OpenMDAO: An Open Source Framework for Multidisciplinary Analysis and Optimization

Justin Gray*
MDAO Branch, NASA Glenn Research Center, Cleveland, OH
Kenneth T. Moore† and Bret A. Naylor‡
DB Consulting Group, Inc., Cleveland, OH

This paper describes the progress made in the development of OpenMDAO, an open source framework for performing Multidisciplinary Analysis and Optimization (MDAO). NASA intends to use OpenMDAO to aid in the design of unconventional aircraft, but the general structure and methods may be applied to solve any number of engineering-related design problems. The framework currently supports data passing capabilities, and several example problems have been executed with it. Recent work has focused on enabling the creation of more complex MDAO strategies, such as collaborative optimization and surrogate modeling techniques. An example is presented that demonstrates an implementation of the surrogate model generation using Kriging surrogate models augmented with the expected improvement method.

Nomenclature

API	Aplication Programming Interface
DOE	Design of Experiments
EGO	Efficient Global Optimization
EI	Expected Improvement
GUI	Graphic User Interface
MDAO	MultiDisciplinary Analysis and Optimization
NOSA	NASA Open Source Agreement

I. Introduction

In order to survive the present and future economic and environmental challenges facing air transportation, aviation design may neet to expand its focus beyond today's conventional "wing-body-tail" aircraft and investigate more advanced and efficient vehicle concepts. The ability to design unconventional aircraft to improve mobility and air transport efficiency is of paramount importance and is expected to deliver additional benefits, such as reduced fleet fuel costs, minimized environmental impact, and improved mission capabilities.

However, traditional design methods rely on the accumulated experience of the designer and the availability of data and are not necessarily well suited for unconventional concepts. Test data and design experience are inherently limited for unconventional aircraft; thus there is a strong need to incorporate more sophisticated computational models and methods — in particular, physics-based Multidisciplinary Analysis and Optimization (MDAO) — into current and future design practices. To address this need, NASA's Fundamental Aeronautics Program has undertaken the development of MDAO technologies. One key capability is an integration framework that can encapsulate MDAO processes while enabling the incorporation of higher fidelity tools in the design process.

*Aerospace Engineer, MDAO Branch, Mail Stop 5-11, NASA Glenn Research Center, AIAA Member
†Senior Systems Engineer, MDAO Branch, Mail Stop 500-105, NASA Glenn Research Center, AIAA Senior Member
‡Senior Software Engineer, MDAO Branch, Mail Stop 500-105, NASA Glenn Research Center

After surveying the existing framework candidates and not finding any that were entirely consistent with the agency's long-term vision for MDAO, NASA decided to develop a new open source framework called OpenMDAO.[1] Being open source, this framework can attract a large pool of international users from industry, academia, and government groups who are interested in conceptual, preliminary, and detailed manufacturing design and analysis. Users of OpenMDAO will benefit from having access to the source code; however, open source licensing has other benefits as well. Users can contribute enhancements to the core framework, and engineering components can be contributed to the general community as well. The mission of OpenMDAO is to advance the science of MDAO by promoting collaboration and cooperation among industry, government, and academia through the use of open source tools.

II. The Open Source Development Process

The OpenMDAO project is managed through a central website, http://OpenMDAO.org. This website hosts the OpenMDAO discussion forums as well as the Software Configuration Management system. It also provides downloads for all official OpenMDAO release versions. Lastly, OpenMDAO.org is home to all of the OpenMDAO documentation. OpenMDAO.org was set up to encourage outside development and provide convenient access to support resources for users.

The OpenMDAO project uses the software configuation managment tool Trac.[2] OpenMDAO's Trac server can be accessed from the main project website at http://OpenMDAO.org. Registration is open to anyone with an email address. Since its initial development, the OpenMDAO project has followed formal methods of software verification and validation. The process began with the gathering of over 600 requirements, both functional and nonfunctional, spanning both the framework and the design analysis tools. The OpenMDAO Software Requirements Specification documents this work.[3] A group of discipline experts inspected and prioritized these requirements, and 207 were identified as having a direct impact on the framework. Each of these 207 requirements is cataloged in the project's software configuration management system on OpenMDAO.org. Every code submission is ultimately traceable to a requirement, a set of requirements, or a new feature request.

