44th AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit
21 - 23 July 2008, Hartford, CT

AIAA 2008-4860

Design of a Model Execution Framework: Repetitive Object-Oriented Simulation Environment (ROSE)

Justin S. Gray* Jeffery L Briggs†

The ROSE framework was designed to facilitate complex system analyses. It completely divorces the model execution process from the model itself. By doing so ROSE frees the modeler to develop a library of standard modeling processes such as Design of Experiments, optimizers, parameter studies, and sensitivity studies which can then be applied to any of their available models.

The ROSE framework accomplishes this by means of a well defined API and object structure. Both the API and object structure are presented here with enough detail to implement ROSE in any object-Oriented language or modeling tool.

I. Introduction

Today, state of the art systems analysis methods revolve around the ability to execute a given system model using statistical methods and optimizers. These methods need to be repeatable and widely applicable to a range of models, regardless of what modeling environment is used. There are many different programming languages and as many different modeling environments currently in active use today. C++, Java, Python, Fortran, Simulink, Model Center, EnSight, and NPSS are just are just a few from the available language and modeling environment options. With so many options, there is a need for a unifying structure to define analysis processes and solution paths which can be easily applied in any language or modeling environment available.

A second feature of almost any modern analysis methodology is the large case sets required to complete them. Optimizations and statistical methods all rely on the computation power available today to execute many simulations of a given model in order to obtain the necessary information about the behavior of a model. Very often, the many individual simulations are completely independent of each other and can be executed concurrently, assuming the necessary parallel computing resources are available. In recent years, large clusters of inexpensive servers have provided an unprecedented level of parallel computing power. Similarly, in very recent time even desktop workstations have come equipped with multi-core CPU's which can make great use of parallelized processes. Any model execution framework needs to be able to seamlessly take advantage of these types of computation resources as they become available.

The ROSE framework was designed primarily to serve the needs of a modern systems analysis methodology. It addresses both the issue of creating general repeatable methodologies, as well as challenges involved with utilizing parallel computing resources. Presented here is a structural outline of the ROSE framework detailed enough to implement it in the language and/or modeling environment of your choice, along with an example implementation written in Python.

II. ROSE Design Philosophy

The Repetitive Object-Oriented Simulation Environment (ROSE) was designed to be simple to be implemented in any object-Oriented programming language or modeling environment. Languages such as Python[7] or Modeling Environments such as the Numerical Propulsion Systems Simulation (NPSS)[5] tool are both perfect examples of candidates for ROSE implementations. Analysis methods that are implemented using ROSE can be easily ported to any environment where ROSE itself has been implemented.

*Aerospace Engineer, Airbreathing Systems Analysis Office, 21000 Brookpark Rd., Cleveland, OH 44135
†Aerospace Engineer, Airbreathing Systems Analysis Office, 21000 Brookpark Rd., Cleveland, OH 44135

To ensure that the analysis methods developed do not become specific to any given model, any proposed framework would need to completely divorce the model from the analysis method.[3] ROSE accomplished this, and in doing so allows the analyst to develop analysis tools without undue consideration for the specifics of the model to be analyzed. Of course, certain model characteristics will inevitably affect the choice of available analysis methods. For instance, complex CFD simulations are slow and can be computationally unstable so it may be impossible to use one directly inside of a large optimization where it would need to be executed thousands of times. In that situation, a surrogate model would be appropriate.[2] However the important point is that whether the model is a surrogate or a CFD simulation of a wing in flutter, or a 1-D turbine engine model of a high-bypass turbofan, the same optimizers and probabilistic analysis tools developed in ROSE will still be applicable.

To help generalize the interface between a Driver and a model the concept of model attributes being described as key-value pairs is used within ROSE objects. All model attributes can be described by a key, representing the name of the attribute, and a value representing the data contained by the attribute. An attribute can be nearly any data type, and ROSE makes no specific assumption about which types will be used. This dissociation helps to further insulate a ROSE Driver design from any single language or modeling tool. By design, an analyst can use ROSE to generate many distinct Drivers, creating a library of possible analysis tools to choose from for any given analysis.

III. ROSE API

ROSE provides an analyst with the ability to create any type of model execution process he or she requires. To accomplish this a number of tools are made available to structure optimizers, parameter studies, design of experiments studies, uncertainty studies, etc. Each model execution process defined by an analyst is called a Driver.

A. Driver Structure

To build a Driver, ROSE provides five classes to use as building blocks. These 5 classes work together to capture all the parts of an execution process. The classes are listed below along with brief descriptions of their functions.

- Iterator: Provides a Driver with model input values for each simulation
- Outerator: Records model outputs from each simulation a Driver runs
- Case: Data structure describing the inputs and outputs for a given simulation.
- Operator: Responsible for communicating with whatever server or software being used for model simulation
- Conductor: Provides for communication between other ROSE classes and servers

These classes serve as a skeleton for more complex objects needed to build Drivers. They can each be extended to create objects suited for specific tasks. For example, a simple Iterator can be created to read off case inputs from a comma separated file, while a different Iterator can execute the complete set of available combinations from a set of inputs and their allowed values. Similarly, many different Operators can exist to facilitate communication with different computer clusters. Different Outerators can produce output in different forms, i.e. textual or database outputs.

Regardless of the particular combination of Iterator, Outerator, Conductor, and Operator employed, all Drivers employ an iterative process to successfully complete a set of executions. Figure 1 portrays an effective iterative process that can be used for a wide range of Drivers. Here it is clear that execution begins in the Operator, whose Start() method will grab a ready server and pass a pointer to it to the Conductor, via a call back method. Then the Conductor is responsible for determining the proper action to do with the given server, and utilizes the Iterator, Outerator, and Case objects to do it. Each object and its full role in any given Driver are explained in detail in the following sections.

1. *Case*

A Case is a data structure which holds the input and output information for a given simulation. It provides the following methods:

- setInput(key,value): Takes in an attribute key and a corresponding value, which it then adds to the list of key, value pairs for the case. Returns success or failure.
- setOutput(key,[value]): Takes in a attribute key and optional value. Allows the user to add an attribute to be tracked as output, or allows user to set the value of an attribute associated with the given key.
- getInput(key): returns the input value associated with the prescribed key.
- getInputs(): returns a list of all key, value pairs of inputs in a case.
- getOutput(key): returns the output value associated with the prescribed key.
- getOutputs(): returns a list of all key, value pairs of outputs in a case.

2. *Iterator*

An Iterator is responsible for serving up cases to the rest of a Driver. There are no restrictions placed on how an Iterator determines which case or cases to serve and in what order they are served. It provides the following methods:

- initialize(): initializes the Iterator. Can be called at any time to re-initialize. Expected to return success or failure.

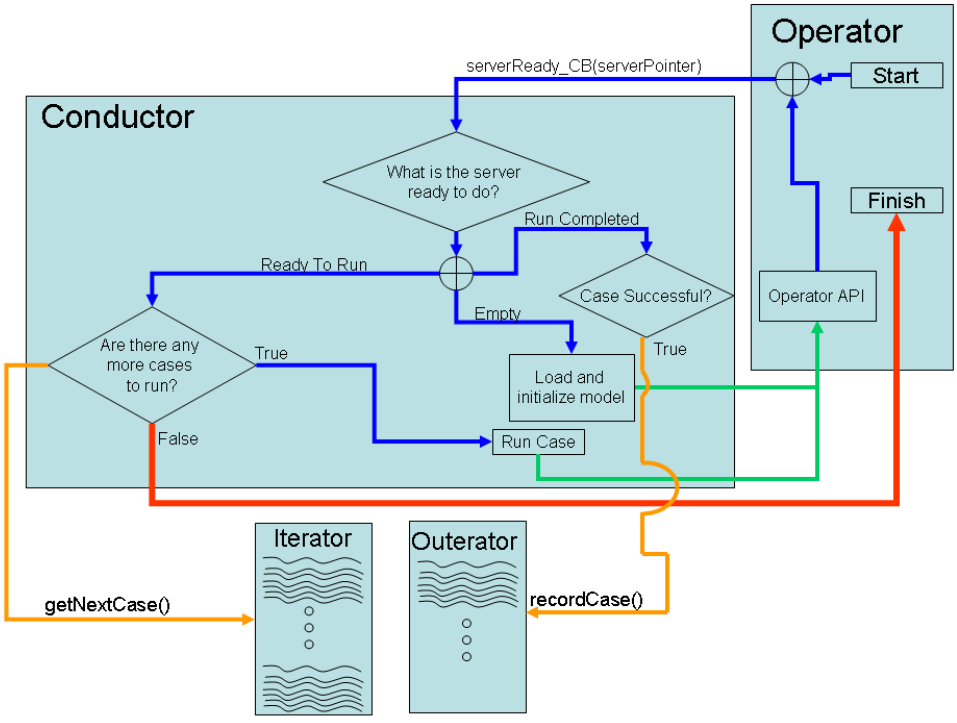


Figure 1. Basic Driver iterative process.

