will be used to enact model adaptivity in future
# strategy developments.

variables,\
    continuous_design = 2\
      cdv_initial_point    0.9    1.1\
      cdv_upper_bounds     5.8    2.9\
      cdv_lower_bounds     0.5   -2.9\
      cdv_descriptor       'x1'   'x2'

# Responses specification
# NOTES: This specification implements a generalized Dakota data set by specifying a
# set of functions and the types of gradients and hessians for these functions.
# Optimization data sets require specification of num_objective_functions,
# num_linear_constraints, and num_nonlinear_constraints. Multiobjective opt. is not yet
# supported, so num_objective_functions must be = 1. Uncertainty quantification data
# sets are specified by num_response_functions. Nonlinear least squares data sets  are
# specified with num_least_squares_terms. Gradient type specification may be
# no_gradients, analytic_gradients, numerical_gradients or mixed_gradients:
#       > no_gradients is invalid with gradient-based opt. methods
#       > no_gradients or analytic_gradients are complete specifications
#       > if numerical_gradients, then:
#           >> method_source = vendor OR dakota
#           >> interval_type = forward OR cental
#           >> fd_step_size = <float>
```

```
#       are additional optional parameters in the specification.
#     > mixed_gradients uses id_numerical & id_analytic lists to specify
#       the gradient types for different function numbers. This capability is not yet
# completely implemented within the Iterators. Hessian type specification may currently
# be no_hessians or analytic_hessians. The only optimizers to currently support
# analytic_hessians are the OPT++ full Newton methods.

responses,\
    num_objective_functions = 1\
    num_linear_constraints = 0\
    num_nonlinear_constraints = 2\
    analytic_gradients\
    no_hessians

# Strategy specification
# NOTES: Contains specifications for hybrid, SAO, and OUU strategies. The
# single_method strategy is a "fall through" strategy, in that in only invokes a single
# iterator. If no strategy specification appears, then single_method is the default.

strategy,\
    single_method

# Method specification
# NOTES: method can currently be dot_frcg, dot_mmfd, dot_bfgs, dot_slp, dot_sqp,
# npsol_sqp, optpp_cg, optpp_q_newton, optpp_g_newton, optpp_newton,
# optpp_fd_newton, optpp_ba_newton, optpp_baq_newton, optpp_bc_newton,
# optpp_bcq_newton, optpp_bc_ellipsoid, optpp_pds, optpp_test_new, sgopt_pga_real,
# sgopt_pga_int, sgopt_coord_ps, sgopt_coord_sps, sgopt_solis_wets, sgopt_strat_mc,
# nond_probability, nond_mean_value, or parameter_study. Most method control
# parameters are optional with meaningful defaults, although sgopt_coord_ps,
# sgopt_coord_sps, parameter_study, nond_probability, and nond_mean_value have
# some required control parameters. Default values for optional parameters are defined in
# the DataMethod class constructor and are documented in the Method Commands
# section of the User's Instructions manual.

method,\
        dot_sqp,\
      max_iterations = 50,\
      convergence_tolerance = 1e-4\
      output verbose\
      optimization_type minimize
```

text_book.C

This simple application program reads the parameters and writes the responses directly; therefore, the NO_FILTER option may be used. The output must be formatted based on the DakotaResponse IO operators.

```
#include <iostream.h>
#include <iomanip.h>
```

```
#include <fstream.h>
#include <math.h>
#include <rw/cstring.h>

#ifdef SYMANTEC
#include <console.h>
#endif


double eval(const double* x, int len);
int main(int argc, char** argv)
{
#ifdef SYMANTEC
  argc = ccommand(&argv);

  for(int num=0; num<argc; num++) {
    cout << argv[num] << " ";
  }
  cout << '\n';
#endif

  ifstream fin(argv[1]);
  ofstream fout(argv[2]);

  // Get the first line and use info for array allocation
  int num_vars, num_fns;
  RWCString vars_text, fns_text;
  fin >> num_vars >> vars_text >> num_fns >> fns_text;

  // Get the parameter vector and ignore the labels
  //vector<double> x(num_vars);
  double* x = new double [num_vars];
  int i;
  for(i=0; i<num_vars; i++) {
    fin >> x[i];
    fin.ignore(256, '\n');
  }

  // Get the ASV vector and ignore the labels
  int* ASV = new int [num_fns];
  for(i=0; i<num_fns; i++) {
    fin >> ASV[i];
    fin.ignore(256, '\n');
  }

 // Compute the results and output them directly to argv[2] (the NO_FILTER
 // option is used).  Response tags are now optional; output them for ease
 // of results readability.
 fout.precision(10);
 fout.setf(ios::scientific);
 fout.setf(ios::right);
 // **** f:
```

```
if (ASV[0]==1 || ASV[0]==3 || ASV[0]==5 || ASV[0]==7)
  fout << "                      " << eval(x, num_vars) << " f\n";

// **** c1:
if (num_fns>1) {
  if (ASV[1]==1 || ASV[1]==3 || ASV[1]==5 || ASV[1]==7)
    fout << "                      " << (x[0]*x[0] - 0.5*x[1]) << " c1\n";
}

// **** c2:
if (num_fns>2) {
  if (ASV[2]==1 || ASV[2]==3 || ASV[2]==5 || ASV[2]==7)
    fout << "                      " << (x[1]*x[1] - 0.5     ) << " c2\n";
}

// **** df/dx:
if (ASV[0]==2 || ASV[0]==3 || ASV[0]==6 || ASV[0]==7) {
  fout << "[ ";
  for (i=0; i<num_vars; i++)
    fout << 4.*pow(x[i]-1,3) << " ";
  fout << "]\n";
}

// **** dc1/dx:
if (num_fns>1) {
  if (ASV[1]==2 || ASV[1]==3 || ASV[1]==6 || ASV[1]==7) {
    fout << "[ " << 2.*x[0] << " " << -0.5;
    for (i=3; i<=num_vars; i++)
      fout << " " << 0.0;
    fout << " ]\n";
  }
}

// **** dc2/dx:
if (num_fns>2) {
  if (ASV[2]==2 || ASV[2]==3 || ASV[2]==6 || ASV[2]==7) {
    fout << "[ " << 0.0 << " " << 2.*x[1];
    for (i=3; i<=num_vars; i++)
      fout << " " << 0.0;
    fout << " ]\n";
  }
}