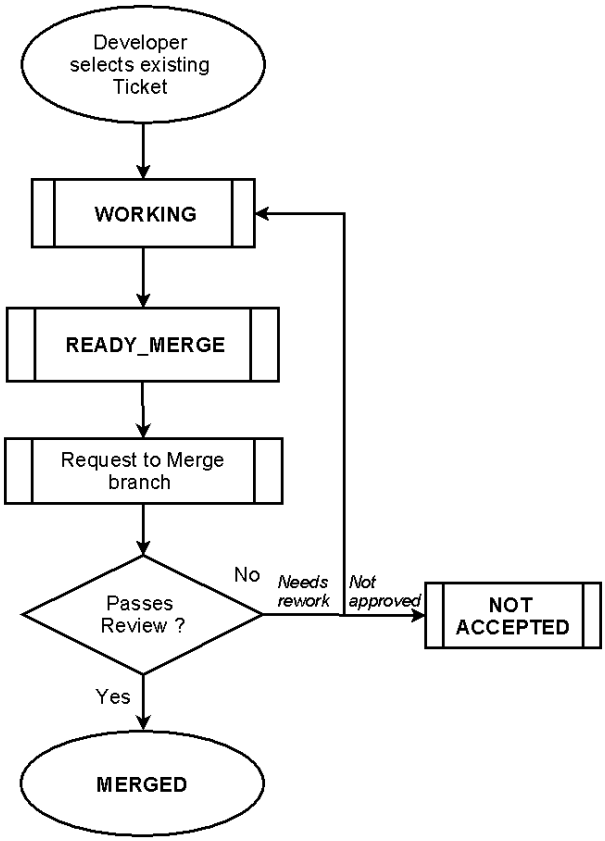

Figure 1: Lifecycle of an OpenMDAO ticket.

A. Code Contribution

External development of OpenMDAO capabilities falls under two categories:

1. OpenMDAO plugin development
2. Core framework development

In the first case, developing OpenMDAO plugins, new capabilities are added to OpenMDAO by means of external code modules which can be used within an analysis. These code modules are usually distributed by the people who develop them, separate from the OpenMDAO framework. Plugin development is essentially independent from OpenMDAO development since the code is not contributed back to the project.

In the second case, core framework development, new modules are added and old modules are updated within the framework itself. In general, these types of code contributions can be further divided into two sub-

categories: enhancements and bug fixes. In either case, the new code is contributed back to the OpenMDAO project and gets integrated into an official software release.

Users enter tickets into the Trac system, classifying them appropriately as enhancements or bugs. The OpenMDAO development team uses Trac to manage their own work, but Trac also manages code contributions from outside developers. Once a new ticket has been created, it progresses through a series of stages (states) in its lifecycle as it is worked on, completed, and integrated into the main code base. Figure 1 shows this lifecycle and illustrates how a ticket progresses. If an outside developer chooses to work on a particular ticket, he would use Trac to assign it to himself and then transition the ticket to a *working* state. This lets everyone else know that someone is working on the ticket and prevents duplication of effort. Once the work is completed, the developer assigned to the ticket moves it to the *ready_merge* state, and a merge request gets submitted on the OpenMDAO launchpad site. This lets the development team know that the work is complete and ready to be reviewed. Once the new code passes review, the development team merges it into the main code base; then the ticket is moved to the *merged* state. At this point, work on that particular ticket is considered complete.

B. Open Source Licensing

OpenMDAO is licensed under the NASA Open Source Agreement (NOSA). NOSA is a permissive open source license which is based on the Mozilla Public License (MPL). Permissive open source licenses, also called non-viral licenses, provide a good balance between protecting the integrity of the core framework and allowing for integration with commercial and proprietary plugins.[1]

Under NOSA, OpenMDAO can be freely distributed and used. If changes are made to any of the modules delivered as part of OpenMDAO, the resulting new modules retain the NOSA license. They must either be contributed back to the OpenMDAO development community or be provided as stand-alone modules licensed under NOSA. However, plugin development is not at all impacted by NOSA. Commercial plugins can be developed without needing to be licensed as open source software. Plugins will be distributed completely separate from OpenMDAO itself, so users would first acquire OpenMDAO and then acquire the specific plugin they want. Plugins are also an appropriate development method when users wish to develop their own capabilities and distribute them internally within their own company. These proprietary plugins also need not be licensed as open source projects.

C. Testing

Testing is an essential development tool for the OpenMDAO project. An extensive test suite of over 300 tests is included with every OpenMDAO distribution and development branch. Users can run the test suite to ensure that their OpenMDAO installation is functioning properly, and developers can use the tests to ensure that any changes they have made to the code did not have any unintended side effects. External code contributions must be accompanied by a set of new or updated unit tests before the external code will be approved for integration into the main development code base.

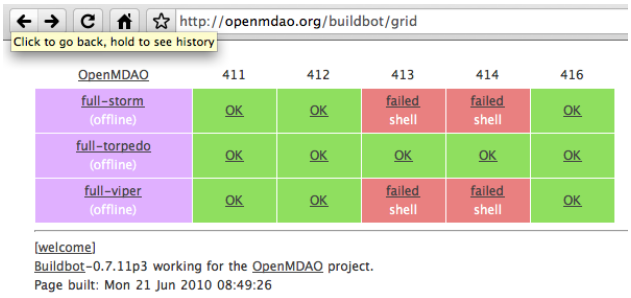

Figure 2: Testing results from the Buildbot tool.