- getNextCase([Number]): Takes an optional argument indicating how many cases should be returned in a list of cases, defaults to a single case. Returns a list of pointers to Cases. It is expected to return an empty list of pointers when there are no more cases to execute.
- cleanUp(): called at the end of a driver execution to allow the Iterator to perform any necessary operations such as releasing file resources or closing databases.

3. Outerator

An Outerator records the specified output from a given case. It provides the following methods:

- initialize(): initializes the Outerator. Can be called at any time to re-initialize. Expected to return success or failure.
- recordCase(caseName): Takes a pointer to a Case as an argument and records the output from that Case. Expected to return success or failure.
- cleanUp(): called at the end of a drive execution to allow the Outerator to perform any necessary operations such as releasing file resources or closing databases.

4. Operator

An Operator is an object which acts as a translation layer between the other objects in a Driver and the server or servers which that Driver must communicate with. By abstracting the communication between the other Driver objects and Servers, the Operator class provides a simple way to switch between computational resources without changing the rest of a Driver or the underlying analysis processes. In Figure 2 the abstraction is evident.

Operators are a unique class because they handle event driven sections of an analysis process. With an array of available servers the Operator will initiate a given simulation, using a set of call back functions to a Conductor, when any server is ready to handle a simulation. An Operator should issue call backs based on servers becoming available until it is told to stop by an associated Conductor.

Operators provide the following methods:

- initialize(): called to initialize the Operator. Can be called at any point to re-initialize. Should open connections to all servers and verify functional connection. Expected to return success or failure.
- setConductor(conductorpointer): Specifies which Conductor to issue call backs to. Must be set before a start() is called.
- start(): initiates the call backs to the paired Conductor. Expected to return success or failure.
- cleanUp(): called at the end of a drive execution to allow the Operator to perform any necessary operations such as closing connections to servers or releasing system resources such as shared memory.
- loadModel(serverpointer): Conductor issued call to load a model to the specified server. This function will instantiate a model on the server and initialize it. After a model is loaded, it is expected to be ready to execute cases. Expected to return success or failure.
- modelSet(serverpointer, key, value): Conductor issued call to be used before a execution to set the value of the specified model attribute. Is not called during model execution. Expected to return success if the attribute value was set or failure if it was not.
- modelGet(serverpointer, key): Conductor issued call to be used before or after model execution. It is not called during model execution. Expected to return the value of the model attribute associated with the given key, or to return failure.
- modelExec(serverpointer): Conductor issued call to initiate the execution of a model. Expected to return success or failure of the initiation of the model execution.
- modelSuccess(serverpointer): Conductor issued call to test for successful execution of a model. Expected to return True if the execution succeeded and False if it failed.

- modelCleanUp(serverpointer): Conductor issued call used after a model execution has been completed to perform any necessary clean up operations on the model. This function can release system resources, close files, or do nothing. Expected to return success or failure.

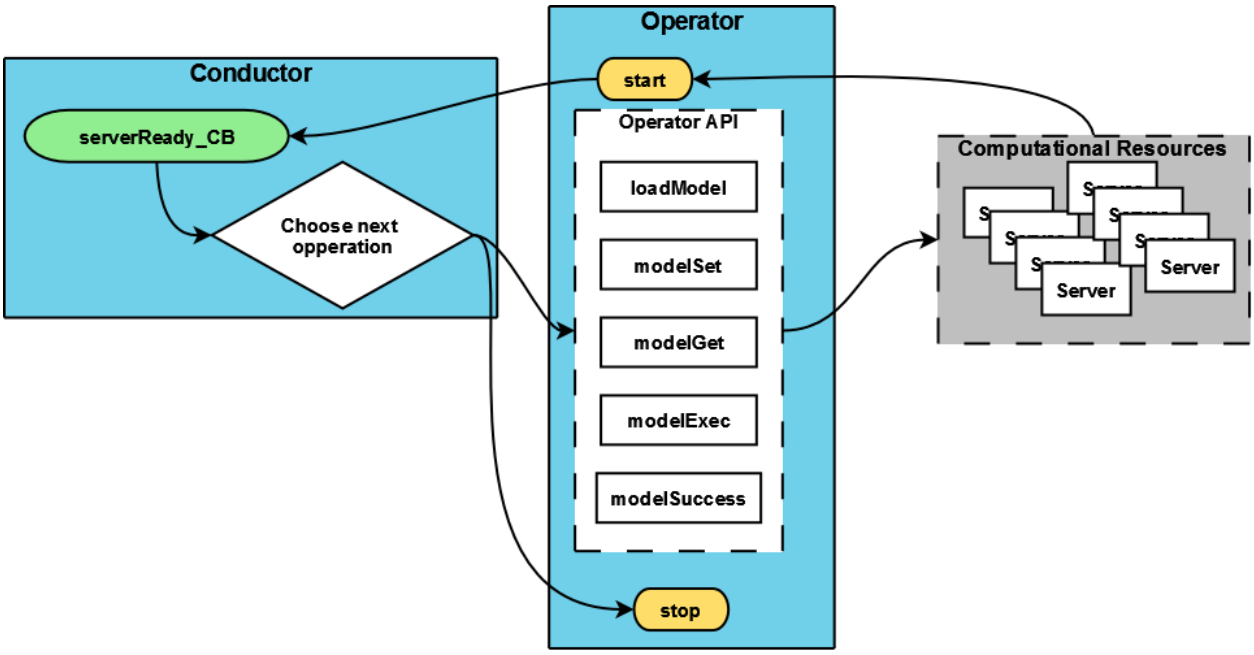


Figure 2. Operator, Conductor communication loop.

5. *Conductor*

A Conductor provides a means of communication between all the other objects in a Driver. By using a Conductor to tie together the Iterator, Outerator, Operator, and case objects it becomes simple to swap out components without needing to change the rest of the logic that ties a Driver together.

A Conductor is designed to respond to call backs from an Operator. An Operator relies on a Conductor to indicate whether or not there are any more cases to run, and so whether or not to continue to issue call backs. In Figure 2 this sequence is illustrated. The process begins at start, and runs through until the Conductor issues a stop to the Operator.

A Conductor also provides the capability of a ROSE Driver to execute independent cases concurrently. It abstracts the asynchronous software requirements away from the rest of a Driver by maintaining relevant information about which server is doing what actions. Hence, when a Conductor receives call backs from an Operator, it is always given a pointer to the server that triggered the call back. Then it is the Conductor's responsibility to decide what to do with the server via the Operator API.

Conductors provide the following methods:

- initialize(): prepares the conductor to begin executing cases. Expected to return success or failure. After return, if successful, a conductor is considered ready to begin executing cases.
- setOperator(operatorpointer): indicates which Operator the Conductor is currently paired to. Must be called before any call backs can be issued.
- setIterator(iteratorpointer): identifies which Iterator a Conductor should communicate with. Must be called before any call backs can be issued.
- setOuterator(outeratorpointer): identifies which Outerator a Conductor should communicate with. Must be called before any call backs can be issued.