// **** d^2f/dx^2: (full Newton unconstrained opt.)
if (ASV[0]>=4) {
  fout << "[[ ";
  for (i=0; i<num_vars; i++)
    for (int j=0; j<num_vars; j++)
      if (i==j)
        fout << 12.*pow(x[i]-1,2) << " ";
      else
        fout << 0. << " ";
```

```
    fout << "]]\n";
  }

  // **** d^2c1/dx^2: (ParamStudy testing of multiple Hessian matrices)
  if (num_fns>1) {
    if (ASV[1]>=4) {
      fout << "[[ ";
      for (i=0; i<num_vars; i++)
        for (int j=0; j<num_vars; j++)
          if (i==0 && j==0)
            fout << 2. << " ";
          else
            fout << 0. << " ";
      fout << "]]\n";
    }
  }

  // **** d^2c2/dx^2: (ParamStudy testing of multiple Hessian matrices)
  if (num_fns>2) {
    if (ASV[2]>=4) {
      fout << "[[ ";
      for (i=0; i<num_vars; i++)
        for (int j=0; j<num_vars; j++)
          if (i==1 && j == 1)
            fout << 2. << " ";
          else
            fout << 0. << " ";
      fout << "]]\n";
    }
  }

  fout << flush;
  delete [] x;
  delete [] ASV;

  return 0;
}

//double eval(const vector<double>& x)
double eval(const double* x, int len)
{
  double value=0;

  for(int i=len; i--; ) {
    value += pow(x[i]-1, 4);
  }

  return value;
}
```

Invokation of text_book

The command syntax which DAKOTA will use is as shown below. Parameters and results file names will be passed on the command line to the specified executable and file tagging will be employed to keep the file names unique. The names of the parameters and results files are passed on the command line for the convenience of the application developer, since these arguments can be used to remove hard coding of file names and improve generality:

```
text_book text_book.in.1 text_book.out.1
```

The text_book.in.1 parameters file is:

```
2 variables 3 functions
9.0000000000e-01 x1
1.1000000000e+00 x2
                 1 ASV_1
                 1 ASV_2
                 1 ASV_3
```

and the text_book.out.1 results files is:

```
2.0000000000e-04 f
2.6000000000e-01 c1
7.1000000000e-01 c2
```

Results

dot_sqp converges to the optimal solution in 17 total function evaluations when foward finite differences are used

```
<<<<< Single method iteration completed.
<<<<< Function evaluation summary: 17 total (16 new, 1 duplicate)
<<<<< Best design parameters  =
                     5.9442052455e-01 x1
                     7.0668310706e-01 x2
<<<<< Best objective function =
                     3.4460496673e-02
<<<<< Best constraint values  =
                    -5.7935237028e-06
                    -5.9898619602e-04
```

text_book recast in GOMA format: Filter Introduction

There are several ways of interfacing DAKOTA with a simulation code. The method used here uses DAKOTA's 1-piece Interface capability. In this method, DAKOTA makes one system call per function evaluation, all the control of the evaluation is given to the user. DAKOTA also has the capability to use a 3-piece Interface capability which performs separate system calls for the input filter, simulation code, and the output filter in that order to evaluation the cost function and

constraints. In the optimizations described here, the evaluation of the cost function is performed by a combination of C programs and a supervisory UNIX shell program using DAKOTA's 1-piece Interface capability.

Figure 1 outlines how the various files are passed and how the various codes interact. An input specification to DAKOTA (e.g., **`dakota_sample.in`**) sets up the optimization parameters such as the communication files (e.g., **`params.in`** and **`results.out`**), the number of design variables, their bounds and starting values, information concerning the number of constraints and how gradients are calculated, and the optimization technique desired.

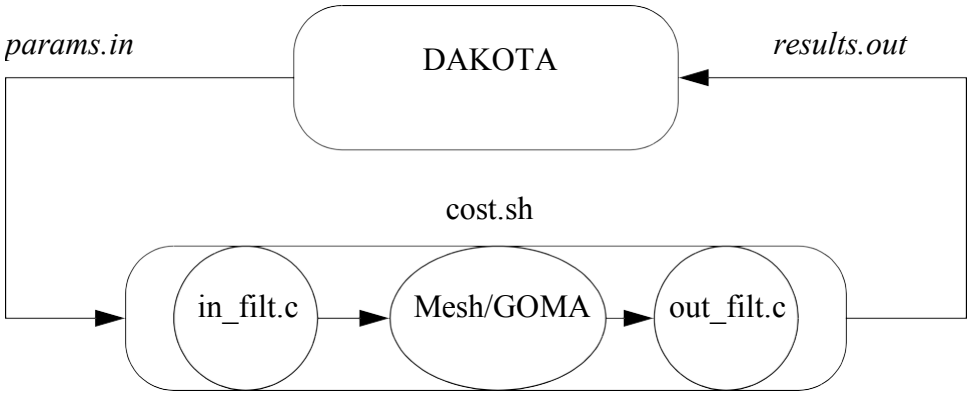

Figure 1. DAKOTA Interface Scheme

When performing a function evaluation, DAKOTA writes the file **`params.in`**. This file contains the current values of the design variables and an "active set vector" code request for the value of the function and constraints, their gradients, and/or their Hessian matrix. DAKOTA then spawns a system call and this file is read by the user's shell program **`cost.sh`**. The shell program is fairly simple in that all it does is execute **`in_filt.c`**, GOMA, and **`out_filt.c`** in the appropriate order.

The input filter, **`in_filt.c`**, places the design variables identified in file **`params.in`** into a file that is formatted for use by APREPRO. This file is "included" into the mesh generator file or into the GOMA input deck. GOMA is then run and an EXODUS file is generated. The output filter, **`out_filt.c`**, then reads the EXODUS file, extracts the necessary results, computes the cost function and the value of the constraints, then writes the file **`results.out`** into the format necessary for DAKOTA.

The files **`in_filt.c`** and **`out_filt.c`** are written in a general format. The input filter can be run without any modification in most optimizations. A skeleton output filter is provided that only requires the subroutines to evaluate the cost functions and the constraints. The code required to write the file **`results.out`** file is provided.

DAKOTA Filter Tutorial

The text_book example will be revisited in this portion of the tutorial, but will be recast in the form used by the GOMA applications. The problem formulation, as before, is:

$$f = (x_1 - 1)^4 + (x_2 - 1)^4$$

$$g_1 = x_1^2 - \frac{x_2}{2} \leq 0$$

$$g_2 = x_2^2 - 0.5 \leq 0$$

subject to simple bounds on the variables: x_1 and x_2 range between -10 and +10.

1. Change directories into the "tutorial" directory (this location will depend on the course you are taking and how you installed your software).

2. You will notice the directory contains five files: **in_filt.c**, **out_filt.c**, **dak_goma.h**, **cost.sh**, and **tut.in**. The file **tut.in** is the DAKOTA input specification as discussed earlier. Issue the following commands:

   ```
   more tut.in
   ```

   The first part of the files lets DAKOTA know how the interface is set up. The slash at the end of the line signifies a continuation. It must be present immediately before any carriage return prior to the end of a keyword specification (the parser detects keyword input completion with a newline, so newlines entered for readability must be escaped with a '\'). Note that the communication files are set up to be **params.in** and **results.out**. The cost function is evaluated in the file **cost.sh**. **cost.sh** will be called by DAKOTA as

   ```
   cost.sh params.in results.out
   ```

   The shell function **cost.sh** must therefore be coded with this in mind (as we will see next).

   ```
   interface,\
       application system,\
         input_filter =          'NO_FILTER'    \
         output_filter =         'NO_FILTER'    \
         analysis_driver=        'cost.sh'\
         parameters_file=        'params.in'\
         results_file=           'results.out'\
         analysis_usage =        'DEFAULT'
   ```

   Next, the design variables are set up. Note that there are two design variables, x_1 and x_2, and the starting point is (2, 2). Each variable may range between -10 and +10.

   ```
   variables,\
       continuous_design = 2\
         cdv_initial_point        2.0      2.0\
   ```