The OpenMDAO framework supports multiple platforms — Linux, Windows, Mac OS X — so the test suite must be executed on each of these before changes can be committed to the code repository. However, for each of these platforms there are variations in the operating system series and version that may also require testing (e.g, Windows XP, Windows 7, Windows Server 2000, etc.). This adds up to quite a few systems where both the installation and execution of OpenMDAO must be tested, so test automation is necessary. A continuous integration tool called Buildbot[4] has proved to be very useful for this. Buildbot can create servers, known as build *slaves,* which can be commanded to install OpenMDAO, run the test suite, and post the results to another server hosting a build *master.* The build master monitors the code repository for new code then commits and launches tests on the build slaves as new code is introduced. A typical report is shown in Fig. 2.

The build reports include the full test results so that any errors can be identified. The project maintains three build slaves:

1. Linux RedHat Enterprise Level 5
2. Windows Server 2000
3. OSX 10.5 (Leopard)

BuildBot has the ability to support external build slaves. For example, if a user in the community has a system that runs an OS that is unrepresented in the current set of targets (e.g., Ubuntu), and this system is connected to the Internet, then the user could offer it as a new target build slave. Once this is done, any time the build master commands it, OpenMDAO will be installed and tested on the new target system. The report of this new build slave would be included in the overall OpenMDAO Buildbot report.

This capability allows for a very scalable automated testing scheme. It is reasonable that users would want to ensure that testing is done on environments which match what is available in their organizations. By setting up a new system configured to run a build slave, this can easily be accomplished. Buildbot results from the latest builds are publicly available at http://www.openmdao.org/buildbot.

D. Documentation and Training

It is absolutely critical to have strong documentation for an open source project, particularly when it is a complex engineering tool like a framework. In many cases, the most critical parts of the documentation are the simplest – the instructions that guide a new user through installation and use of the most commonly used functions. This kind of documentation is often the most overlooked by developers.[5] The OpenMDAO team has given a high priority to documentation and is committed to maintaining both an extensive User Guide and Developer Guide for the framework.

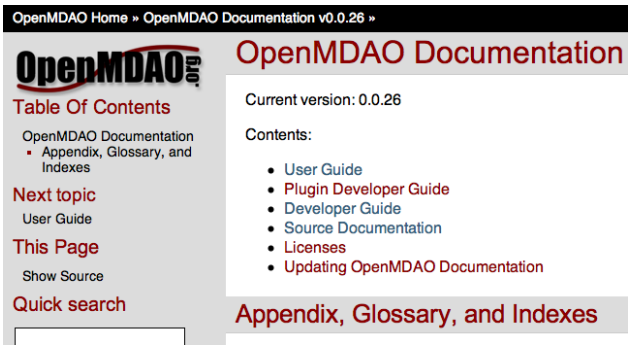

Figure 3: The OpenMDAO User Guide.

The User Guide (Fig. 3) contains documentation for anyone who wants to build and execute models in OpenMDAO, starting with instructions for installing the framework on the supported platforms. The team carefully considered several problems that could be used as examples in a set of tutorials that range from the simplest of problems – unconstrained optimization of a simple function over two design variables – to a much more complicated automotive drivetrain design problem. The tutorials will expand as more capability is added to the framework and as more problems are contributed by the community. The User Guide also contains a complete guide to OpenMDAO's scripting interface. Patterned after the Pylons[6] User Guide, this section details most of OpenMDAO's usable classes and functions, including those in the standard library, which are illustrated with inline examples. Additionally, the User Guide explains the process of wrapping external codes (written in C, Fortran, etc.).

The documentation text is written using a lightweight markup format called reStructuredText. For display, the reStructuredText is converted to HTML using the Sphinx[7] document generator. Sphinx can autodocument the source code using the docstrings of each Python function and class. Additionally, one of the most useful features of Sphinx is the support for different builders, such as the testdocs builder. With this builder (and a small amount of additional markup), all of the code snippets inside of the docs can be automatically tested when the unit tests are executed. This assures that the code examples presented in the docs will always be accurate and in sync with the framework development.

The latest version of the documentation, as well as the documentation for previous releases, is always available on the OpenMDAO website. The documentation is included in the software configuration management database, so it is tied directly to specific distributions and even specific development versions of OpenMDAO. Just as with the testing, when outside contributions are made to OpenMDAO, they should be accompanied with updated documentation.

III. Framework Capabilities

OpenMDAO version 0.1 has been released, and is avaiable for download from the project website. This is an alpha version of OpenMDAO and a detailed description of the current framework capabilities is presented below.