- serverready_CB(serverpointer): call back issued by the Operator indicating that the server, whose pointer is passed as the argument, is ready and waiting for the Conductor work with it. Returns a boolean: True if the Operator is to continue issuing call backs for that server, False if the Operator is to stop issuing call backs for that server.

IV. Implementation Example

A. Overview

To demonstrate the viability of the ROSE framework, an example implementation was developed using Python 2.5.[7] Python was chosen because it provided a well developed and powerful scripting language along with a set of powerful extension modules for matrix manipulations[6] and 2-D or 3-D data plotting.[4] A generic ROSE Driver was developed which allowed a Design of Experiments(DOE) to be executed on an arbitrary Numerical Propulsion Systems Simulation (NPSS)[5] model. This Driver was then used in conjuction with a DOE designed specifially for a complex model of a medium-bypass mixed flow turbojet model, to facilitate a complex system level optimization of a supersonic business jet.

The system level optimization contained a large number of design variables, resulting in a large number of simulations required by the optimizer to converge on a design. While NPSS is a powerful and flexible cycle analysis tool, it is also a fairly computationally intensive tool. A single case can take more than ten seconds to execute. Consequently using NPSS directly inside the aircraft optimization was not feasible.

To conquer this issue, the ROSE DOE Driver executed the specified DOE (over 20,000 cases) on the given NPSS model, and the resulting data was used to build a fast executing meta model using Response Surface Equations. The large case set was chosen to better illustrate the ROSE DOE Driver's capability. Though the execution of all 20,000 cases took about three days, the resulting metamodel could then be run many 10,000's of times in only minutes, thus drastically reducing the computational load of the engine calculations in the larger aircraft optimization.

	A	B	C
1	Fan_Pressure_Ratio	Overall_Pressure_Ratio	Bypass_Ratio
2	1.2	20	5
3	1.2	25	5
4	1.3	20	5
5	1.3	25	5
6	1.4	20	5
7	1.4	25	5
8	1.2	20	6.5
9	1.2	25	6.5
10	1.3	20	6.5
11	1.3	25	6.5
12	1.4	20	6.5
13	1.4	25	6.5
14			

Figure 3. Simple DOE created with Spreadsheet software.

B. Iterator Design

The first step in executing a DOE was to read in cases from that DOE. A comma separated value (CSV) file, where the first row was used as a header row indicating the model attribute referred to in each DOE column, was chosen as the DOE format. Successive DOE rows contained the attribute values to be used for each case. This format is simple to generate in an any spreadsheet program. Using this format, any extra model attributes that were needed for a simulation would simply be added by creating extra columns in the DOE. Similarly, any extra cases could be added by simply appending extra rows to the DOE.

As an example, for the mixed flow turbo fan model, Figure 3 is a very simple DOE that would run 12 simulations representing a simple study of the effects of fan pressure ratio, overall pressure ratio, and bypassRatio on engine performance.

To read this DOE the Iterator parsed rows from the CSV file, and placed the data into a Case objects, whenever the getNextCase() method was called by a Driver.

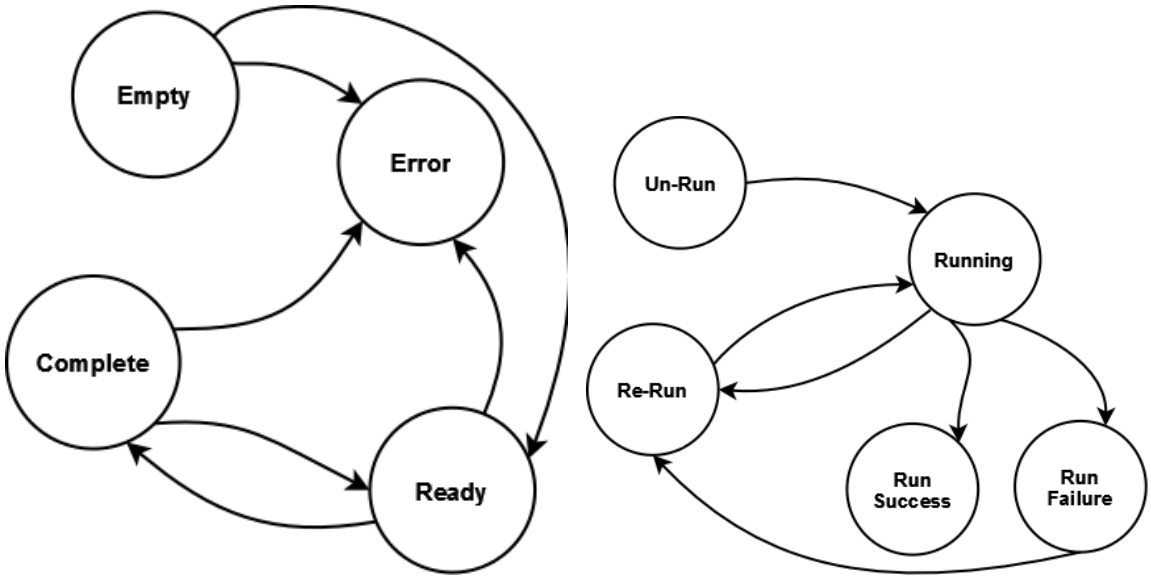
C. Outerator Design

The data produced by a simulation gets recorded when the Case object containing the information for that simulation is passed to the recordCase() method of an Outerator.

To make the data easily accessible once it has been stored the Comma Separated Value format was reused. Once again, the header row represented the attribute name being tracked and each row contained the value from that output attribute for that simulation. Additionally, it was useful to have the appropriate input information available in the same output, so the Outerator will store that data along with the case outputs in each row.

D. Conductor Design

Besides communications between all the other ROSE objects, the Conductor handles all the asynchronous simulation scheduling requirements that come along with concurrent executions. To accomplish this the Conductor utilizes a series of state tables to determine the proper action to take for any given server-Ready_CB() from an Operator. Each callback provides a pointer to the server that triggered it. The Con-



(a) State Machine for Servers (b) State Machine for Cases

Figure 4. State machines for the server and case tracking

ductor then checks which state that server is currently in. Using these state machines, a Conductor tracks the progress of an arbitrary number of Cases executing on an arbitrary number of servers. If the number of available servers just happens to be one, then the system will execute each Case successively. However, should more than one server become available the very same Conductor can orchestrate the execution of many Cases simultaneously, assuming that none of the calls in the Operator API are blocking. The Driver does not know, nor does it matter, a priori how many servers will be available for a given execution. The issue of concurrency then lies completely within the implementation of the Operator.

There are four possible states, Empty, Ready, Complete, and Error. All servers enter the system in the Empty state, and move between states according to the rules of the state machine, show in in Figure 4(a). The conductor will try to load a model into an empty server and initialize it, utilizing the ROSE API. If that action succeeds, the server moves to the Ready state. For a server in the Ready state, the Conductor set some action executing on that server and then the server moves to the Complete state. When a callback is received for a server in the complete state the Conductor will perform the necessary steps to re-initialize

the model and return the server to the ready state. Also, at any point if there is some sort of communication error and the server is unavailable the Server can be moved into the Error State. From the Error state, it can be moved back to the empty state if communications can be re-established.

Each time a ServerReady_CB() is made for a server the Conductor refers to a table that provides it with the particular Case running on that server. Then it utilizes that cases state machine, shown in Figure 4(b), to determine the proper action to be taken. Cases have five possible states: Un-Run, Running, Run Success, Run Failure, Re-Run. All cases start in the Un-Run state, when a server is free a case can be loaded into it, and moved to the running state. From there the case can either move to the Run Success or Run-Failure states. Also, if some kind of communication error happens not related to the model itself, the case can be moved to the Re-Run state. A case could also move the Re-Run state from the Run-Failure if necessary.

Having a separate state for the Re-Run cases was not strictly necessary, as the Un-Run state could have been used. But having a separate state for the Re-Run cases allowed for some special scheduling routines to be used to limit the impact of large numbers of Re-Run cases on execution time.