```
    cdv_upper_bounds     10.0     10.0\
    cdv_lower_bounds    -10.0    -10.0\
    cdv_descriptor       'x1'     'x2'
```

The response specification describes the number of constraints and the source of the gradients. In this problem and in the problems utilizing GOMA, the gradients are calculated using a forward difference scheme:

```
responses,\
    num_objective_functions = 1\
    num_linear_constraints = 0\
    num_nonlinear_constraints = 2\
    numerical_gradients\
         method_source  vendor           \
         interval_type forward           \
         fd_step_size = 0.001            \
    no_hessians
```

The last portion selects the optimization technique to be used.

```
method,                                          \
    dot_sqp,                                    \
         max_iterations = 50,                   \
         convergence_tolerance = 1e-8           \
         output verbose                         \
         optimization_type minimize
```

3. Now execute the command:

```
more cost.sh
```

This file is the supervisory file that controls the cost function evaluation. This simple example has no GOMA evaluation.

```
 #! /bin/csh -f
#
#  This shell file evaluates the cost function
#    for a dakota run
#
in_filt $argv[1] out.app

## GOMA run goes here!!

out_filt $argv[1] $argv[2]
```

The input filter, **in_filt.c**, places the design variables identified in file **params.in** into a file, **out.app**. The file **out_filt.c** will take the file **out.app** and evaluate the cost function then write the file **results.out**. The first line of the file **cost.sh** is necessary for the shell to execute correctly. The variables **$argv[1]** and **$argv[2]** refer to the first argument and the second argument in the call statement.

4. Now look at the input filter:

```
more in_filt.c
```

The first portion of the file sets up various definitions and prototypes the functions used in the program:

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

void input_filter(FILE *input_file, FILE *param_file);
```

The program is controlled from **main()**. This routine performs error checking on the number of arguments used to call the program, opens **params.in** for reading and **out.app** for writing and calls the subroutine to perform the filtering operation, **input_filt()**.

```
void main(int argc,char *argv[])
{

FILE *input_file, *param_file;

if (argc<2)
{
  printf("Need an output filename, exiting\n");
  exit(-1);
}

if ((input_file=fopen(argv[1],"r")) == NULL)
{
  printf("Couldn't open file: %s   exiting.\n",argv[1]);
  exit(-1);
}

param_file=fopen(argv[2],"w");

input_filter(input_file, param_file);
exit(0);
}
```

The first line of **params.in** specifies the number of design variables (**n_param**) and a string (**tag**). The next **n_param** lines are the value of each of the design variables with an identification tag. The last lines are the ASV for the function and each of the constraints. The ASV can be ignored in this input filter since only function values will be returned. The initial **params.in** file for this problem is listed below:

```
2 variables 3 functions
2.0000000000e+00 x1
2.0000000000e+00 x2
                 1 ASV_1
                 1 ASV_2
                 1 ASV_3
```

The last portion of the file **in_filt.c** is the input filter subroutine. It just reads **params.in** and writes **out.app** using the format in the above paragraph.

```
void input_filter(FILE *input_file, FILE *param_file)
{
  int i, n_param,n_g;
  float dum_param;
  char tag1[10],tag2[10];

  fscanf(input_file,"%d %s %d %s",&n_param,tag1,&n_g,tag2);

  for (i=0;i<n_param;i++)
  {
    fscanf(input_file,"%f %s",&dum_param,tag1);
    fprintf(param_file,"#{%s = %f}\n",tag1,dum_param);
  }
}
```

The contents of the file **out.app** will look something like:

```
#{x1 = 2.000000}
#{x2 = 2.000000}
```

You will recognize this as input for APREPRO.

5. The final file is the output filter (**out_filt.c**). Normally, it reads an EXODUS file to get the results of a GOMA run. In this case, the file **out.app** will represent the EXODUS file to simplify the description of the filters. As with **in_filt.c**, the first lines set up definitions and prototypes for the remaining code.

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

#define NUM_PARAM 2
#define MAX_LINE 80

float cost_fun(FILE *exoid);

float *asv_read(FILE *input_file,int *n_param, int *n_g, int **asv);

void output_filter(int *asv, int n_param, float *params, int n_g, FILE
*output_file);
```

The **main()** routine in **out_filt.c** once again controls the filter process as with **in_filt.c**. First some error checking is performed to insure the correct number of arguments are being passed. Next the files **params.in** and results.out are opened. The remaining functions are the meat of the program and will be discussed next.

```
int main(int argc,char *argv[])
{

int *asv, n_param,i, n_g;
```

```
float *params;
FILE *input_file, *output_file;

if (argc<3)
{
    printf("Need both input and output files specified, exiting \n");
    exit(-1);
}

input_file=fopen(argv[1],"r");

output_file=fopen(argv[2],"w");

params=asv_read(input_file,&n_param,&n_g,&asv);

output_filter(asv, n_param, params, n_g, output_file);

exit(0);
}
```

The subroutine **asv_read()** reads the **params.in** file returning the ASV information and the values of the parameters. This allows the program to correctly determine what DAKOTA is requesting and to allow the parameters to be available for the cost function and the constraint evaluation. The array **asv[] and the array params[] are alloced in asv_read()**. This is done here with the **calloc()** statement.

