Dakota Software Training

Interfacing to a Simulation

http://dakota.sandia.gov

DAKOTA

CCR
Center for Computing Research

Sandia National Laboratories

Exceptional

service

in the

national

interest

U.S. DEPARTMENT OF **ENERGY**   **NNSA**
National Nuclear Security Administration

Module Learning Goals

Understand:

- … the mechanics of how Dakota communicates with and runs a simulation

- … the requirements this places on the user and interface

- … some basic strategies for developing a simulation interface

- … some of the convenience features Dakota provides for managing simulation runs

So you can interface Dakota to your relevant simulation
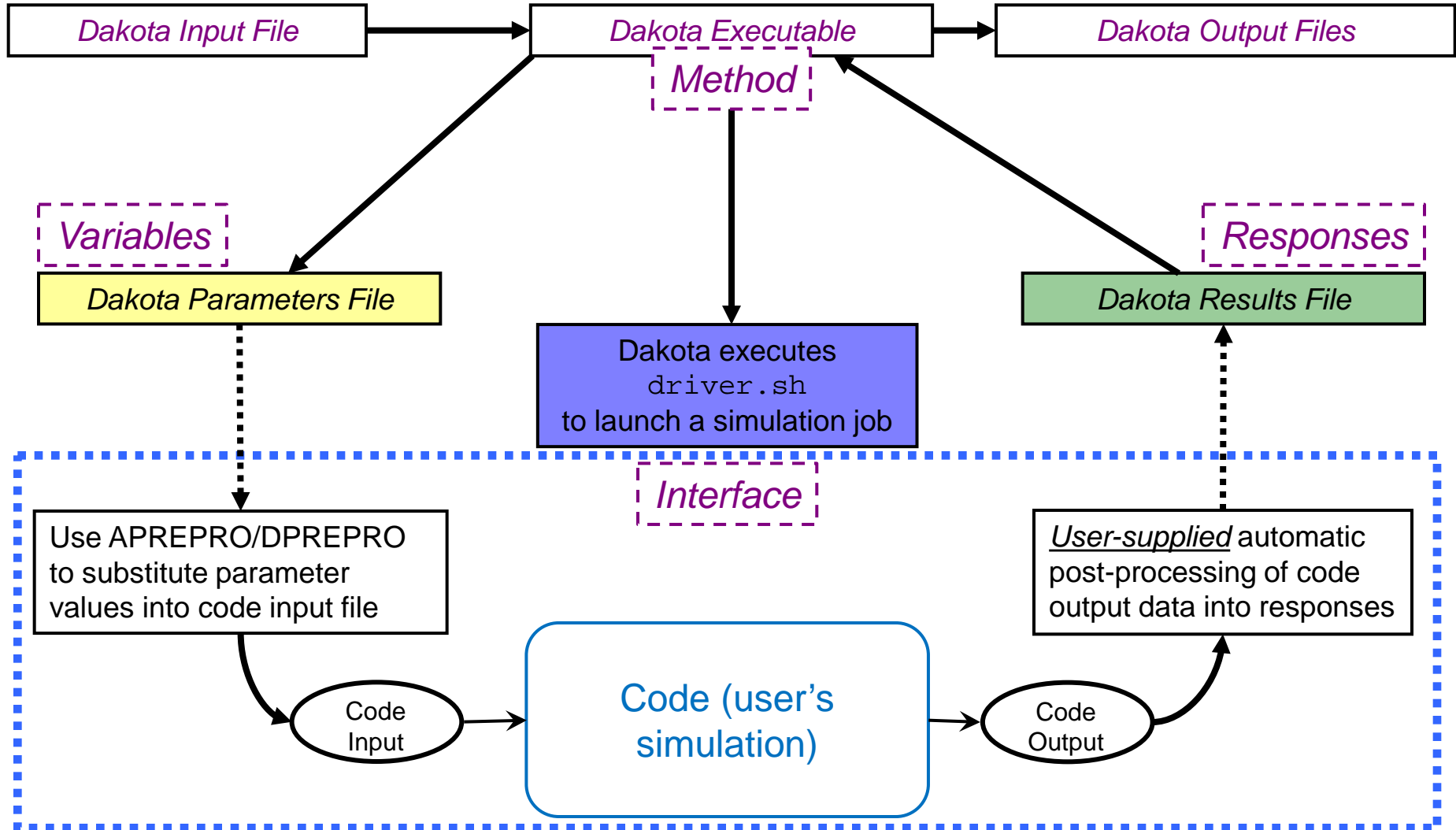
Module Outline