E. Operator Design

The Operator serves as the communication layer between the Conductor and the computational resources available to it. By design, the implementation of the conductor determines the amount of concurrent execution a Driver will handle. By swapping out one Operator for another, a single Driver could possibly utilize vastly different computational resources. For this implementation a simple single server Operator was written. A second Operator was also written which utalized the Python module Parallel Python to enable concurrent execution of multiple cases simultaneously if sufficicent resources are available.

To achieve the greatest level of code portability, an extra layer of abstraction was used to allow a single Operator to communicate with a wider variety of models. The Conductor communicates with the Operator via the API. This operator in turn defines its own API, which allows it to communicate with any model class implementing that API. Constructing the Operator in this manner allowed it to communicate with both native Python models, as well as NPSS models wrapped in a special class which conformed to the model API.

The advantage of this approach is obvious when considering the generation of new, more powerful, concurrent Operators. In this situation using the model API, none of the work put into interfacing external codes with Python needs to be wasted. In this case, the object which wraps NPSS and implements the model API will just as easily communicate with any Operator that utilizes the same API.

1. Model Interface

The following documents the model interface mensioned above. This interface provides a lower level API between the Operator and the model itself. Models written using Python could be implemented with a given model object while models written in Matlab or some other modeling code could use their own model objects.

A Model provides the following methods:

- set(key,value): This method is used to set attributes of the model. The method is passed a key, pointing to a specific model attribute, and a value to set the attribute to. It is expected to return success or failure.
- get(key): This method is used to retrieve the value of a model attribute identified by the given key. It is expected to return the specified value, or a failure.
- initialize(): This method is called only once, immediately after model instantiation, for any given model instance. It is here that any initialization a model requires can occur. It returns a success or failure.
- exec(): This method is called when a model is ready to execute. It will execute a model. It is expected to return nothing
- setup(): This method is called each time just prior to the run() method, but after any set() calls are made to specify case inputs. It is designed to allow the modeler to create model reactions to inputs,

as well as ensure that a model is prepared to execute properly before a run() is called. It is expected to return success or failure of the initiation

- succeeded(): This method is called after run() returns. It is used to test the model for successful completion. It returns a success or failure.
- cleanUp(): This method is used to perform any actions necessary to prepare a model for removal. This includes releasing of any system resources the model holds and no longer needs.

V. Results

The ROSE framework discussed above was used to execute over 20,000 cases on a mixed flow turbojet engine model. The Driver reported the data in the CSV format defined the by the outerator used. Response surface equations (RSE's) were generated using the R statistical analysis toolkit.[8] R is capable of reading in the CSV format output by the Driver directly. The RSE's provided a very valuable tool for studying the design space of the engine model. Because they are so fast to execute, being simple semi-log expressions, the performance of any group of points in the design space were instantly accessible with a reasonable degree of accuracy.

A. Engine Model Inputs and Outputs

The engine model provided six parametric inputs, however only two of the inputs were design inputs. The remaining four inputs were operational settings which determined the flight condition and nozzle setting of the engine.

Though the model contained only two degrees of freedom with respect to the design variables, the remaining four operational variables represented a massive operational envelope for any single design configuration.

There were six RSE's generated: gross thrust, ram drag, fuel burn rate, nozzle exit static temperature, nozzle exit static pressure, and total engine weight. Resulting calculated values included net thrust, and thrust specific fuel consumption.

1. *Inputs*

- top-of-climb Thrust: (design variable:$TOCThrust$) specified in lbs force, the engine gets designed to produce the specified amount of thrust at the top of climb flight condition.
- fan pressure ratio: (design variable:$FanPRdes$) specified as a ratio of the entrance pressure to the exit pressure, the fan was designed to produce the specified pressure ratio at the top of climb flight condition.
- nozzle underexpansion factor: (operational variable:$Knoz$) specified as a fraction less than or equal to one, this setting determined the amount of under expansion the variable nozzle area would achieve.
- altitude: (operational variable:alt) specified in feet, this is the flight altitude the engine is operating at
- mach: (operation variable:MN) specified as Mach number, this is the flight Mach number the engine is operating at
- power code: (operational variable:PC) specified as a number between 10 and 50, this represents the power setting of the engine with 50 being the maximum thrust level of the engine at the current flight condition.

2. *Outputs*

gross thrust:

$$\log(F_{gross}) = 8.542 - alt*.00004560 + MN*1.395 + PC*.01845 + TOCThrust*.0001618 - FanPRdes*.3889 + Knoz*.5299 \quad (1)$$

ram drag:

$$\log(D_{ram}) = 6.901 - alt * .0000441 + \sqrt{MN} * 4.163 + TOCThrust * .0001618 - FanPRdes * .6189 \quad (2)$$

fuel burn rate:

$$\log(W_{fuel}) = 2.932 + \log(Fg) * .5304 - alt * .00002462 + MN * .5397 + PC * .01595 + TOCThrust * .00007597 \quad (3)$$

net thrust:

$$F_{net} = F_{gross} - D_{ram} \quad (4)$$

nozzle exit static temperature:

$$\log(nozzleExitT_s) = 5.294 - alt * .000005553 + MN * .008531 + PC * .004705 + FanPRdes * .1603 + Knoz * .8210 \quad (5)$$

nozzle exit static pressure:

$$\log(nozzleExitP_s) = .8419 - alt * .00004427 - FanPRdes * .008796 + Knoz * 1.945 \quad (6)$$

thust specific fuel consumption:

$$TSFC = W_{fuel}/F_{net} \quad (7)$$

weight:

$$\log(weight) = 7.826 + 3.506/FanPRdes^2 + TOCThrust * .0001454 \quad (8)$$

nozzle weight:

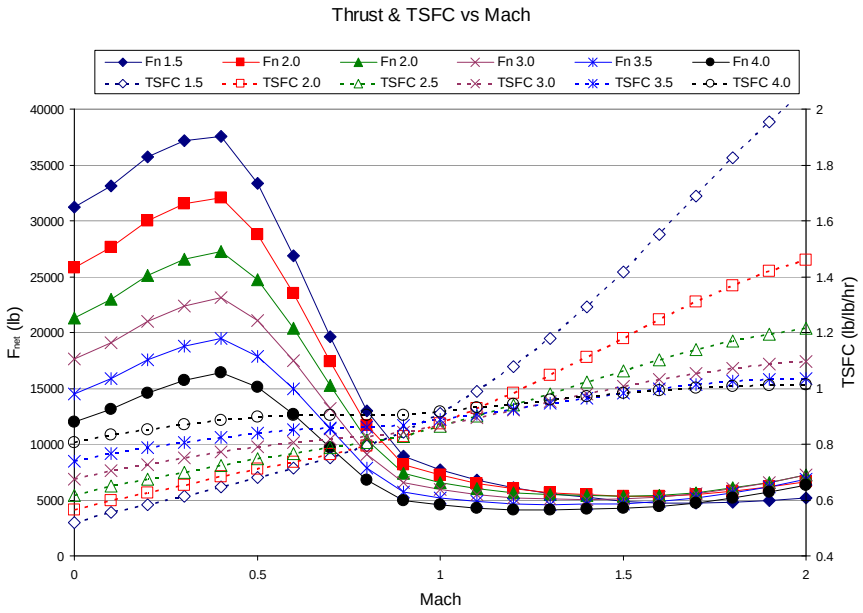
$$W_{nozzle} = \sqrt{weight}^{FanPRdes/2} \quad (9)$$