A. Infrastructure - Components, Drivers, Assemblies, and Sockets

In OpenMDAO, a problem is represented by a system of objects called components. These objects have input and output attributes and can perform some sort of calculation when they are executed. You can connect the inputs and outputs of one component to those of other components, allowing data to be passed between them. Figure 4 gives a conceptual view of what a simple Component might look like. This Component has two inputs (a, b) and one output (c). The calculation that it performs is to add the two inputs to produce the output. In a multidisciplinary analysis, a Component represents a unit of computational work in the

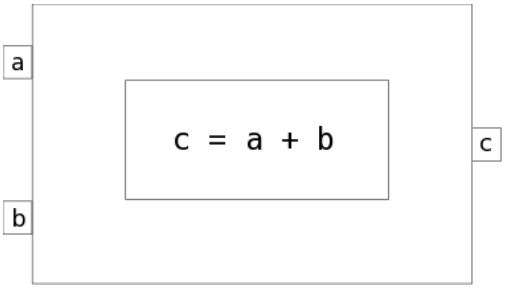

Figure 4: Conceptual view of a simple Component.

framework and could be considered to be a single discipline or analysis. Some componentswill be written purely in Python code. Native Python components are very easy to construct and provide the ability for users to rapidly implement new analysis tools. Other components may be comprised of a Python wrapper for a code written in another language, such as Fortran, C, or C++.

Wrapped components are often used when integrating legacy engineering tools into OpenMDAO as components. However, when computation efficiency is necessary, it makes sense to take a pure Python component and convert it to a wrapped component using a small amount of compiled code representing the core engineering. In the OpenMDAO User Guide, a piston engine model is created as a pure Python component. Then that analysis is converted into a compiled executable written in C and wrapped in Python to make a compiled component. The result of the complied component is a roughly 800% increase in computation speed.

OpenMDAO uses Drivers to perform the following iterative processes: optimization, convergence, and design of experiments (DOE). OpenMDAO currently provides a number of drivers:

1. CONMINdriver: A wrapper of the CONMIN optimizer, which performs a constrained minimization using a gradient-based algorithm
2. Genetic: A genetic algorithm optimizer based on the Pyevolve[8] package
3. DOEdriver: A DOE runner, which can execute any number of specific DOE types (Latin Hypercube, Full Factorial)
4. BroydenSolver: A Newton-Rhapson solver with Broyden approximation to the Jacobian
5. SingleCritEI: An implementation of the Expected Improvement algorithm for locating valuable new training points in adaptive sampling methods with a single objective

The Assembly contains groups of components and drivers that compose an analysis. Figure 5 depicts an assembly containing four components anda single driver. Both the Assembly and Driver classes are subclasses of Component. This means that they can have their own inputs and outputs and can be included in models with other components. This capability allows nested models to be created by having complex simulations contained within an assembly, i.e. a turbine enginesimulation in an Assembly which is used as a component in an aircraft simulation.

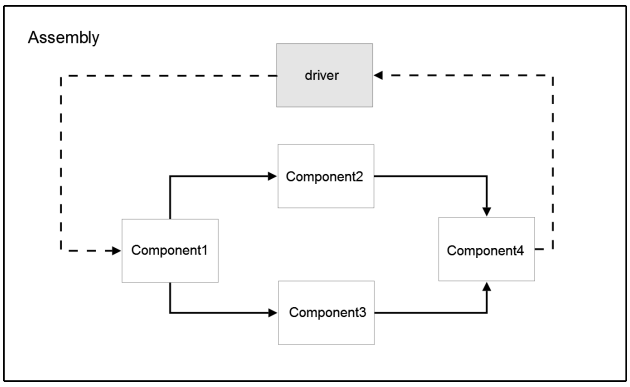

Figure 5: View of an Assembly with one Driver instance.

In OpenMDAO, sockets are placeholders where different objects can be substituted to modify the function of a Component. They are used by Components, Assemblies, and Drivers to provide a place where users can insert objects that meet a certain interface to perform some additional calculations. For example, a component calculating the drag on a wing might provide a socket to allow the user to insert a custom flap drag calculation. In the example problem section, a Design Of Experiment (DOE) Driver uses a socket to allow different types of DOE to be used.

OpenMDAO supports both optional and mandatory sockets. An *optional* socket can be used when its function is not necessary but will enhance the calculation somehow. If the user does not put anything in an optional socket, the analysis simply continues without using it. A *mandatory* socket is required for the execution of a Component. If no object is inserted into a mandatory socket, then an error will be raised and execution cannot continue.

B. Variables and Data Passing

In OpenMDAO, data is passed between components using I/O-traits. I/O-traits are variables that can be connected when an output of one component is required as an input of another component. Python is a weakly typed language, so the Enthought Traits[9] package was used to create a set of strongly typed variable classes. This means, for example, that an integer variable can only be assigned a value that is an integer, and an exception will be raised for an attempt to assign it a string value. The OpenMDAO standard library includes several basic variable I/O-Trait types (Float, Int, Array, Enum, Str, Bool). Additionally, new data types can be created, including those that are complex data objects. Any arbitrary Python object can be passed as data between components using an appropriate I/O-Trait.

I/O-Traits have attributes. Floats and Ints have *high* and *low* attributes which specify the maximum and minimum values respectively for that variable. Any attempt to assign a value outside of those limits results in an exception. All variables have a default value to which they can be reset.