- What happens during a Dakota evaluation

- Considerations for creating a parameterized workflow

- Pre-processing

- Post-processing

- Demonstration

- Exercise

- Dakota input file: specifying an interface block

Interfacing

WHAT HAPPENS DURING A DAKOTA EVALUATION

Dakota to Simulation Workflow

Dakota Input File	→	Dakota Executable	→	Dakota Output Files

Method

Variables

Responses

Dakota Parameters File		Dakota Results File

Dakota executes `driver.sh` to launch a simulation job

Interface

Use APREPRO/DPREPRO to substitute parameter values into code input file

User-supplied automatic post-processing of code output data into responses

Code Input → Code (user's simulation) → Code Output

A Typical Dakota Evaluation

Each time Dakota wants to run the simulation:

1. Dakota writes a parameters file that contains one value for each variable

2. Dakota invokes the user's analysis driver, passing it the names of the parameters and results files as command-line arguments

3. The user's interface driver performs three tasks to map parameters to results:

   1. **Pre-processing:** Create simulation input using values from the Dakota parameters file

   2. **Run:** Run the simulation based on the input

   3. **Post-processing:** Extract scalar quantities of interest (responses) from simulation output and write them to the named Dakota results file

4. The user's interface exits

5. Dakota opens and reads the results file

params.in

Dakota

driver.sh
1. pre-process
2. run simulation
3. post-process

results.out
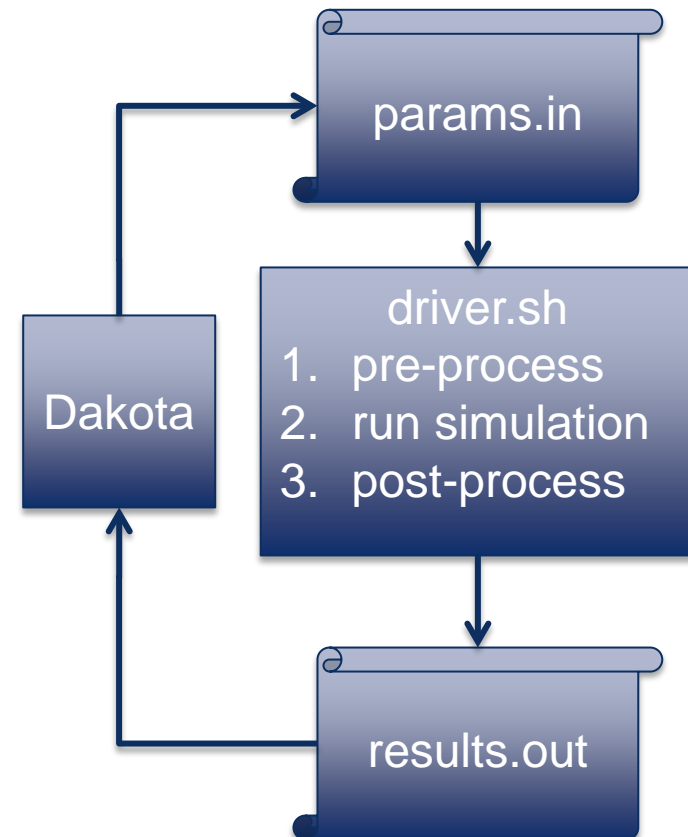
Dakota Input: Simulation Contract

- The **variables** block dictates what Dakota will place in the parameters file
- The **interface** block indicates what driver will run to perform the mapping
- The **response** block dictates what Dakota expects back

```
variables
  active all
  continuous_design = 3
    initial_point  1.0  1.0  10.0
    descriptors    'w'  't'  'L'
  continuous_state = 4
    initial_state   500.  29.E+6  5.  10.
    descriptors     'p'    'E'      'X' 'Y'

interface
  fork
  analysis_driver = 'driver.sh'

responses
  response_functions = 3
  descriptors = 'mass'
                'stress'
                'displacement'
  no_gradients no_hessians
```

Variables with descriptors:
w, *t*, *L*, *p*, *E*, *X*, and *Y*

Responses with descriptors:
*mass*, *stress*, *displacement*

7

Interfacing Preparedness

- "Parameterized" simulations/workflows
  - Must know what your parameters (variables) are
  - Tough if parameters are hard-coded

- Can my analysis be automated/scripted?
  - Does your workflow depend on tools that are challenging to automate on your target platform?
  - Is the simulation robust to parameter variations?