3. *Analysis*

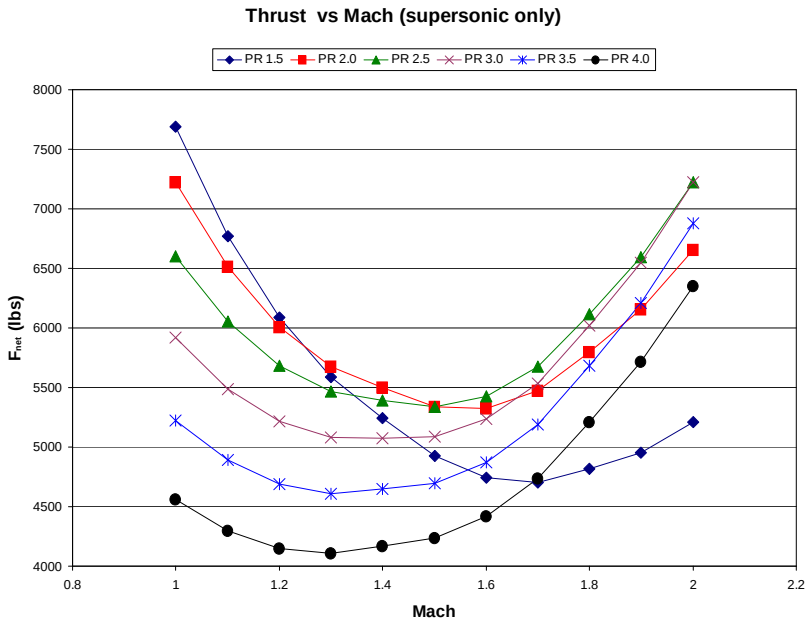
The RSE's provide a rapid and simple method of examining the effect of design changes on engine performance. Figure 5(a) shows the change in thrust lapse and fuel consumption for a series of different fan pressure ratios. As Mach number increases, the thrust of each of the engines drops. This data is examined with an assumed flight profile presented in figure 6.

As the engines approach the top of climb flight condition, their thrusts all converge at around 6000 lbs. This behavior of the RSE's confirms that they exhibit the proper thrust trends, as the engine data used to generate the RSE's was all sized at the top of climb condition. However, the actual accuracy of the RSE's can be better examined in figure 5(b) where only the super sonic part of the flight profile is shown. Here it is clear that the engines do not all produce exactly 6000 lbs of thrust. Although the trends in the data are correct, the accuracy of any single data point may as much as 20% off, such as the data for fan pressure ratio of 1.5.

The utility of RSE's in engineering design and optimization problems is well established.[9] What they can provide in terms of computational speed can often be at the expense of accuracy. The accuracy of the RSE's needed to calculate net thrust are further considered in figures 5(a) through 9. Figure 8(a) Shows how well the gross thrust was prediced by the F_{gross} RSE. The RSE performs very well at the higher altitudes, but is much less accurate at lower altitudes. Figure 8(b) shows the accuracy of the D_{ram} RSE. This RSE is much more accurate throughout the whole altitude and mach number regime considered. These two RSE's together combine to predict the net thrust of any given engine at any given flight condition. Hence, there errors would be additive for net thrust. The residual plot in figure 9(b) shows that the errors for gross thrust predictions are well distributed across all thrust levels. Similarly figure 9(a) indicates that the RSE has better accuracy at the lower gross thrust levels than it does at higher ones.



(a) Full Mach Range



(b) Supersonic Mach Range

Figure 5. Thrust vs flight Mach number is shown on the right axis. TSFC vs flight Mach number is shown on the left axis.

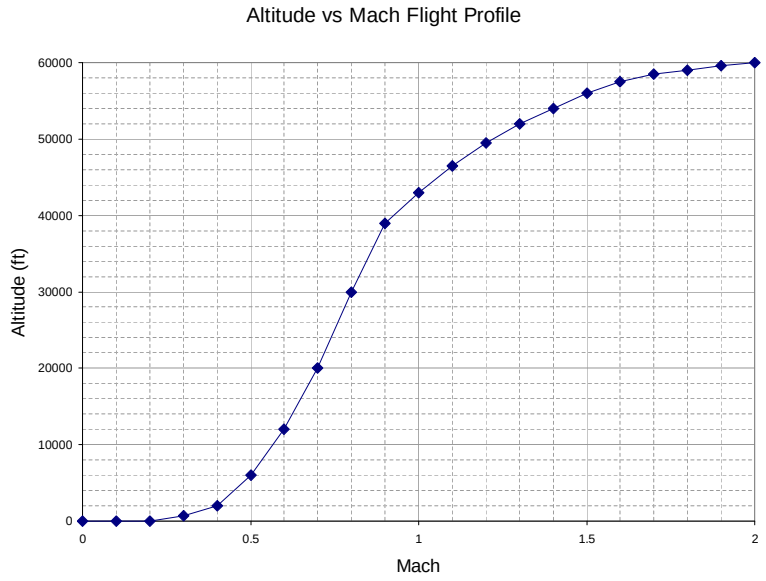


Figure 6. Assumed Mach vs Altitude flight profile used to generate performance data over the flight envelope.

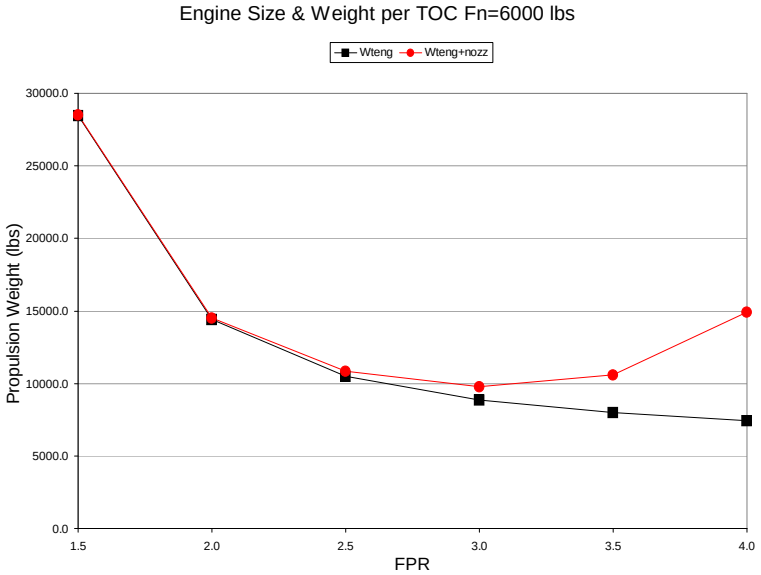
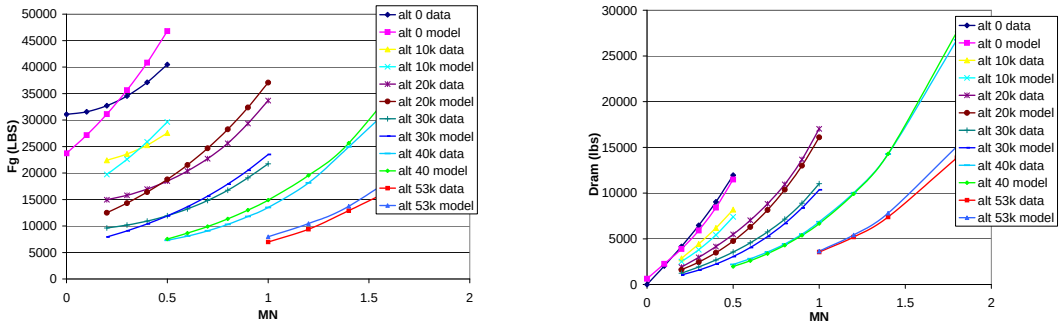
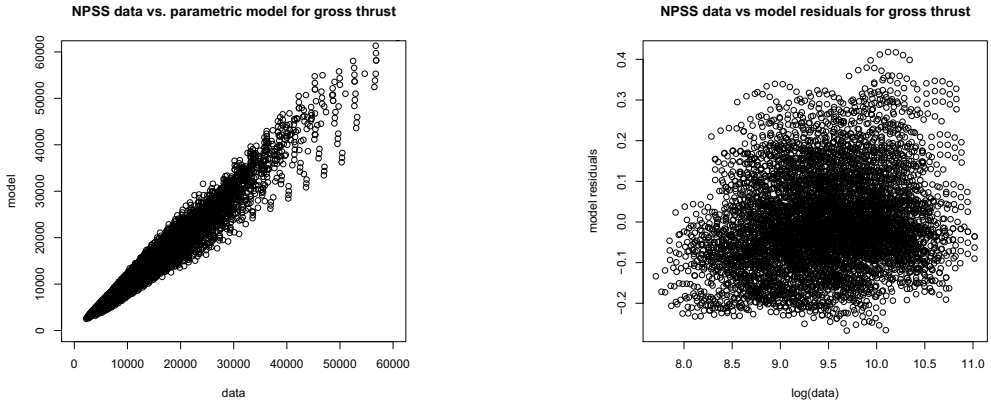


Figure 7. Engine weight with and without a noise supressing nozzle



(a) Gross Thrust (b) Ram Drag

Figure 8. Comparisons for gross thrust and ram drag between the RSE predictions and NPSS model data



(a) 45 degree (b) Residual

Figure 9. A 45 degree plot and a residual error plot for the gross thrust response surface

VI. Conclusion

Using Python, a ROSE implementation was generated which followed a strict adherence to the ROSE API defined here. The result was a ROSE Driver that could execute an arbitrary DOE on an arbitrary model. Using the parallel-python package concurrent execution of individual cases was possible. This Driver was able to efficiently execute over 20,000 simulations of the computationally expensive NPSS model and provide the results in a manner which allowed efficient generation of the RSE's.

