Specifying and Reading Program Input with NIDR

David M. Gay
Optimization and Uncertainty Estimation
http://www.sandia.gov/~dmgay
dmgay@sandia.gov
+1-505-284-1456

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000. Released as SAND2008-6551C.
Outline

• Motivation: improve DAKOTA
  ◦ better error checking
  ◦ simplify maintenance

• DAKOTA overview and input form

• NIDR input specification

• Basic input processing

• Reordering algorithm

• Summary and conclusions
Goal: answer fundamental engineering questions

- What is the best design?
- How safe is it?
- How much confidence in my answer?

Challenges

- Reuse tools and interfaces, leverage commonalities → software
- Nonsmooth/discontinuous/multimodal, expensive, mixed variables, unreliable gradients, simulation failures → algorithm R&D
- ASCI-scale applications & architectures → scalable parallelism

Impact

- DOE: Tri-lab tool, broad application deployment
- External: WFO partners, GNU GPL (~3700 download registrations)
DAKOTA Framework

_strategy: control of multiple iterators and models

**Coordination:**
- Nested
- Layered
- Cascaded
- Concurrent
- Adaptive/Interactive

**Parallelism:**
- Asynchronous local
- Message passing
- Hybrid
- 4 nested levels with
  - Master-slave/dynamic
  - Peer/static

**Design**
- continuous
- discrete

**Uncertain**
- normal/logn
- uniform/logu
- triangular
- exp/beta/gamma
- EV I, II, III
- histogram
- interval

**State**
- continuous
- discrete

**Application**
- system
- fork
- direct
- grid

**Approximation**
- global
- polynomial 1/2/3, NN, kriging, MARS, RBF
- multipoint – TANA3
- local – Taylor series
- multifidelity
- ROM

**Response**
- objectives
- constraints
- least sq. terms
- generic

**Gradients**
- numerical
- analytic

**Hessians**
- numerical
- analytic
- quasi

**Parameters**
- Design
- continuous
- discrete

**Uncertain**
- normal/logn
- uniform/logu
- triangular
- exp/beta/gamma
- EV I, II, III
- histogram
- interval

**State**
- continuous
- discrete

**Application**
- system
- fork
- direct
- grid

**Approximation**
- global
- polynomial 1/2/3, NN, kriging, MARS, RBF
- multipoint – TANA3
- local – Taylor series
- multifidelity
- ROM

**Response**
- objectives
- constraints
- least sq. terms
- generic

**Gradients**
- numerical
- analytic

**Hessians**
- numerical
- analytic
- quasi

**Optimization**
- Uncertainty
- LeastSq

**Strategy**
- Hybrid
- SurrBased
- OptUnderUnc
- UncOfOptima
- ModelCalUnderUnc
- Branch&Bound/PICO
- 2ndOrderProb
DAKOTA Input Example

strategy,

    single_method

method

    max_iterations 100
    solution_accuracy = 1.e-6
    seed = 1234
    max_function_evaluations 1000
    apps
        initial_delta = .2
        threshold_delta = 1.e-4
variables

  continuous_design = 3
  initial_point   -1.0  1.5  2.0
  upper_bounds    10.0  10.0  10.0
  lower_bounds    -10.0 -10.0 -10.0
  descriptor      ’x1’ ’x2’ ’x3’

interface

  system asynch
  analysis_driver = ’text_book’

responses

  num_objective_functions = 1
  no_gradients          no_hessians
NIDR Processing

NIDR = New Input Deck Reader (replacing IDR)

NIDR parser-generator turns grammar file into C/C++ header file for table-driven NIDR parser.

Parser-generator runs only when grammar changes.

Parser calls routines mentioned in grammar file that modify data structures in response to keywords and associated values.