OpenMDAO supports variables with explicitly defined units using the Float or Array variable types. The following specific behavior is enforced when a variable with a defined unit is connected to another framework variable. The system:

- Automatically converts a value passed from an output to an input with compatible units (e.g., “inch” and “m”)
- Raises an exception when an attempt is made to pass a value from an output to an input that has incompatible units (e.g., “kg” and “m”)
- Allows values to be passed between variables without units specified and variables with units; no unit conversion occurs

Note that the unit checking and conversion only occur during data passing in the framework. A variable can be used internally in a component without incurring any additional computational overhead due to the unit capability. The unit conversion is based on the PhysicalQuantities class in Scientific Python[10] but modified for improved performance and flexibility. The unit definitions also come from Scientific Python, although

some additional units were added. A complete list of the default units library is included in the online documentation. The list of units can also be extended by the user either at runtime or with a configuration file.

C. WorkFlows and Iteration Hierarchy

More intricate processes can be implemented in OpenMDAO using WorkFlows in the iteration hierarchy. A WorkFlow is an object that determines execution order for a group of Components. Each driver contains a single WorkFlow. For each iteration, a Driver will execute one pass through the WorkFlow, executing the components contained therein in the order the WorkFlow prescribes. Although in many cases a WorkFlow contains just Components, it can also contain Drivers. This allows nested iterative processes to be created. Figure 6 shows an iteration hierarchy for a collaborative optimization problem with two disciplines.

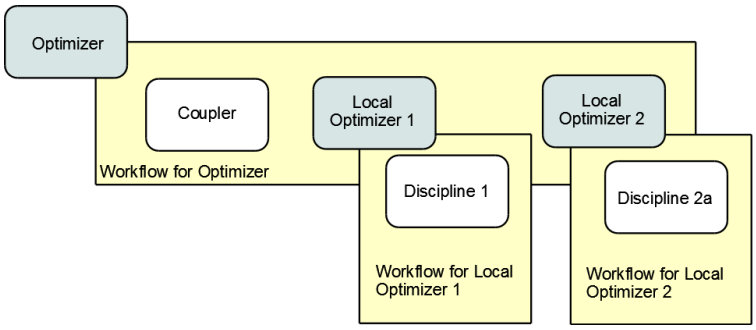

Figure 6: Iteration hierarchy for collaborative optimization.

OpenMDAO also supports a specific subclass of WorkFlow, called DataFlow. A DataFlow infers the execution order of an analysis based on the variable connections between the components. For example, Fig. 7 shows a model with three connected components: A, B, and C. Since Component C relies on input from Component B, which in turn relies on input from Component A, the execution order clearly needs to be A, B, C. OpenMDAO's DataFlow determines this order so that the driver can execute these components in the proper order.

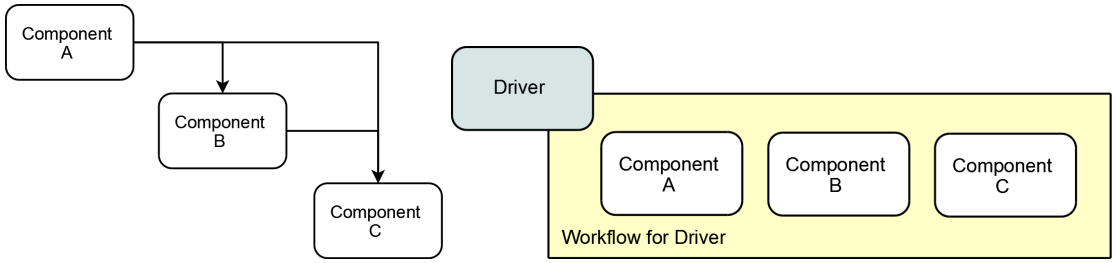

Figure 7: Left: Data flow for a simple problem. Right: Work flow inferred automatically from data flow.

IV. Example - Single Objective Expected Improvement Based Optimization

Jones et al. investigated in depth the use of Kriging surrogate models on the Branin function in their paper on Efficient Global Optimization.[11] The single-objective Expected Improvement (EI) method with Kriging surrogate models they describe was implemented in OpenMDAO. For consistency, the same test function — the Branin function — was used to exercise the algorithm.

A. Problem Definition

$$f(x,y) = \left(y - \frac{5.1}{4\pi^2}x^2 + \frac{5}{\pi}x - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)cos(x) + 10 \quad (1)$$

The Branin function, defined by Eq. (1) and seen in Fig. 8, provides an interesting test case for this EI method because it has three equal global minima in $0 <= x <= 15$ and $-5 <= y <= 10$. The location of each of the three minima is provided in Table 1. This provides a challenge to surrogate-based optimization. To accurately represent the set of optimal solutions, any surrogate model would need to represent the areas around each minimum with high accuracy. Otherwise, one of the minima could incorrectly appear to be a single global minimum.