```
float *asv_read(FILE *input_file, int *n_param, int n_g, int **asv)
{
  int i;
  char junk1[MAX_LINE],junk2[MAX_LINE];
  float *params;

  fscanf(input_file,"%d %s",n_param,junk1,n_g,junk2);
  *n_g = *n_g - 1;

  params= (float *)calloc(*n_param, sizeof(float));
  *avs=(int *)calloc(*n_g +1, sizeof(int));

  for (i=0;i<*n_param;i++) fscanf(input_file,"%f %s\n",&params[i], junk);

  for (i=0;i<=*n_g;i++)
    {
    fscanf(input_file,"%d %s\n",&(*asv)[i],junk1);
    }
  fclose(input_file);
  return(params);
}
```

The next subroutine is the actual output filter (**out_filt.c**). This subroutine opens the EXODUS file (in this case **out.app**) and evaluates the constraints, **g[n_g]** and the cost function, **J_cost.** In this example the cost function is evaluated using the routine **cost()** and the constraints are just combinations of the parameters. The remaining code preforms error checks

on the ASV to be sure that the DAKOTA input specification is correct as far as the gradients and Hessians that can be provided by the output filter. It also writes out the **`results.out`** file with the appropriate information.

```
void output_filter(int *asv, int n_param, float *params, int n_g, FILE
*output_file)
{
  int i;
  float J_cost;
  float *g;
  FILE *exoid;

  g=(float *)calloc(n_g ,sizeof(float));

  exoid=fopen("out.app","r");
/* determine cost function and constraints*/

  g[0] =  params[0]*params[0] - params[1]/2.;
  g[1] = params[1]*params[1] -0.5;

  J_cost =  cost_fun(exoid);

  /* write dakota output file */

  if (asv[0]>3)
  {
      printf("Hessian is not available, exiting\n");
      exit(-1);
  }

  if (asv[0]>2)
  {
    printf("Gradient is not available, exiting\n");
    exit(-1);
  }

  if (asv[0]==1 || asv[0]==3 ||asv[0]==5)
   {
     fprintf(output_file,"%g f\n",J_cost);
   }

  for (i=1;i<=n_g;i++)
  {
    if (asv[i]==1 || asv[i]==3 || asv[i]==5)
      fprintf(output_file,"%g c%d\n",g[i-1], i);
    else
    {
      printf("Number of parameters is probably wrong:      exiting.\n");
      exit(-1);
    }
  }
```

```
  free(g);
}
```

The final routine evaluates the cost function. In this case, this routine is exceptionally simple. It just reads the file **out.app** and runs the parameters through a formula:

```
float cost_fun(FILE *exoid)
{
int i;
float x[NUM_PARAM], J_cost, a, b;
char cdum[2];

for (i=0;i<NUM_PARAM;i++)
  {
   fscanf(exoid,"#{%s = %f}\n",cdum,&x[i]);
   printf(" x[%d] = %g \n",i,x[i]);
  }

a=(x[0]-1.);
b=(x[1]-1.);

J_cost = a*a*a*a + b*b*b*b;

return J_cost;
}
```

6. To compile a program with EXODUS subroutines in it, excute a command similar to:

```
cc -o in_filt in_filt.c -lexoIIv2c -lnetcdf -lnsl -lm
```

7. Now the optimization can be run. To execute the optimization, issue the command:

```
dakota -i tut.in
```

Wait until the thing finishes and enjoy the results.

Dryer Design Example

This section presents an extension of the tutorial problem to an example problem that you care about. The shell program changes trivially, the input filter doesn't change at all, and only the cost function evaluation changes in the output filter. The problem that is being solved is the multilayer drying problem shown in Figure 2. The one dimensional problem has two solvents and a substrate. The cost function is the concentration of solvent 0 at the end of the simulation, which for this case is 200 sec. The design variable is the oven temperature, which has a constraint of 370K to prevent boiling.

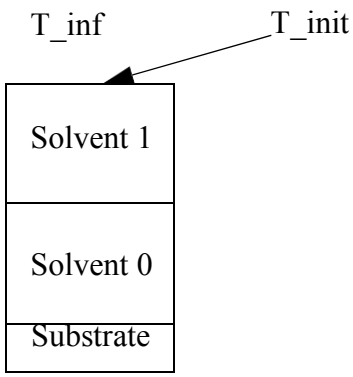

Figure 2. Drying Problem Setup

Dryer Design Tutorial

1. There are a few differences in the input specification to DAKOTA. The specification is in **dryer.in**. The first is the change in the name of the analysis driver:

```
analysis_driver=        'dryer.sh'
```

   The variable description also changes:

```
variables,                                          \
        continuous_design = 1                                  \
          cdv_initial_point       300.0 \
          cdv_upper_bounds     600.0  \
          cdv_lower_bounds       0.   \
          cdv_descriptor    'T_inf'
```

   The final change is in the responses section. Here the number of constraints must be changed to reflect the current problem:

```
 num_nonlinear_constraints = 1           \
```

2. The shell function **dryer.sh** is identical to **cost.sh** described above except for the GOMA evaluation. To look at the file execute:

```
more dryer.sh
```

   The file looks like

```
#! /bin/csh -f
#
#  This shell file evaluates the cost function
#    for a dakota run
#
in_filt $argv[1] dryer.app

goma -a -i ml_input -se stderr -so stdout

dryer $argv[1] $argv[2]>>& goma.src
```

3. The input filter is identical. If you don't believe me execute:

   ```
   more in_filt.c
   ```

   Note that the output of the input filter is **dryer.app** as can be seen from the file **dryer.sh**, which contains the oven temperature. This file is read by the GOMA input file **Defs.app**. Check this file now to see the include statement at the top of the file.

4. The simulation is identical to the template **dryer.ml** provided to you in your distribution. If you are not familiar with it, become familiar with it.

5. The major changes are in the output filter, **dryer.c** for this problem. Only the cost function evaluation will be discussed. The remaining code is as it was described above. Any variables in all capital letters are defined in the header.

   The cost function requires interrogating the EXODUS file that is generated by GOMA for the concentration of the solvent Y0 at a node near the substrate (node 8). Open the file **dryer.c** by executing:

   ```
   emacs dryer.c
   ```

   Move the cursor down to the portion of the code where the function **output_filter()** is defined. After the definition of the necessary variables, the first line of code opens the EXODUS file using an EXODUS provided subroutine:

   ```
   /* page 25 of SAND92-2137 */
   /* Open file */