- Responses: simulation outputs Dakota will study
  - Also known as quantities of interest (QoIs);
    typically a few key values derived from the total simulation output
  - As with parameters, must know what the responses are
  - Can you extract them automatically?
  - If your response is poorly behaved (nonsmooth, noisy),
    is there another you could choose?

Interfacing

PRE-PROCESSING
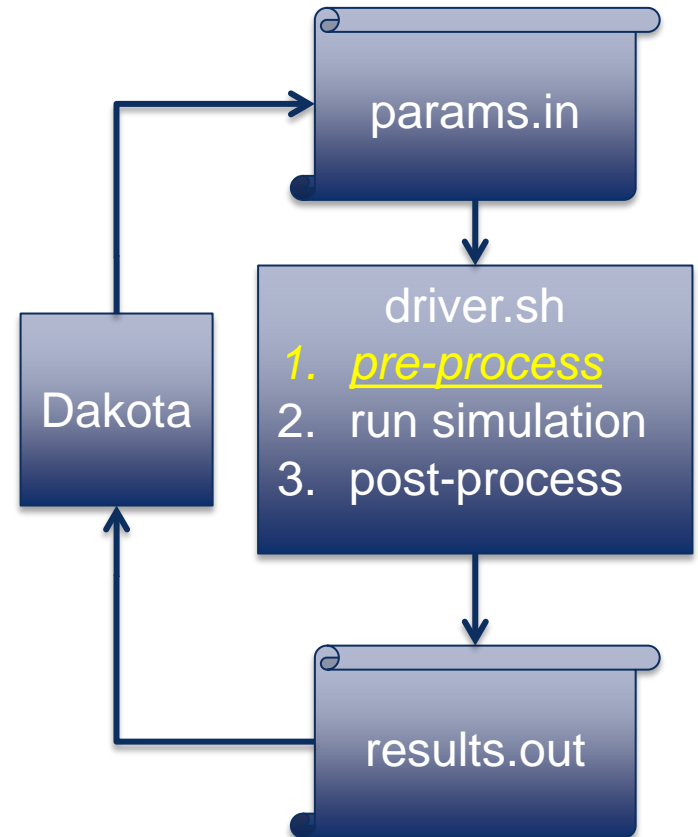
Interface Workflow: Pre-processing

What it is: Converting a Dakota parameters file into usable input for your simulation

Example tasks:

- Substituting parameter values into a text-based input deck ("configuring" an input file)

```
material 2 elements 3 4
variable 1 -1.0
variable 2  1.0
end
```

*nominal simulation
input file*

```
material 2 elements 3 4
variable 1 {x1}
variable 2 {x2}
end
```

*templated simulation input file,
accepting Dakota parameters
x1 and x2*

Interface Workflow: Pre-processing

What it is: Converting a Dakota parameters file into usable input for your simulation

Example tasks:

- Passing parameter values to a simulation as command line arguments, e.g., `how2getrich.exe --param1=42 --param2=-1.4`

- Running a script or program to generate more complex input based on parameter values, e.g. parameterized mesh or geometry in a finite element analysis

| Dakota parameters | → | genmesh.py | → | mesh.exo | | sim.input mesh = 'mesh.exo' |