The creation of two distinct Operators displayed the power of adhearing to the ROSE API. Each operator interfaced with the other Driver objects only through the API, and hence to the rest of the Driver did not change. However, one Operator was utalizing only a single server and offered no concurrent capability. The other Operator enabled conccurent execution, and hence offered a more powerful computation tool for large case sets. Similarly any Driver object coudl be replaced in the system to change the behavior of the driver. New Iterators and Outerators can be made and better Conductors can be created.

Further use of the powerful Python modules available in the open source community could quickly result in Optimizers, and statistical analysis Drivers which could provide capability similar to the Design Analysis Kit for Optimization and Terascale Applications (DAKOTA),[1] but with greater flexibility and extensibility.

References

[1]Mike Eldred. *DAKOTA Developer's Manual*. Sandia National Laboratories.

[2]M.S. Eldred and D.M. Dunlavy. Formulations for surrogate-based optimization with data fit, multifidelity, and reduced-order models. AIAA, 2006.

[3]Anthony A. Giunta. Use of data sampling, surrogate models, and numerical optimization in engineering design. AIAA, 2002.

[4]John Hunter and Darren Dale. *The Matplotlib Users Guide*, May 2007.

[5]NASA Glenn Research Center. *NPSS User Guide*, NPSS_1.6.4 REV:AC edition, August 2007.

[6]Travis Oliphan. *Numpy Home Page*, 2007.

[7]Python Software Foundation. *Beginner's Guide to Python, Python for Programmers*, August 2007.

[8]R Deelopment Core Team. *The R Manuals*, 2008.

[9]Gerhard Venter, Raphael T. Haftka, and James H. Starnes Jr. Construction of response surface approximations for design optimization. *AIAA Journal*, 36(12), December 1998.

Appendix

A. Iterator and Case

Code Listing 1. Iterator class implementation: caseRowIterator.py

```
import case

class caseRowIterator(object):
    """ The ROSE Iterator retrieves sets of input values that can be given to a
    model/simulation.  The Iterator generates files that are two lines.  The
    first line is a header that conatins variable names, while the second
    contains the values for those variables.  The Iterator assumes that the
    attributes stay the same each time you ask it to retrieve a case."""

    def __init__(self,inputFileName="ROSE.in",delimiter=','):
        self.delimiter = delimiter
        self.inputFileName = inputFileName
        self.initialize()

    def initialize(self):
        try:
            # open the input file for reading and easy searching
            self.f = open(self.inputFileName,"r")
            line = self.f.readline().strip("\n")
            #print line
            self.attributeList = line.split(self.delimiter)
        except IOError:
            #for now we just raise the error
            #TODO: Implement ROSE error classes and error handling
            raise

    def getNextCase(self):
        """ Creates a case, and sets the input variables with the next unused line of
        input values.  It returns a pointer to this case.  If all lines of input
        variables have been used, and empty pointer is returned """

        line = self.f.readline().strip("\n")
        if line != "":
            casePointer = case.case()
            valuelist = [float(x) for x in line.split(",")]
            for key,value in zip(self.attributeList,valuelist):
            casePointer.setInput(key,value)
            return casePointer
        else:
            return


    def cleanUp(self):
        self.f.close()
```

Code Listing 2. Case class implementation:case.py

```
class case(object):
    def __init__(self,name='not_named',inputs={},outputs={}):
        self.success = 0
        self.name = name
        self.inputs = []
```

```
        self.outputs = []

        [self.setInput(key,inputs[key]) for key in inputs]
        [self.setOutput(key,outputs[key]) for key in outputs]

        self.setInput("outputs",[key for key in outputs])


    def setInput(self,key,value=None):
      inp = [x for x in self.inputs if x.key==key]
      if  inp != []: #if the input exists, just set the value
        inp[0].value = value
        else: # create the input
        self.inputs.append(caseInput(key,value))

    def getInput(self,key):
      return [x.value for x in self.inputs if x.key == key][0]

    def getInputs(self):
      return [[str(x.key),x.value] for x in self.inputs]

    def setOutput(self,key,value=None):
      out = [x for x in self.outputs if x.key == key]
      if out !=[]:
          out[0].value = value
      else:
          self.outputs.append(caseOutput(key,value))
          #track names of all outputs as an attribute
          [x.value.append(key) for x in self.inputs if x.key == 'outputs'][0]

    def getOutput(self,key):
      return [x.value for x in self.outputs if x.key == key][0]

    def getOutputs(self):
      return [[x.key,x.value] for x in self.outputs]

class caseInput(object):
    def __init__(self,key,value):
      self.key = key
      self.value = value

class caseOutput(object):
    def __init__(self,key,value=None):
      self.key = key
      self.value = value
```

B. Outerators

Code Listing 3. Comma Separated File Outerator implementation: cseRowOuterator.py

```
import time
import case

class caseRowOuterator(object):
    """ This ROSE Outerator records case output data in a flat file format. The
    first line of the output file generated is a header containing the names of
    each of the model attributes that were recorded. The Outerator assumes that
    all cases record the same list of model attributes. It automatically records
```

```
    the case inputs first and the case outputs next on the same row for a given
    simulation."""

    def __init__(self,outputFileName,delimiter=','):
        self.delimiter = delimiter
        self.outputFileName = outputFileName
        self.headerWritten = False
        self.initialize()

    def initialize(self):
        #open the output file for writing and start the blank string
        self.f = open(self.outputFileName,"w")
        print >> self.f,"Created on " + time.strftime("%A %B %d, %Y")

    def recordCase(self,casePointer):
        outstr = ""

        inputs = casePointer.getInputs()
        outputs = casePointer.getOutputs()

        if(not self.headerWritten):
        for (key,value) in inputs:
            outstr += str(key) + self.delimiter
        for (key,value) in outputs:
            outstr += str(key) + self.delimiter
        print >> self.f,outstr.strip(self.delimiter)
        outstr = ""
        self.headerWritten = True

        for (key,value) in inputs:
            outstr += str(value) + self.delimiter
        for (key,value) in outputs:
            outstr += str(value) + self.delimiter
        print >> self.f, outstr.strip(self.delimiter)

        def cleanUp(self):
        self.f.close()
```