Table 1: Three global minima for the Branin test function.

x	Y	f(x,y)
$-\pi$	12.275	0.39789
π	2.275	0.39789
9.42478	2.745	0.39789

In the case of a Kriging surrogate model, high accuracy infers the need for high sample point density around each minimum. By generating sample points using a full-factorial Design Of Experiments (DOE), an initial point set can be created. Because the Branin function is a simple algebraic equation, it is feasible to simply increase the number of steps for each parameter in the DOE and run a lot of points. With sufficient density of the DOE, the surrogate would have sufficient accuracy.

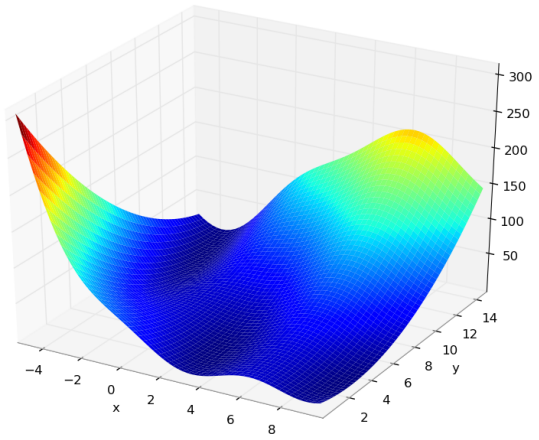

Figure 8: The Branin function.

This approach is not feasible for almost any real problem, where the computational cost is non-trivial and there are more than two degrees of freedom. Instead, the EI method evaluates the surrogate model to find the point with the best chance of improving the predicted optimum, which is measured as the point with the highest value of EI. This point is then added as a new training point for the surrogate model, and the process iterates until some stopping condition is reached. In this way, each successive training point is *adaptively* selected based on the current knowledge of the model behavior and the training data. Computational time is not wasted evaluating the model in places where its value is likely to be much higher than the current known minimum.

The EI function (Eq. (2)) calculates an estimate of the improvement in the surrogate model's prediction of the optimum due to the addition of a given new training point. Specifically, the EI is a measure of the amount in objective value a given point has the potential to improve relative to the current best point. Also, EI is calculated in the same units as the objective function, so it's value is directly comparable to the objective itself.[12]

$$EI(x) = \left(y^* - \mu(x)\right)\Phi\left(\frac{y^* - \mu(x)}{\sigma(x)}\right) + \sigma(x)\phi\left(\frac{y^* - \mu(x)}{\sigma(x)}\right) \tag{2}$$

In Eq. (2), y^* represents a given value for the optimum of the function. For this process, y^* is calculated by filtering the current set of training data to find the point with the lowest objective value. $\mu(x)$ and $\sigma(x)$ represent the mean and standard deviation of the surrogate model prediction at a given point, x. $\Phi(\cdot)$ and $\phi(\cdot)$ represent the standard normal density and the distribution functions respectively.

B. Implementation

Figure 9 shows the iteration hierarchy used in OpenMDAO to implement the single objective EI optimization. When an analysis is run, the first thing that happens is that the top level Driver, called *driver,* executes its WorkFlow. For any simulation in OpenMDAO, the top level Driver is always named *driver*. The WorkFlow of *driver* will execute *DOE_trainer* and then *iterate* consecutively. Since both *DOE_trainer* and *iterate* are

Drivers themselves, each has its own WorkFlow that gets executed whenever the WorkFlow is run. In this case, *driver* simply executes one time, and then it is done. This nesting of Drivers with individual WorkFlows is how an iteration hierarchy is defined within OpenMDAO.

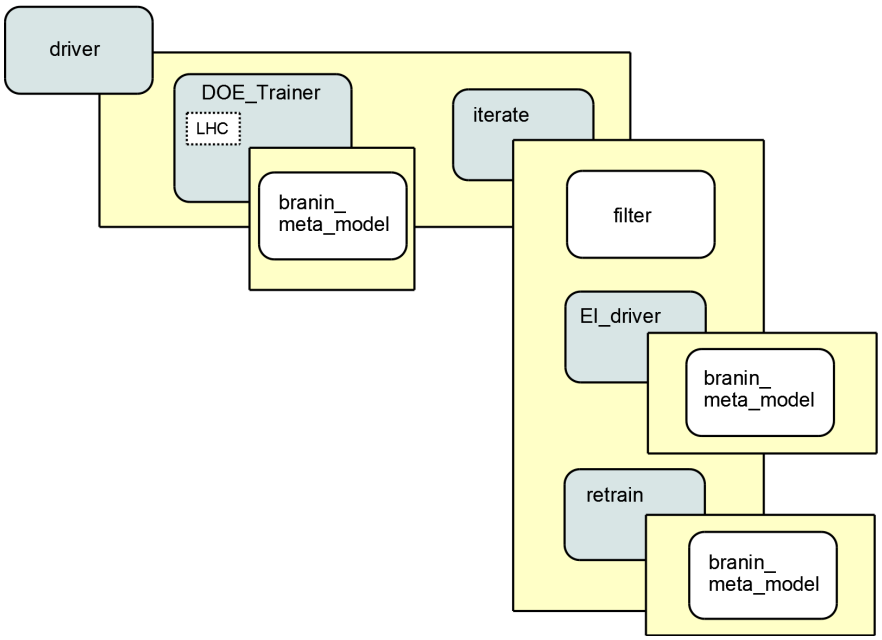

Figure 9: Iteration hierarchy for the single objective EI optimization.