     CPU_word_size=sizeof(float);
     IO_word_size=0;

     exoid=ex_open(filename,EX_READ,&CPU_word_size,&IO_word_size,&version);
   ```

   Note that the comments give the reference page in the EXODUS users manual which is available on the Sandia Internal Web and externally also as a PDF file from the CUBIT Web site at *http://endo.sandia.gov/cubit/help-beta/Appendix/Exodus_II_File_Specification.html*

   The **exoid** output is used to reference the file. It is of type **int**. Now the cost function **cost_fun()** is called with the argument being **exoid**. After some variable definitions, the number of variables are determined from the database and the array for the variable names is set up:

   ```
   /* page 133 of SAND92-2137 */

     error=ex_get_var_param(exoid,"n",&num_nodal_var);

     for (i=0;i<num_nodal_var;i++)
       var_names[i] = (char *) calloc((MAX_STR_LNG+1),sizeof(char));
   ```

   Next, the variable names are extracted and the index of the appropriate variable, Y0, is determined. The variables are referenced in the database by their index and this will be needed when extracting the concentration time history.

   ```
   /* page 137 of SAND92-2137 */
   ```

```
  error=ex_get_var_names(exoid,"n",num_nodal_var,var_names);

/* find the velocity variables */

  for (i=0;i<num_nodal_var;i++)
  {
    if (strcmp(CONC,var_names[i])==0)  CONC_var_index=i+1;
  }
```

Next, the number of time steps are determined and the arrays to hold all the time step values and the values of the concentration history at node 8 are allocated. The array **times** will contain the time axis.

```
/* page 41 of SAND92-2137 */
/* determine number of time steps and use the last one */
  error=ex_inquire(exoid,EX_INQ_TIME,&num_time_steps,&fdum,cdum);

concentration = (float *) calloc(num_time_steps,sizeof(float));
times = (float *) calloc(num_time_steps,sizeof(float));

/* page 143 of SAND92-2137 */
error = ex_get_all_times(exoid,times);
```

Now the concentration history at node 8 is read and the last value of the concentration is used for the cost function

```
/* page 167 of SAND92-2137*/

error = ex_get_nodal_var_time(exoid, CONC_var_index,NODE,1,
          num_time_steps,concentration);
printf(" %g   %g \n",concentration[0],concentration[num_time_steps-1]);
J_cost=concentration[num_time_steps-1];
```

6. To compile a program with EXODUS subroutines in it, execute a command similar to:

```
cc -o in_filt in_filt.c -lexoIIv2c -lnetcdf -lnsl -lm
```

7. To run the simulation, just type:

```
dakota -i dryer.in
```

8. Sit back and watch it run.

Dryer Parameter Study in Fortran

This section will go through an example of a FORTRAN interface between DAKOTA and GOMA. The example will be a multi-dimensional parameter study using the same cost function as the previous example, namely the concentration of the solvent at the substrate at the end after 200 sec. The two variables that will vary are the oven temperature, T_inf, and the convection coefficient, Kh.

FORTRAN Tutorial

1. The input specification, **dryer.in**, for DAKOTA is as follows:

```
interface,\
    application system,\
      input_filter =           'NO_FILTER'    \
      output_filter =          'NO_FILTER'    \
      analysis_driver=         'dryer.sh'\
      parameters_file=         'params.in'\
      results_file=            'results.out'\
      analysis_usage =         'DEFAULT'


variables,\
    continuous_design = 2\
      cdv_initial_point        300.0   -50 \
      cdv_upper_bounds         600.0   -50\
      cdv_lower_bounds         0.      0\
      cdv_descriptor    'T_inf'       'Kh'


responses,\
    num_objective_functions = 1\
    num_linear_constraints = 0\
    num_nonlinear_constraints = 1\
    no_gradients\
    no_hessians

strategy,\
    single_method


method,                                              \
        multidim_parameter_study\
    partitions = 10 10
```

The main difference between this file and the ones discussed in the previous examples is the "variables" section and the "method" section.

2. The next file necessary is the shell file **dryer.sh** which runs the simulation and controls the cost function evaluation. It is pretty simple and has been discussed in both the previous examples:

```
#! /bin/csh -f
#
#  This shell file evaluates the cost function
#    for a dakota run
#
in_filt

goma -a -i ml_input -se stderr -so stdout

dryer
```

3. The input filter in FORTRAN is a little less general than the one written in C. It is not easy to pass command line arguments in FORTRAN and therefore the files read and written by the input filter have to be hard coded. It is imperative that the files coded to be read in the input filter are identical to those used in the dakota input specification. This means that file_tag and file_save cannot be used, nor can the default file names for the parameter file or the results file.

```
      program in_filt
c
c  This is a poor man's version of
c  the c program in_filt.c
c  LEARN C!!!
c

      integer i, nparam, nfun
      real dparam
      character*10 tag, junk

      open(22,file='params.in',status='old')
      open(33,file='dryer.app',status='unknown')

      read(22,*) nparam, tag, nfun, junk

      do 10 i=1, nparam
         read(22,*) dparam,tag
         write(33,'(1x,a2,a10,a1,f16.8,a1)') "#{",tag,"=",dparam,"}"
 10   continue

      end
```

4. The function that evaluates the cost function, **dryer.f**, is now described. As with the input filter the filenames have to be hard coded, limiting the generality of the code. The main program does little except call the appropriate subroutines in the appropriate order

```
      program dryer
      include '/usr/local/inc/exodusII.inc'
      character*12 infile, outfile
      integer asv(3), nparam, ng
      real params(2)

      infile = 'params.in'
      outfile = 'results.out'

      call asvrd(infile, nparam, ng, asv, params)

      call outfilt(asv, nparam, params, ng, outfile)

      stop
      end
```

The first subroutine, **asvrd()**, reads the parameters file, params.in, and determines the values of the parameters and the ASV

```
      subroutine asvrd(infile, nparam, ng, asv, params)
      character*12 infile
      character*50 junk, junk1
      integer i, nparam, ng, asv(3)
      real params(2)

      open(unit=22, file=infile, status='old')

      read(22,*) nparam, junk, ng, junk1
      ng=ng-1

      do 10 i=1,nparam
         read(22,*) params(i), junk
10    continue

      do 20 i=1,ng+1
         read(22,*) asv(i), junk
         write(*,*) asv(i), junk
20    continue

      close(22)
      end
```

The next subroutine, **`outfilt()`**, opens the EXODUS database and controls the writing of the file, **`results.out`** for DAKOTA to read. It does a lot of checks to make sure that the function values and their gradients that DAKOTA asks for through the ASV are, in fact, available