Code Listing 4. Database based Outerator implementation: dbOuterator.py

```
from sqlalchemy import *
from sqlalchemy.orm import *
from case import case, caseInput, caseOutput
import types
import os

class dbmanager(object):
    def __init__(self):
        self.engines = {}
        self.metadatas = {}
        self.case_tables = {}
        self.caseinput_tables = {}
        self.caseoutput_tables = {}
        self.sessionmakers = {}

    def getengine(self,database):
        try:
            return self.engines[database]

    except KeyError:
```

```
        self.engines[database] = create_engine('sqlite:///'+database,echo=False)
        return self.getengine(database)

    def getmetadata(self,database):
        try:
            return self.metadatas[database]

        except KeyError:
            engine = self.getengine(database)
            self.metadatas[database] = MetaData()
            #associate the engine to be the default for the metadata
            self.metadatas[database].bind = engine
            return self.metadatas[database]

    def gettables(self,database):
        try:
            return {"case_table":self.case_tables[database],
            "case_inputs":self.caseinput_tables[database],
            "case_outputs":self.caseoutput_tables[database]}

        except KeyError:
            engine = self.getengine(database)
            metadata = self.getmetadata(database)
            #create case Table
            self.case_tables[database] = Table('cases', metadata,
                Column('id',Integer,primary_key = True),
                Column('state',Text),
                )

        #create caseInputTable
        self.caseinput_tables[database] = Table('caseInputs',metadata,
            Column('id',Integer,primary_key = True),
            Column('key',Text),
            Column('value',PickleType),
            Column('case_id',Integer, ForeignKey('cases.id')),
            #links rows a case entry
            # prevents duplicate keys for a given case
            UniqueConstraint('case_id', 'key', name='uix_1')
            )

        #create caseOutputTable
        self.caseoutput_tables[database] = Table('caseOutputs',metadata,
            Column('id',Integer,primary_key = True),
            Column('key',Text),
            Column('value',PickleType),
            Column('case_id',Integer,ForeignKey('cases.id')),
            #links rows a case
            entry
            # prevents duplicate keys for a given case
            UniqueConstraint('case_id', 'key', name='uix_1')
            )

        #Actually create the tables to reflect the objects created above
        metadata.create_all()

         #Link the case object to the case_table
         #apply the links between the case_table and the other two tables above.
         mapper(case,self.case_tables[database],properties={
             'inputs':relation(caseInput,backref='case', cascade="all, delete,
             delete-orphan"),
```

```
            'outputs':relation(caseOutput,backref='case', cascade="all, delete,
            delete-orphan")}
            )
        #Link the caseInput to the caseinput_table
        mapper(caseInput,self.caseinput_tables[database])
        #link the caseOutput to the caseoutput_table
        mapper(caseOutput,self.caseoutput_tables[database])

        return self.gettables(database)

    def getsession(self,database):
        try:
            maker = self.sessionmakers[database]
            return maker()

        except KeyError:
            engine = self.getengine(database)
            metadata = self.getmetadata(database)
            tables = self.gettables(database)
            self.sessionmakers[database] =
            sessionmaker(bind=engine,autoflush=True,transactional=True)
            return self.getsession(database)



#create the manager
manager = dbmanager()

class dbbase(object):
    def __init__(self,dbname,accessFlag=False):
        self.session = manager.getsession(dbname)
        self.database = dbname
        self.tables = manager.gettables(dbname)
        self.session = manager.getsession(dbname)

    def grabdata(self,returnType,*conditions):
        """grabs data from the simulation results based on the conditions
        provided as (key,value) tuples"""
        #define the query with the approriate joins
        q = self.session.query(case).join('inputs').join('outputs')
        #filter the results based on
        for condition in conditions:
            # assumes that all numbers in the database will be floats and
            corrects for it
            if(isinstance(condition[1],int)): value=float(condition[1])
            q =
            q.filter(or_(and_(caseInput.key==condition[0],caseInput.value==value)
            ,and_(caseOutput.key==condition[0],caseOutput.value==value)))
            returnlist = list()
            returndict = dict()
            if(returnType == list):
                for c in q.all():
                    returnlist.append(tuple([i.value for i in c.inputs]+[i.value
                    for i in c.outputs]))

                return tuple(returnlist)

            elif(returnType == dict):
                for c in q.all():
```

```
                    returndict[c.id] = dict([(str(i.key),i.value) for i in
                    c.inputs+c.outputs])

                return returndict

            else:
                raise TypeError,"Improper return type, " + str(returnType) +
                "requested, only list or dict supported"


    def exportcsv(self,filename):
        """prints the contents of the database to a csv file with specified
        name.
        The function assumes that all cases have the same set of inputs and
        outputs."""
        f = open(filename,'w')
        #get all the cases in the db
        q = self.session.query(case)
        #Create header row
        header = "id,"
        ins = outs = ""
        for i in q.first().inputs:
            ins += i.key + ","

        for o in q.first().outputs:
            outs += o.key + ","

        header += (ins + outs).strip(',') + "\n"

        #write the case data
        data = ""
        for c in q.all():
            line = str(c.id) + ","
            for i in c.inputs:
                line += str(i.value) + ','
            for o in c.outputs:
                line += str(o.value) + ','
            data += line.strip(',') + '\n'
        print >> f,header + data
        f.close()

class dbOuterator(dbbase):
    def __init__(self,dbname='simulationOutput'):
        """ Initialzies the outerator. Takes, as an argument, the name of the
        file database to be used to store data. This function will erase any
        data
        previously held in the database during initialization."""
        #make sure database is clear of any old data
        if(os.path.exists(dbname)):
            os.remove(dbname)

        #initialize the base class
        dbbase.__init__(self,dbname)

    def recordCase(self,casePointer):
        self.session.save_or_update(casePointer)
        self.session.commit()

    class dbAccess(dbbase):
        def __init__(self,dbname='simulationOutput'):
```

```
            #initialize the base class
            dbbase.__init__(self,dbname)
```