files needed by simulation
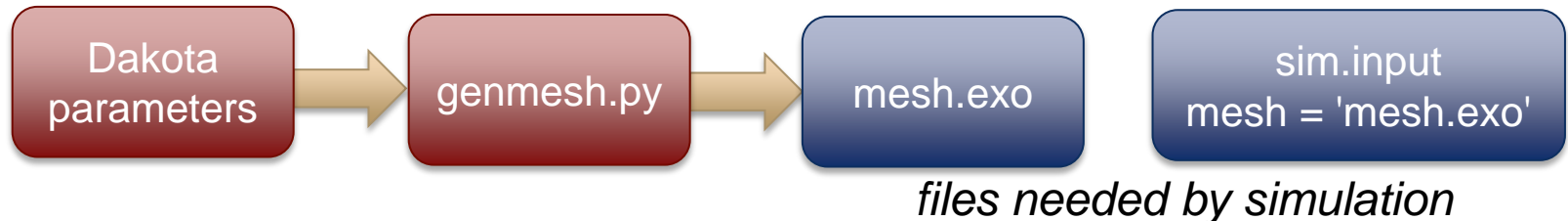
Parameters File Format

Dakota uses a parameters file to inform your code of parameter values and to request responses. (See sections 9.6 and 9.7 of the User's Manual)

```
                      6 variables
2.500000000000000e+000 w
2.500000000000000e+000 t
5.000000000000000e+000 L
4.000000000000000e+004 R
2.900000000000000e+007 E
5.000000000000000e+002 X
1.000000000000000e+003 Y
                      3 functions
                      1 ASV_1:mass
                      1 ASV_2:stress
                      1 ASV_3:displacement
                      6 derivative_variables
                      1 DVV_1:w
                      2 DVV_2:t
                      3 DVV_3:R
                      4 DVV_4:E
                      5 DVV_5:X
                      6 DVV_6:Y
                      0 analysis_components
                      1 eval_id
```

Parameter Values

Requested Responses

Parameter Substitution with dprepro

dprepro (**D**akota **Pre**-**Pro**cessor) is a Dakota team-developed command line tool to simplify parameter substitution.

1. Create a template input file, e.g., `sim.template`, for your simulation with parameter values replaced by substitution tokens.

   Tokens look like: `{dakota_descriptor}`

2. Run dprepro, which replaces tokens with Dakota variable values to create the simulation input file for this evaluation.
   Syntax:
   ```
   dprepro <dakota parameters file> <sim template file>
           <sim input file>
   ```
   Example:
   ```
   dprepro params.in sim.template sim.input
   ```

Parameter Substitution with dprepro

```
variables
 continuous design = 2
    descriptors 'dak_x1' 'dak_x2'
```
← Dakota input file `dakota.in`

`sim.template:`
User-created template →
```
x1 = {dak_x1}
x2 = {dak_x2}
```

`params.in:`
Dakota parameters file →
```
                                      2 variables
       2.500000000000000e+000 dak_x1
       4.500000000000000e+000 dak_x2
```

```
dprepro params.in sim.template sim.input
```

```
x1 = 2.5
x2 = 4.5
```
← `sim.input:`
Run-ready simulation input file, written by dprepro

Additional dprepro Features

- **Token delimiters** (left and right braces) can be changed to other characters or strings:
  - `dprepro --left-delimiter='[[' --right-delimiter=']]' ...`

- Simple **arithmetic statements** within tokens will be evaluated.
  - Exponentiation: `{10.0**dakota_descriptor}`
  - Multiplication: `{dakota_descriptor_1 * dakota_descriptor_2}`

- **Default parameters**:
  - `{dakota_descriptor = 3.0}` is replaced by the value of dakota_descriptor if present in the parameters file; 3.0 otherwise.

- See also: `dprepro --man`

Exercise: Using dprepro

1. Change directory to `exercises/dprepro` and examine `dakota_pstudy.in` and the example Dakota-generated `params.in`

2. Edit the mock simulation input file `ros.nominal`, saving as `ros.template`:
   1. Replace the nominal values of variables 1 and 2 with template parameter placeholders for Dakota variables x1 and x2
      optionally leave the default values, e.g., {x1 = -1.0}
   2. Parameterize the y coordinate of nodes 2, 4, 6, and 8 with `yc_upper`
   3. Parameterize the x coordinates of all nodes with a multiplier `xc_mult` on their default values

3. Run dprepro with the parameters and template files to produce ros.input

4. Verify your resulting ros.input against ros.gold (`diff ros.gold ros.input`). If there are differences, revisit the template editing step.