Each keyword can have start and stop routines called.
Grammar File Example (Simplified)

KEYWORD strategy

[ graphics ]

[ tabular_graphics_data
  [ tabular_graphics_file_file STRING ] ]

[ iterator_servers INTEGER ]

( single_method
  [ method_pointer STRING ] )

| ( multi_start
  method_pointer STRING
  [ random_starts INTEGER
    [ seed INTEGER ] ]
  [ starting_points REALLIST ] )
Grammar File Entries

Input: \textit{keyspec} [ \textit{keyspec} ... ]

Keyspec: \textit{keyword} [\textit{ALIAS keyword} ...] [\textit{Valuekind}]
\{\textit{startfunc}, \textit{startarg}, \textit{stopfunc}, \textit{stoparg}\}

Alternatives: \textit{keyspec} | \textit{keyspec} | ... 

Required

Grouping: ( \textit{initial\_keyspec} \textit{keyspec} ... )

Optional

Grouping: [ \textit{initial\_keyspec} \textit{keyspec} ... ]
Grammar File Detail Example (Detailed)

KEYWORD strategy {strategy_start}
[ graphics {N_stm(true,graphicsFlag)} ]
[ tabular_graphics_data
  {N_stm(true,tabularDataFlag)}
  [ tabular_graphics_file STRING
    {N_stm(str,tabularDataFile)} ] ]
[ iterator_servers INTEGER
  {N_stm(int,iteratorServers)} ]
( single_method
  {N_stm(lit,strategyType_single_method)}
  ...

Well-Ordered Input

Top-level keywords (e.g., strategy) and initial entries of required or optional groupings introduce contexts that can contain other keywords.

When all keywords from a context appear before the next keyword of an enclosing context, the input is well-ordered.
Processing Well-Ordered Input

Upon reading keyword K (and associated values),

while K is not in the current context,

Call the *stop* routine (if any) of the keyword that introduced the context;

Make the enclosing context current.

When K is found, call its *start* routine (if any);

if K introduces a new context, make that context current.
Start and Stop Routine Signature

struct Values {
    size_t n;
    Real *r;
    int *i;
    const char **s;
};

typedef void (*Kwfunc)(const char *keyname, Values *val, void **g, void *v);
void NIDRProblemDescDB::
method_start(const char *keyname, Values *val,
   void **g, void *v)
{
    DataMethod *dm = new DataMethod;
    if (!(*g = (void*)dm))
      botch("failure in method_start");
    dm->maxIterations = 100;
    dm->maxFunctionEvaluations = 1000;
}
Value Start Routine Example

```c
void NIDRProblemDescDB::
method_nnint(const char *keyname, Values *val,
    void **g, void *v)
{
    int n = *val->i;
    if (n < 0)
        botch("%s must be non-negative", keyname);
    (*(DataMethod**)g)->**(int DataMethod::**)v = n;
}
```
Context Stop Routine Example

```c++
void NIDRProblemDescDB::
method_stop(const char *keyname, Values *val,
    void **g, void *v)
{
    DataMethod *p = *(DataMethod**)g;
pDDBInstance->dataMethodList.insert(*p);
delete p;
}
```
Relaxed Input Ordering

Users prefer “cumulative distribution” to “distribution cumulative”.

Algorithm for relaxed input:

Maintain set of unrequited keywords.

Attach keywords to open contexts.

Close contexts before opening a new higher or parallel context.
Automating Some Error Checking

Automatically complain about

missing required keywords;

duplicate keywords;

missing values;

values of the wrong kind.

Manually complain about

values out of bounds;

lists of the wrong length.
Implementation Details

NIDR parser-generator written in *lex*.

NIDR run-time parser written in *lex* and *yacc*.

Parser-generator produces one table per context: array of *Keyword* structures.

NIDR source available under BSD license, including both source and C for parser-generator and runtime lexer.

DAKOTA source is under GPL license.
Summary and Conclusions

- Simple, general input processing
  - Nested contexts
  - Relaxed ordering
  - Error checking partly automated
  - Simple syntax overview
- Scales well — uses static tables
- Convenient use with C/C++
Some Pointers

http://www.cs.sandia.gov/DAKOTA

http://www.sandia.gov/~dmgay