C. Conductor

Code Listing 5. Conductor implementation: conductor.py

```
class serverError(Exception):
    def __init__(self,value):
        self.value = value

    def __str__(self):
        return str(self.value)

class stateError(Exception):
    def __init__(self,value):
        self.value = value

    def __str__(self):
        return str(self.value)

class stateTracker(object):

    def __init__(self,initialState = 1):
        self.entities = []
        self.states = []
        self.initialState = initialState

  def getState(self,entity):
      try:
          i = self.entities.index(entity)
          return self.states[i]

      except ValueError:
          self.entities.append(entity)
          self.states.append(self.initialState)
          return self.getState(entity)

    def setState(self,entity,state):
      try:
          i = self.entities.index(entity)
          self.states[i] = state

      except ValueError:
          raise stateError("Warning:  "+str(entity)+" did not exist in state\
          table")

      except stateError:
          raise

    def deleteEntity(self,entity):
      try:
          i = self.entities.index(entity)
          self.entities.remove(entity)
          self.states.pop([i])

      except ValueError:
```

```
            raise stateError("Warning: "+str(enity)+" did not exist in state\
            table")

        except stateError:
            raise

    def listEntitiesWithState(self,state):
        entitiesWithState = [x for x,y in zip(self.entities,self.states) if y \
        == state]
        return entitiesWithState


class conductor(object):
    """ put some documentation here """

    def __init__(self,outputKeys=[]):
        self.serverTracker = stateTracker("empty")
        self.caseTracker = stateTracker("unrun")
        self.caseServerTracker = stateTracker("none")
        self.initialize()
        self.iterator = None
        self.outerator = None
        self.operator = None

        self.outputKeys = outputKeys
        self.reloadmodel = True

    def initialize(self):
        return

    def setOperator(self,operatorPointer):
        self.operator = operatorPointer

    def setIterator(self,iteratorPointer):
        self.iterator = iteratorPointer

    def setOuterator(self,outeratorPointer):
        self.outerator = outeratorPointer

  def serverready_CB(self,serverPointer):
      serverState = self.serverTracker.getState(serverPointer)
      self.caseServerTracker.getState(serverPointer)
      if serverState == "empty":
          # add error handling
          try:
              self.operator.loadModel(serverPointer)
              self.serverTracker.setState(serverPointer,"ready")
              return True

          except serverError:
              self.serverTracker.setState(serverPointer,"error")
              return True

          except:
              raise
              # in the operator class, we should come up with
              #a class called serverError

      elif serverState == "ready":
          # checks if there are cases that need reruns that need to be rerun
```

```
        if len(self.caseTracker.listEntitiesWithState("rerun")) > 0:
            casePointer = self.caseTracker.listEntitiesWithState("rerun")[0]
            try:
                # setup case and run case
                self.caseTracker.setState(casePointer,"running")
                self.caseServerTracker.setState(serverPointer,casePointer)
                # run the model

                inputs=casePointer.getInputs()
                for (key,value) in inputs:
                    self.operator.modelSet(serverPointer,key,value)
                self.operator.modelExecute(serverPointer)
                self.serverTracker.setState(serverPointer,"complete")
                return True

            except serverError:
                self.serverTracker.setState(serverPointer,"error")
                self.caseTracker.setState(casePointer,"rerun")
                return True

            except:
                raise

        # check if there are any cases that are unrun
        elif len(self.caseTracker.listEntitiesWithState("unrun")) > 0:
            casePointer = self.caseTracker.listEntitiesWithState("unrun")[0]
            try:
                # setup and run case
                self.caseTracker.setState(casePointer,"running")
                self.caseServerTracker.setState(serverPointer,casePointer)
                # run the model

              inputs=casePointer.getInputs()
              for (key,value) in inputs:
                  self.operator.modelSet(serverPointer,key,value)

                self.operator.modelExecute(serverPointer)
                self.serverTracker.setState(serverPointer,"complete")
                return True

            except serverError:
                self.serverTracker.setState(serverPointer,"error")
                self.caseTracker.setState(casePointer,"rerun")
                return True

            except:
               raise

        else:
            try:
                casePointer = self.iterator.getNextCase()
                if casePointer != None:
                    self.caseServerTracker.setState(serverPointer,casePointer)
                    self.caseTracker.getState(casePointer)
                    self.caseTracker.listEntitiesWithState("unrun")
                    # setup and run first case in the list
                    self.caseTracker.setState(casePointer,"running")
                    # run the model
                    for (key,value) in casePointer.getInputs():
                        #print casePointer.getInputs()
```

```
                    self.operator.modelSet(serverPointer,key,value)

                self.operator.modelExecute(serverPointer)
                self.serverTracker.setState(serverPointer,"complete")
                return True

            else: # There are no more cases to run
                return False

        except serverError:
            self.serverTracker.setState(serverPointer,"error")
            self.caseTracker.setState(casePointer,"rerun")
            return True

        except:
            raise


elif serverState == "complete":
    try:
      i = self.caseServerTracker.entities.index(serverPointer)
      casePointer = self.caseServerTracker.states[i]
      if self.operator.modelSuccess(serverPointer):
          self.caseTracker.setState(casePointer,"runsuccess")
          #grab the data from the model
          for key in self.outputKeys:
              casePointer.setOutput(key,
                            self.operator.modelGet(serverPointer,key))

          #record the data
          self.outerator.recordCase(casePointer)
          if(self.reloadmodel):
              self.operator.modelCleanUp(serverPointer)
              self.operator.loadModel(serverPointer)

          self.serverTracker.setState(serverPointer,"ready")
          return True

      else:
          print "FAILED"
          self.caseTracker.setState(casePointer,"runfail")
          self.operator.loadModel(serverPointer)
          self.serverTracker.setState(serverPointer,"ready")
          return True

    # handle server error separately
    except serverError:
        return True

    except:
        raise
        # if there is a server error server to state 4 and case to state 4
        # if there was another error just raise the error

elif serverState == "error":
    try:
        self.operator.loadModel(serverPointer)
        self.serverTracker.setState(serverPointer,"ready")
        return True
```

```
        except serverError:
            return True
```

D. Operator

Code Listing 6. Operator implementation: operator.py

```
class pyOperator(object):

    def __init__(self,modelclass):
        #pointer to the model to be used for analysis.
        self.modelclass = modelclass

    def initialize(self,serverPointer):
        pass

    def setConductor(self,pointer):
        self.conductorPointer = pointer;

    def start(self):
        #passing a None pointer because this is a local operator
        while(self.conductorPointer.serverready_CB(None)):
          pass

    def cleanUp():
        pass

    def loadModel(self,serverPointer):
        self.runningmodel = self.modelclass()

    def modelSet(self,serverPointer,key,value):
        setattr(self.runningmodel,key,value)

    def modelGet(self,serverPointer,key):
        return getattr(self.runningmodel,key)

    def modelExecute(self,serverPointer):
        self.runningmodel.execute()

    def modelSuccess(self,serverPointer):
        return self.runningmodel.succeeded()

    def modelCleanUp(self,serverPointer):
        return self.runningmodel.cleanup()
```

E. Model

Code Listing 7. Native python model implementation: model.py

```
class model(object):
    def __init__(self):
        pass

    def execute(self):
        self.setup()
```

```
        self.preexecute()
        self.calculate()
        self.postexecute()

    def setup(self):
        pass

    def preexecute(self):
        pass

    def execute(self):
        pass

    def postexecute(self):
        pass

    def succeeded(self):
        return True
```

Code Listing 8. NPSS model implementation: npssmodel.py

```
from npss import npss

class npssmodel(npss):
    def __init__(self,modelpath,npssmodelclass):
        """npss model class is a string with the name of the model class"""
        npss.__init__(self,['-I ' + x for x in modelpath])
        #add '-I' to each path location

        self.create(npssmodelclass,npssmodelclass,"model")
        self.setTop('model')
        self.init()
        self.create("","string1D","outputs")

    def execute(self):
        self.setup()
        self.run()

        rtn = {}
        for output in self.outputs:
            rtn[output] = getattr(self,output)
        return rtn
```

F. Driver

Code Listing 9. DOE driver implementation with csv outerator: DOEdriver.py

```
import baseClasses
import utils

class DOEdriver(object):

    def __init__(self,modelpath,modelClass,inputDOEFile,outputDataFile,
                                                        outputKeys=[]):
        self.inputDOEFile = utils.findinpath(inputDOEFile,modelpath)
        self.outputDataFile = outputDataFile
        self.iterator = baseClasses.caseRowIterator(self.inputDOEFile,",")
        #self.outerator = baseClasses.caseRowOuterator(outputDataFile," ")
```

```
        self.outerator = baseClasses.outerator(outputDataFile)

        self.conductor = baseClasses.conductor(outputKeys)
        self.operator = baseClasses.pyOperator(modelClass)

        self.conductor.setOuterator(self.outerator)
        self.conductor.setIterator(self.iterator)
        self.conductor.setOperator(self.operator)

        self.operator.setConductor(self.conductor)

        self.outputdata = outputKeys

    def execute(self):
        self.operator.start()
        return baseClasses.dbAccess(self.outputDataFile)
```

Code Listing 10. DOE driver implementation with database Outerator: dbDOEDriver.py

```
import baseClasses
import utils

class DOEdriver(object):

    def __init__(self,modelpath,modelClass,inputDOEFile,outputDataFile,
                                                        outputKeys=[]):
        self.inputDOEFile = utils.findinpath(inputDOEFile,modelpath)
        self.outputDataFile = outputDataFile
        self.iterator = baseClasses.caseRowIterator(self.inputDOEFile,",")
        #self.outerator = baseClasses.caseRowOuterator(outputDataFile," ")
        self.outerator = baseClasses.dbOuterator(outputDataFile)

        self.conductor = baseClasses.conductor(outputKeys)
        self.operator = baseClasses.pyOperator(modelClass)

        self.conductor.setOuterator(self.outerator)
        self.conductor.setIterator(self.iterator)
        self.conductor.setOperator(self.operator)

        self.operator.setConductor(self.conductor)

        self.outputdata = outputKeys

    def execute(self):
        self.operator.start()
        return baseClasses.dbAccess(self.outputDataFile)
```

G. Helper Code

Code Listing 11. Utilitiy function :utils.py

```
import os

def findinpath(fname, paths):
     """search for the given file in the given paths. If found, return the
     absolute path, else return None. paths can be in the form of a single
     string with delimited paths, or a list of paths"""
     if isinstance(paths,str):
```

```
        ppaths = paths.split(os.pathsep)
    else:
        ppaths = paths

    for p in ppaths:
        if os.path.exists(os.path.join(p,fname)):
            return os.path.abspath(os.path.join(p,fname))
    raise IOError, "No such file : '" + fname + "'"
```