5. Experiment with other dprepro options that may be relevant for you
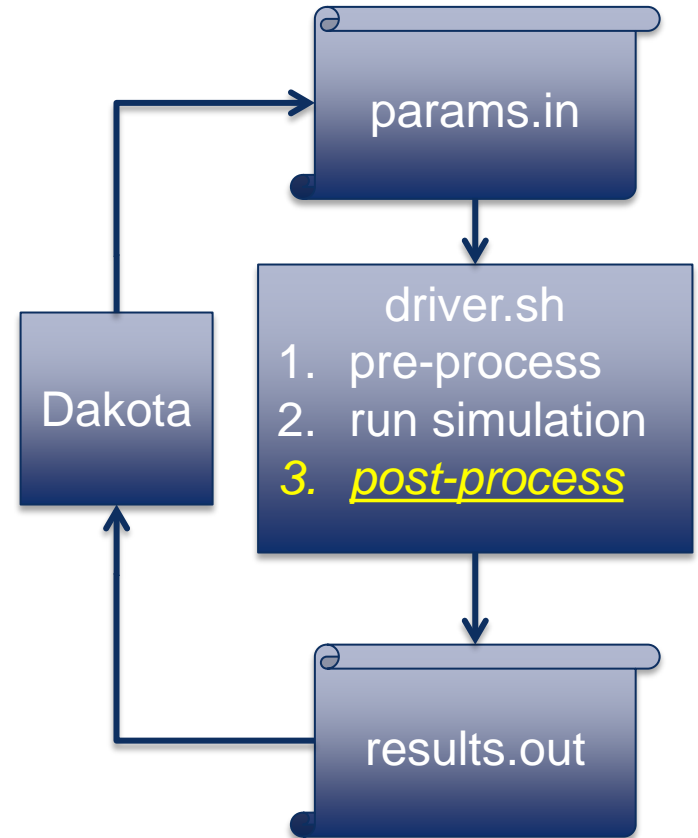
Interfacing

RUN
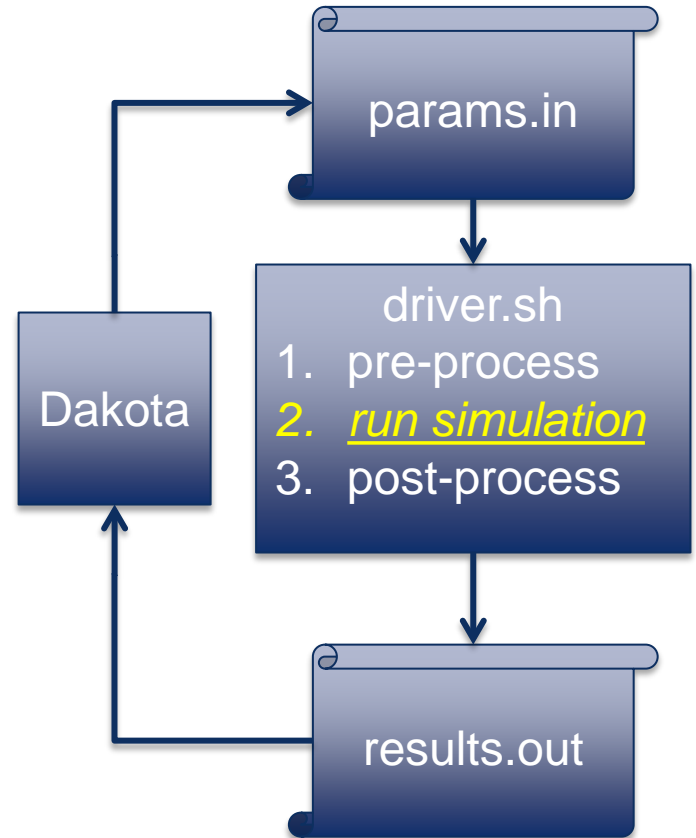
Interface Workflow: Run

- The run phase of the workflow must batch automate the execution of the simulation

- Often as simple as running an executable (or script), given relevant input files:

  ```
  mysimulation.exe sim.input mesh.exo
  predict_weather.py weather.config climate.dat
  ```

- Can be more complex:
  - Submit a job to an HPC cluster or other job server
  - Automate mouse clicks to run a GUI-only application
  - Email the inputs to a colleague and wait for the answer to come back

Interfacing

POST-PROCESSING

Interface Workflow: Post-processing

What it is:

1. Automatically extracting quantities of interest (responses) from simulation output

2. Placing them in a properly formatted and named Dakota results file

- Can be challenging; no quick shortcuts like dprepro!