```
      subroutine outfilt(asv, nparam, params, ng, outfile)
      include '/usr/local/inc/exodusII.inc'
      integer asv(3), i, nparam, ng
      real params(2)
      character*12 outfile
      real J_cost, g(2)

      integer cpu_ws, exopen, exread, io_ws, idexo, ierr
      real vers

      cpu_ws=0
      io_ws=0

c page 25 of SAND92-2137

      idexo = EXOPEN ("out.exoII", EXREAD, cpu_ws, io_ws, vers, ierr)

      g(1) = params(1) - 370.0

      J_cost = costf(idexo)

      open(unit=33, file=outfile, status='unknown')

      if (asv(1) .gt. 3) then
         write(*,*) 'Hessian is not available, exiting '
```

```
         call exit(0)
      endif

      if (asv(1) .gt. 2) then
         write(*,*) 'Gradient is not available, exiting '
         call exit(0)
      endif

      if (asv(1) .eq. 1 .or. asv(1) .eq. 3 .or. asv(1) .eq. 5 ) then
         write(33,*) J_cost, ' f'
      endif

      do 30 i=1,ng

         if (asv(i) .gt. 3) then
            write(*,*) 'Hessian is not available, exiting '
            call exit(0)
         endif

         if (asv(i) .gt. 2) then
            write(*,*) 'Gradient is not available, exiting '
            call exit(0)
         endif

         if (asv(i) .eq. 1 .or. asv(i) .eq. 3 .or. asv(i) .eq. 5 ) then
            write(33,*) g(i), ' g1'
            write(*,*) g(i)
         endif

30    continue

      end
```

The last function, **costf()**, calculates determines what the value of the solvent at the substrate is at the end of the simulation (200 sec). It uses a lot of calls from the EXODUS subroutine library and page numbers in the EXODUS reference guide are give to facilitate reading the code. The variable exoid is used to reference the EXODUS database file that the GOMA results will be read from.

```
      real function costf(idexo)
      include '/usr/local/inc/exodusII.inc'
      integer cvarind, extims, i, idexo, ntime, nvar, ierr

      real redum, time
      real time(500), concen(500)
      character*(MXSTLN) vname(20)
      character cdum
```

First, we need to know how many variables are in the database

```
c page 133 SAND92-2137
```

```
      call EXGVP(idexo, "n", nvar, ierr)

c page 137 SAND92-2137
```

Next, we read the variable's names in and determine which one is the one of interest. In this case we are interested in "Y0".

```
      call EXGVAN(idexo, "n", nvar, vname, ierr)

      do 40 i=1,nvar
         if (vname(i) .eq. "Y0") then
            cvarind = i
         endif
 40   continue
```

Now we find out how many time steps are in the database

```
c page 41 of SAND92-2137

      call EXINQ(idexo, EXTIMS, ntime, redum, cdum, ierr)
c page 144 of SAND92-2137

      call EXGATM(idexo, time, ierr)
```

Finally we read in all the values of Y0 through time and take the last one, then close the file

```
c page 167 of SAND92-2137

      call EXGNVT(idexo, cvarind, 8, 1, ntime, concen, ierr)

      costf=concen(ntime)

c page 27 of SAND92-2137

      call EXCLOS(idexo,ierr)
      return
      end
```

5. To compile a FORTRAN program with EXODUS commands in it, execute a command similar to:

```
f77 -o dryer dryer.f -lexoIIv2for -lexoIIv2c -lnetcdf -lnsl
```

6. To run the simulation, just type

```
dakota -i dryer.in
```

7. Sit back and watch it run.

Slot Coater Example

The slot coater example utilizes the failure capture option in DAKOTA. There are two ways to insure a solution throughout the optimization: The first is to set the relaxation schedule very conservatively and the other is to rely on continuation. By relying on continuation, the optimization runs significantly faster. In this optimization, there was no relaxation used.

The parameterization used for this example is shown in Figure 3. Only the gap and the angle a were used in the optimization. The parameters used for the starting point of the optimization were taken from

Sartor (1990), *Slot Coating*, PhD. Thesis University of Minnesota, Available on University Microfilms, Ann Arbor, Michigan.

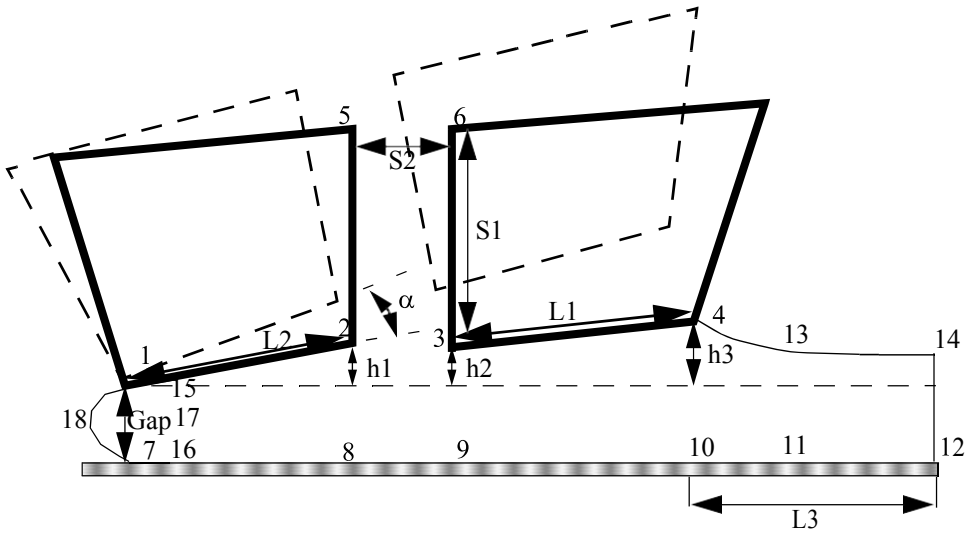

Figure 3. Slot Coat Parameterization

The cost function used for this optimization is

$$J = \left(\frac{28.5}{0.02}\right)\frac{\partial}{\partial u_{web}}(x_{dcl}) + \left(\frac{2850}{0.01}\right)\frac{\partial}{\partial P_{vac}}(x_{dcl})$$

$$0.35 < \text{Gap} < 0.7$$

$$-0.2 < \alpha < 0.2$$

The design variable, as it is currently set up, is the gap length and the angle α. The cost function was choosen to minimize the sensitivity of the movement of the dynamic contact line to changes in the webspeed or the back pressure.

Slot Coater Tutorial

1. As with all the examples before, the first file necessary is the DAKOTA input specification. This example is identical to all the optimization problems so far except for the specification of the design variables and the addition of the failure capture command in the interface specification. The file is called **slot.in**. The changed portions of the specification for this problem is