DOE_trainer has a WorkFlow that contains just *branin_meta_model*. When *DOE_trainer* runs, it iteratively runs each point in the specific DOE on *branin_meta_model*. This allows the *branin_meta_model* component to collect a set of initial training data points. Jones suggests that for this problem a 21-point Latin Hypercube DOE is sufficient to seed the surrogate model and bootstrap the adaptive sampling process.[11] *DOE_trainer* is an instance of the DOEdriver class, which has a socket for a DOEgenerator. A DOEgenerator provides the DOEdriver with non-dimensional values to assign to each parameter for a given case. In this example, a Latin Hypercube DOEgenerator is used, but other DOEgenerators could be be substituted into the DOEdriver.

The *branin_meta_model* component is an instance of the MetaModel class. This class provides generalized meta modeling capabilities to the OpenMDAO framework. It has two sockets, one for a particular surrogate model generator and a second for the model that is being approximated. In this case, a Kriging surrogate model generator was used along with a simple component implementing the Branin test problem. The MetaModel component has two operating modes: training and prediction. When run in *training* mode, Metamodel passes its given inputs down to the model in its socket and runs the model. Then it stores the outputs from the model to use for generating a surrogate model later. When run in *predict* mode, MetaModel will check for any new training data and, if present, will generate a surrogate model for each model output with the data. Then it will make a prediction of the model outputs for the given inputs.

After *DOE_trainer* finishes executing the full set of cases in its DOE, *iterator* is executed. The *iterator* repetitively executes its WorkFlow until a stopping condition (provided by the user) is satisfied. In the case of an EI algorithm, a good choice of stopping condition is when the value of EI drops below some specified proportion of the optimum. Another option would simply be to iterate until a specific number of iterations has been reached. Here, both stopping conditions are applied: *iterator* will iterate 30 times or until the value of EI for the next case drops below .03 (about 10% of the optimum value), whichever occurs first.

The WorkFlow of *iterate* includes three components: *filter, EI_driver,* and *retrain.* These three components comprise the core of the EI adaptive sampling algorithm. The *filter* is a ParetoFilter component, which filters a given set of cases into Pareto-optimal cases and dominated cases. ParetoFilter can be given

any number of criteria, from 1 to N, for judging cases. In the situation where only one criterion is given, as in this example, a ParetoFilter instance will select the single best case. *EI_driver* is an instance of SingleCritEI and is responsible for finding the point in the design space with the highest value for EI, given the current training data set. Lastly, *retrain* is an instance of CaseIteratorDriver, and its job is to execute the case that *EI_driver* found and retrain *branin_meta_model* with that point.

EI_driver and *retrain* both have their own WorkFlows, which contain only *branin_meta_model.* In fact, this is the same instance of *branin_meta_model* that is in the WorkFlow for *DOE_trainer.* Each of the three drivers is iterating on the same component.

C. Results

Figure 10 shows the iteration history of the EI algorithm as it moved toward completion. The initial set of 21 DOE points is shown in brown. Each successive sample point shows as a color, ranging from pink to yellow, indicating when (which iteration) it was selected. Pink indicates earlier iterations, and the color progressively transitions to yellow for the later iterations. There were 10 sample points taken, and then the iterations ended due to the value of EI dropping to below the stopping threshold. Including the initial DOE, there were a total of 31 sample points taken. The 10 EI-based sample points are highly clustered around each of the three minima in the Branin function. However, the distribution of pink and yellow points is also fairly even among all of the three minima. The algorithm would start sampling points around one minimum, and when the EI around that one got too low, then it would move on to another one. Effectively the EI algorithm was allotting equal amounts of sample points to each minimum. This is the proper behavior since each minimum has equal values.