- If the QoI appears in console or text file output, time-honored Unix/Linux text processing commands can often extract it:
    - grep, cut, awk, head, tail, tr, bc, etc.
    - See http://www.tldp.org/LDP/abs/html/textproc.html for a nice list with descriptions, instructions, and examples.

- It may be necessary to calculate a QoI from simulation output, like the max, min, or integral of a trend

Results File Format

```
responses
  response_functions = 3
  descriptors = 'mass'
                'stress'
                'displacement'
  no_gradients no_hessians
```
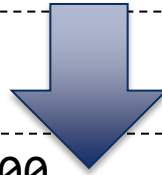
```
6.2500000000e+00    mass
1.7600000000e+04    stress
1.6943165672e+00    displacement
```

Simulation post-processing step must produce a file Dakota can read with:

- One response per line
- Value (format unimportant) followed by (optional) descriptor
- Descriptors are ignored; order is what matters!
- No spaces in descriptors

(See Chapters 10 & 11 of the User's Manual for more information.)

Discussion: Post-processing

- Share with your neighbor a quantity of interest you wish to study with Dakota.

- In what kind of output file does it appear?

- Is it easy to extract for return to Dakota? Why or why not?

Discussion: Post-processing

- Share with your neighbor a quantity of interest you wish to study with Dakota.

- In what kind of output file does it appear?

- Is it easy to extract for return to Dakota? Why or why not?

- Potential topics:
  - Circuit simulator that outputs time/voltage traces in columnar tabular data; want to extract voltage 4 at time 0, time final, and its max.
  - CASL VERA, which outputs on reactor geometry to binary HDF5; want to extract the outlet pressure and the average pin power.
  - Finite element analysis: force integrated over a sideset or nodeset.

- Tools: shell commands, Python, Perl, Matlab, C, C++, Fortran, Java, VBS, ParaView, VTK, VisIt, etc.

Interfacing Recipe

1. **Identify** relevant variables and responses,

2. **Sketch, then build** a scripted workflow, e.g., `driver.sh`…

3. **Develop** a dprepro text template (or other process) to convey a Dakota parameters instance to the simulation input files
   Test: does this pre-processing step produce valid simulation input, given a mocked-up Dakota parameters file?

4. **Automate** the execution of the simulation
   Test: does the pre-processing and run together produce valid output?

5. **Automate** the simulation output post-processing to produce a Dakota results file containing the responses for Dakota study

6. **Test the workflow**: run and check; does it map `params.in` to `results.out`?
   driver.sh params.in results.out

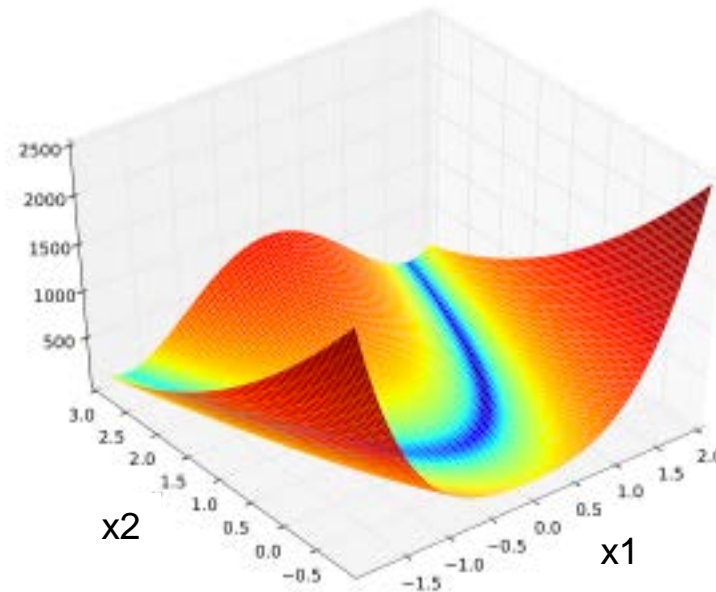7. **Test with Dakota**: create a simple Dakota input file that runs the simulation once and make sure it works