```
interface,                                              \
        application system,                                     \
          input_filter =          'NO_FILTER'           \
          output_filter =         'NO_FILTER'           \
          analysis_driver=        'slot.sh'             \
          parameters_file=        'params.in'    \
          results_file=           'results.out' \
          analysis_usage =        'DEFAULT'      \
        failure_capture continuation

######
variables,                                      \
        continuous_design = 2                           \
          cdv_initial_point           0.05  0.0\
          cdv_upper_bounds           0.07     0.2\
          cdv_lower_bounds          0.035 -0.2\
          cdv_descriptor      'Gap_new'   'alpha_new'
```

2. The C-shell file, **slot.sh**, should look pretty familiar also

```
#! /bin/csh -f
#
#  This shell file evaluates the cost function
#    for a dakota run
#
in_filt $argv[1] slot.app
```

```
goma -a -i slot_input -se stderr -so stdout

slot $argv[1] $argv[2]
```

3. The input filter **in_filt.c** is identical to all the previous input filters.

4. The major difference is in the cost function, **slot.c**. The subroutines **main()** and **asv_read()** are the same. However, the routine **output_filter()** has been changed to incorporate a failure capturing scheme. There has been add to GOMA four global variables that indicate the convergence status of the GOMA simulation. They are:
   - CONVBoolean convergence (1=> converged, 0=> not converged)
   - NEWT_ITNumber of Newton Iterations specified in the GOMA input file
   - MAX_ITNumber of Newton Iterations taken by the simulation
   - CONVRATEThe log10 relative convergence rate at the second to last and the last iteration taken

   The subroutine converge in **slot.c** takes care of reading these values. The subroutine **out_filt** look like

```
void output_filter(int *asv, int n_param, double *params, int n_g,
       FILE *output_file)
{
  char filename[]=GOMA_FILE;
  int CPU_word_size, IO_word_size;
  float version;
  int exoid, i;
  double J_cost;
  double *g;
  int newt_it, max_it,error;
  double convrate;

  g=(double *)calloc(n_g ,sizeof(double));

/* page 25 of SAND92-2137 */
/* Open file */

  CPU_word_size=sizeof(double);
  IO_word_size=0;
```

   This section of the code is the most different. Note how the EXODUS file is opened, then the convergence is checked. If the simulation didn't converge, a failure is flagged and the program exits. If the simulation didn't converge but it ran out of newton iterations, then the program exits and a "1" is returned so the shell program can rerun GOMA (not yet implemented). If it has converged, then it writes the **results.out** file as before.

```
exoid=ex_open(filename,EX_READ,&CPU_word_size,&IO_word_size,&version);

error = converge(exoid, &max_it, &newt_it, &convrate);

if (!error) {
  /* determine cost function and constraints*/
```

```
    system("cp soln.dat contin.dat");

    g[0] = - 0.5e-4;
    J_cost =  cost_fun(exoid);

    J_cost=J_cost*J_cost;

    printf("J= %g\n",J_cost);

    /* write dakota output file */

    if (asv[0]>3) {
      printf("Hessian is not available, exiting\n");
      exit(-1);
    }

    if (asv[0]>2) {
      printf("Gradient is not available, exiting\n");
      exit(-1);
    }

    if (asv[0]==1 || asv[0]==3 ||asv[0]==5)  fprintf(output_file,
       "%g f\n",J_cost);

    for (i=1;i<=n_g;i++) {
      if (asv[i]==1 || asv[i]==3 || asv[i]==5) {
    fprintf(output_file,"%g c%d\n",g[i-1],i);
      }
      else {
    printf("Number of parameters is probably wrong:      exiting.\n");
    exit(1);
      }
    }
    return;
  }
  if (newt_it == max_it && convrate > 0.0) {
    printf("Not converged!! \n");
    exit(1);
  }
  else {
    fprintf(output_file,"FAIL\n");
  }

  free(g);
}
```

5. The **converge**() routine is fairly basic. It reads the global variables from the EXODUS database, then sends them back.

```
int converge(int exoid, int *max_it, int *newt_it, double *convrate)
{
  int i, inewt, iconv, imax, irate;
```

```
  int ret_int, ntime, nvar, conv;
  int error;
  char *cdum=0, *gvar_name[NUM_G_VAR];
  float fdum;
  double gvar[NUM_G_VAR];

  error=ex_inquire(exoid, EX_INQ_TIME, &ntime, &fdum, cdum);

  error=ex_get_var_param(exoid, "g", &nvar);

  for (i=0; i<nvar;i++) gvar_name[i]= (char *) calloc((MAX_LINE+1),
         sizeof(char));

  error=ex_get_var_names(exoid, "g",nvar, gvar_name);

  for (i=0;i<nvar;i++) {
    if (strcmp(CON_VAR,gvar_name[i])==0) iconv=i;
    if (strcmp(NEWT_VAR,gvar_name[i])==0) inewt=i;
    if (strcmp(MAX_VAR,gvar_name[i])==0)  imax=i;
    if (strcmp(RATE_VAR,gvar_name[i])==0) irate=i;

  }

/* Page 159 SAND92-2137 */
  error=ex_get_glob_vars(exoid, ntime, nvar, gvar);

  if (error == 0) {
    *newt_it=(int) gvar[inewt];
    *max_it=(int) gvar[imax];
    *convrate= gvar[irate];
    conv=(int) gvar[iconv];
  }
  else {
    *newt_it= -1;
    *max_it= -1;
    *convrate= -999999.0;
    conv=0;
  }
  return !conv;
}
```

The cost function evaluation subroutine, **cost_fun()**, is more complicated. Actually it isn't that difficult, it just looks that way. Basically there are two files, **webspeed.app** and **vacuum.app** which are read by **cost_fun()**. First, the nominal position of the dynamic contact point is read. **cost_fun()** then perturbs the values in **webspeed.app** and calls GOMA, then reads the perturbed value of the dynamic contact point. This is repeated for the back pressure. The perturbed values are then used for a finite difference calculation.