Interfacing

EXAMPLE AND EXERCISE

Example:
Rosenbrock Black Box Interface

Rosenbrock Function

Courtesy Wikipedia
https://en.wikipedia.org/wiki/Rosenbrock_function

- Walk through example (Section 10.3 in Dakota User's Manual)
- Example files in `dakota/examples/script_interfaces/generic`

Rosenbrock Interface Summary

Dakota writes *before* it runs simulator_script

dakota_ rosenbrock.in

Describes variables and responses to Dakota

Dakota reads *after* it runs simulator_script

Preprocessing	**Run**	**Postprocessing**

params.in

+

ros.template

dprepro →

ros.in

rosenbrock_bb →

ros.out

grep, mv →

results.out

Add substitution tokens

simulator_ script

Interface script; Dakota runs once per evaluation

ros.nominal

Nominal simulation input

27

Red….Created once, invariant
Blue…Changes every evaluation
Gold...Operations

Sandia National Laboratories

General Interfacing Advice

- Think about automation from the beginning

- Sketch the workflow you are automating, and how Dakota will interact with it

- Test each component/step (dakota input, pre-processing, run, post-processing) in isolation

- Capture the results of each step to log files

- If a step fails, how would I know?
    - Silent failures, where an erroneous value is returned to Dakota rather than a crash, are the most dangerous!

- Avoid long chains of tools that multiply points of failure

- Treat your interface as though your conclusions depend on it!

Interfacing Recipe

1. **Identify** relevant variables and responses,

2. **Sketch, then build** a scripted workflow, e.g., `driver.sh`…

3. **Develop** a dprepro text template (or other process) to convey a Dakota parameters instance to the simulation input files
   Test: does this pre-processing step produce valid simulation input, given a mocked-up Dakota parameters file?

4. **Automate** the execution of the simulation
   Test: does the pre-processing and run together produce valid output?

5. **Automate** the simulation output post-processing to produce a Dakota results file containing the responses for Dakota study

6. **Test the workflow**: run and check; does it map `params.in` to `results.out`?
   driver.sh params.in results.out

7. **Test with Dakota**: create a simple Dakota input file that runs the simulation once and make sure it works

Scenario: Coat Hook Design

Scenario: Your manager would like to place some coat hooks in your building's lobby. To ensure they will safely hold heavy winter coats, he asked you to conduct a computational study, suggesting they are similar to cantilever beams.

You immediately assign this critical task to a trusted intern. She downloads the latest version (12.1.4) of *Cantilever Physics*, an advanced cantilever beam simulation tool developed by Sandia National Laboratories, and gets to work. After she shows you the results of a few test cases, you remain uncertain about the design of the coat hooks. It occurs to you that Dakota could help you to achieve greater confidence in your analysis, and you ask her to begin developing an interface between Dakota and Cantilever Physics.
Unfortunately, the summer ends before she is able to finish, leaving you on the hook to complete the job.

Your task: Follow the Interfacing Recipe to complete the Dakota-Cantilever Physics interface that your intern left unfinished.

Exercise: Cantilever Beam Interface

Follow the Interfacing Recipe to create a working Dakota to cantilever simulation interface.

Exercise materials are located in `~/exercises/interfacing`. They include:

- The Cantilever Physics executable, `cantilever`, along with an old input file named `cantilever.i`. Try running it:

    ```
    ./cantilever cantilever.i
    ```

- A Dakota input file, `dakota_cantilever_centered.in`

- An example Dakota parameters file, `params.in`, which you can use for testing. (Generate your own `params.in` by adding the `file_save` keyword to the Dakota input file.)

- `driver.sh`, a partially-complete interface shell script (you may opt to use a different programming language if you prefer)

Cantilever Interface Hints

Preprocessing

Run

Postprocessing

Dakota writes *before* it runs driver.sh

dakota.in

Describes variables and responses to Dakota

Dakota reads *after* it runs driver.sh

params.in

+

???

dprepro

cantilever

???

???

???

results.out

???

cantilever.i

Nominal simulation input

driver.sh

Interface script; Dakota runs once per evaluation

Red….Created once, invariant
Blue…Changes every evaluation
Gold...Operations

Interfacing

DAKOTA INPUT

Dakota Input - Interface Block

The interface section of the input file tells Dakota how to run a simulation

Example:

```
interface
 analysis_drivers = 'driver.sh'

  fork

  parameters_file = 'params.in'
  results_file = 'results.out'

  file_tag
  file_save
```

← Interface executable name

← Interface type

← Names of param. and results files

← Save and tag param. and results file

Discussion: Browse to the Dakota Reference Manual, interface section and call out some keywords that look particularly helpful.

Excursis: Fork vs System vs Direct

- **`direct`** interface is used to run an analysis driver that has been linked or compiled into Dakota
  - Largely beyond the scope of this training
  - One matter of practical interest: text_book
  - See the Developer's Manual on the website for more information
- **`fork`** and **`system`** are used to launch an external simulation
  - The fork interface waits for your analysis driver to exit, then tries to open your results file
  - The system interface polls the OS for the existence of the results file, and tries to read it as soon as it is written
  - Practical consequence: When using the system interface, don't create the results file until the very end of your interface script
  - *In general, we recommend the fork interface*

What else can Dakota do?

- Work directories separate (concurrent) evaluations

```
work_directory
  named  'workdir'
  directory_save directory_tag
```

- Naming optional
- Directories deleted by default
- Dakota writes params file to each work directory, and launches interface from there.

```
rundir/
  workdir.1/  workdir.2/  workdir.3/
  workdir.4/  workdir.5/
```

- Work directories can share common template files

```
work_directory ...
  link_files 'template/input.template'
  copy_files 'modified/*'
```

- Copy or link files or entire directories into your work dir before launching interface
- Wildcards permitted

```
workdir.1/
  *
  input.template
```
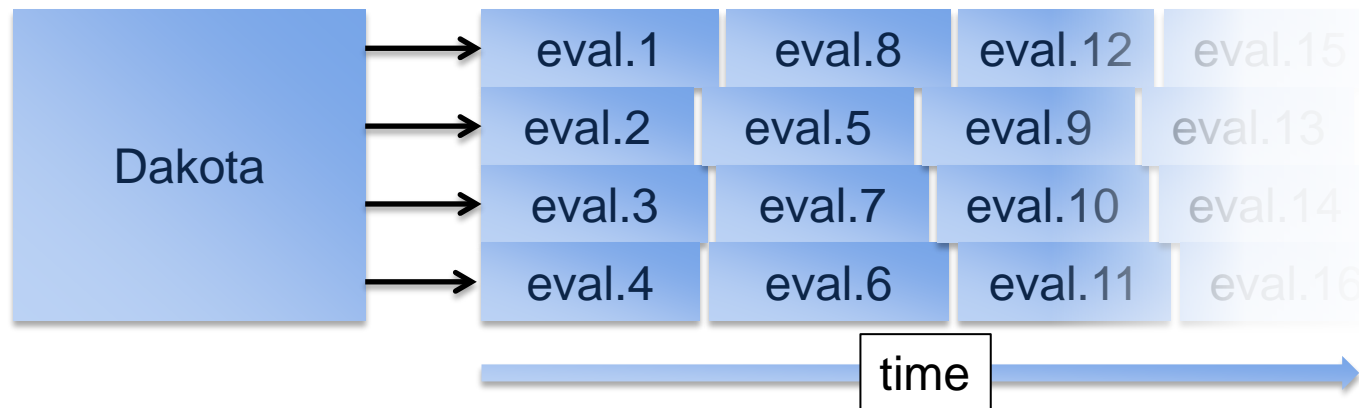
What else can Dakota do?

- Asynchronous execution helps with local parallelism

```
asynchronous
   evaluation_concurrency 4
```

- Run multiple evaluations concurrently
- By default, ALL available evals will run
- MUST use `file_tag` or `directory_tag`

Your Simulation

- Ideally, you will leave training this week with a working interface to a simulation of relevance to you

- Give it a shot this evening and discuss during office hours