```
double cost_fun(int exoid_nom)
{
  int i, CPU_word_size, IO_word_size;
```

```
  int error, exoid_delta ,idum;
  int ns_num_nodes, *ns_node_list;
  double fdum, J1, J2;
  float version;
  double webspeed_nom,webspeed_delt;
  double Pvacuum_nom,Pvacuum_delt, g1,g2;
  double *ns_X,*ns_Y,*ns_Z,*ns_displx_nom, *ns_displx_delt;
  char filename[]=GOMA_FILE,cdum[9];
  FILE *in_file;

  error=ex_get_node_set_param(exoid_nom, NSET, &ns_num_nodes,&idum);
  ns_node_list=(int *) calloc(ns_num_nodes,sizeof(int));

  error=ex_get_node_set(exoid_nom,NSET, ns_node_list);

  ns_X=(double *) calloc(ns_num_nodes,sizeof(double));
  ns_Y=(double *) calloc(ns_num_nodes,sizeof(double));
  ns_Z=(double *) calloc(ns_num_nodes,sizeof(double));
  ns_displx_nom=(double *) calloc(ns_num_nodes,sizeof(double));
  ns_displx_delt=(double *) calloc(ns_num_nodes,sizeof(double));

  get_displ(exoid_nom,NSET,ns_num_nodes,ns_node_list, ns_X, ns_Y, ns_Z,
       ns_displx_nom);

  /******************/
  in_file=fopen(WEBFILE,"r");
  fscanf(in_file,"${%s = %lf}",cdum,&webspeed_nom);

  fclose(in_file);

  webspeed_delt=(1.0+FDEPS)*webspeed_nom;
  in_file=fopen(WEBFILE,"w");
  fprintf(in_file,"${webspeed = %f}\n",webspeed_delt);
  fclose(in_file);

  system("/home/prschun/.sun5/bin/goma -a -i slot_input -se stderr -so
stdout");
  /*system("/home/prschun/.sun5/bin/goma -a -i slot_input");*/
  in_file=fopen(WEBFILE,"w");
  fprintf(in_file,"${webspeed = %f}\n",webspeed_nom);
  fclose(in_file);

  CPU_word_size=sizeof(double);
  IO_word_size=0;


exoid_delta=ex_open(filename,EX_READ,&CPU_word_size,&IO_word_size,&versi
on);

  get_displ(exoid_delta,NSET,ns_num_nodes,ns_node_list, ns_X, ns_Y, ns_Z,
       ns_displx_delt);
  /***************/
```

```
  g1= webspeed_nom/Ls;g1=1.0e3;
  J1= (ns_displx_delt[0] - ns_displx_nom[0])/(webspeed_delt-
webspeed_nom);
  /*****************/
  in_file=fopen(PRESSFILE,"r");
  fscanf(in_file,"${%s = %lf}",cdum,&Pvacuum_nom);
  fclose(in_file);
  Pvacuum_delt=(1.0+FDEPS)*Pvacuum_nom;
  in_file=fopen(PRESSFILE,"w");
  fprintf(in_file,"${vacuum = %f}\n",Pvacuum_delt);

  fclose(in_file);
  system("/home/prschun/.sun5/bin/goma -a -i slot_input -se stderr -so
stdout");
  /*system("/home/prschun/.sun5/bin/goma -a -i slot_input");*/
  in_file=fopen(PRESSFILE,"w");
  fprintf(in_file,"${vacuum = %f}\n",Pvacuum_nom);
  fclose(in_file);

  CPU_word_size=sizeof(double);
  IO_word_size=0;


exoid_delta=ex_open(filename,EX_READ,&CPU_word_size,&IO_word_size,&versi
on);

  get_displ(exoid_delta,NSET,ns_num_nodes,ns_node_list, ns_X, ns_Y, ns_Z,
        ns_displx_delt);

  /***************/
  g2=abs(Pvacuum_nom/Ls);g2=1.0e7;
  J2= (ns_displx_delt[0] - ns_displx_nom[0])/(Pvacuum_delt - Pvacuum_nom);
  /*printf("J1 = %e , J2 = %e \n",J1,J2);*/
  return ALPHA*J1 + BETA*J2;
}
```

6. Now compile the code and run DAKOTA.

Appendix

This appendix will briefly describe the process of using DAKOTA with another analysis driver such as FIDAP. The procedure is basically identical to when GOMA is used for the analysis.

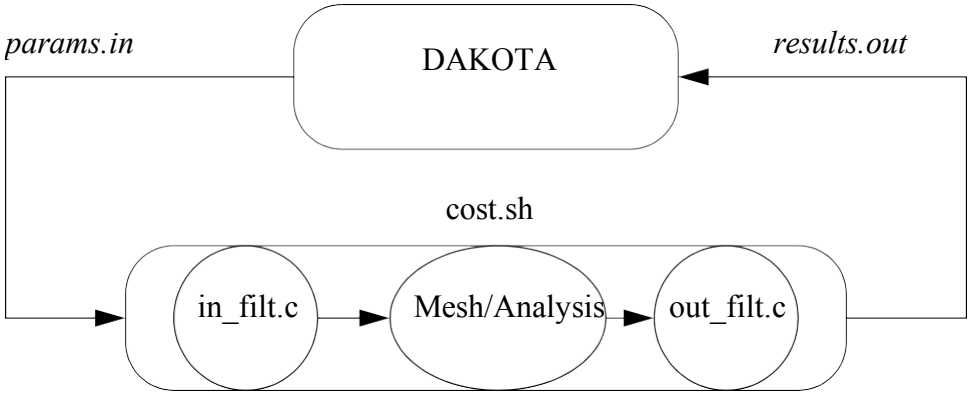

Figure A. DAKOTA interface scheme

1. Set up optimization by writing a DAKOTA input file. (See page 2-4 for example)
2. Write an input filter to take the file params.in generated by DAKOTA (format of parameter file is on page 8) and write and output file that can be used by your analysis code. An easy way to do this is to use APREPRO. If APREPRO is always used, the input filter in_filt.c can be written generally enough so that it can be used for all optimizations. (see pages 11-13)
3. Now parameterize your model so that the design variables that you want to vary can be easily changed by APREPRO. Make sure the output from your code has the information you will need to evaluate your cost function.
4. Write a program (out_filt.c) that takes the output from your code, evaluates your cost function, and writes a file (results.out) that (i) has the information requested from DAKOTA (this is specified in params.in) and (ii) is in a format that DAKOTA can read. (see pages 13 - 16)
5. In this tutorial, the programs that result from steps 2-4 are driven by a shell program cost.sh. DAKOTA, therefore, only has to call the shell program to evaluate the cost function.