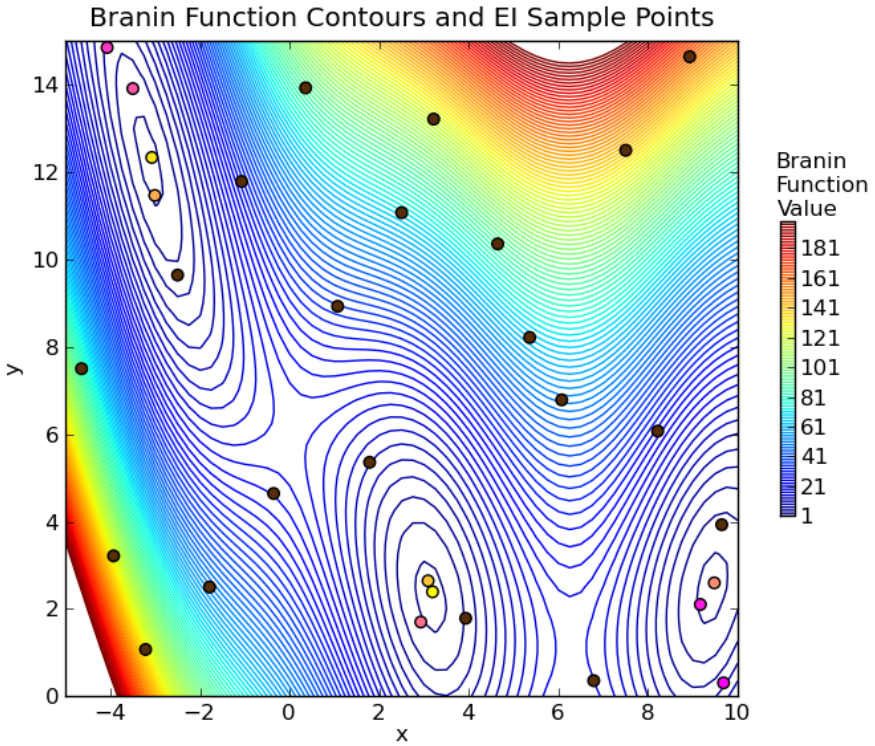

Figure 10: Iteration history of the EI algorithm. Branin function contours are displayed. Brown points represent the initial DOE. EI iteration points are colored from pink to yellow representing iteration count.

It should be noted that the SingleCritEI driver class makes use of a genetic algorithm, internally, to find the point with the maximum expected improvement. Since genetic algorithms are stochastic in nature, two successive runs of the EI algorithm will not necessarily result in the exact same sample point set.

After the EI process was completed, the *branin_meta_model* component was tested at each of the three known minima. The resulting predictions are shown in Table 2.

Table 2: Branin test function, f(x,y), compared to the predicted values, p(x,y).

x	Y	f(x,y)	p(x,y)
$-\pi$	12.275	0.39789	0.3925
π	2.275	0.39789	0.40482
9.42478	2.745	0.39789	0.48228

Although the data in Table 2 shows prediction errors around 20%, the key feature of the data is that all three of the minima have values that are very close to each other. Also, considering that the Branin function takes values as high as 200 in other regions of the design space, values below 0.5 would all look to be excellent candidates. Fig. 11 shows the contours from the predicted value from the *branin_meta_model* component. There is clear agreement between the predicted function and the true contours in Fig. 10

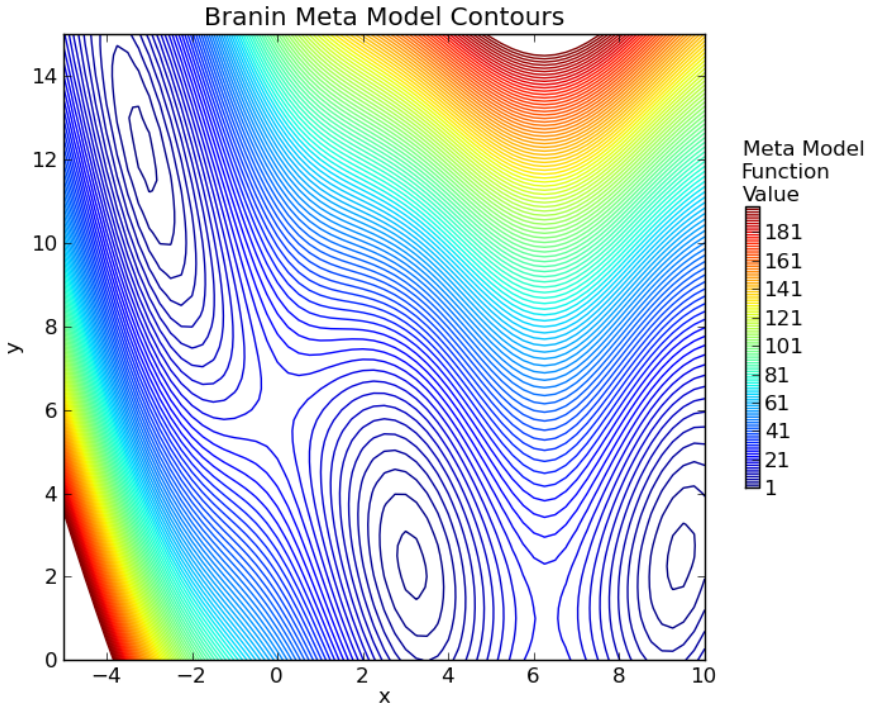

Figure 11: Contours of the predicted function from the meta model, after the EI sampling process.

V. Conclusion

The OpenMDAO framework is being developed to meet NASA's MDAO needs, in particular, the need to encapsulate more advanced MDAO processes and the need to allow the integration of higher-fidelity tools into the design process. Leveraging the advantages of open source development, a process has been developed to allow for the inclusion of code contributions from sources outside the core development team. Crucially, OpenMDAO is licensed with a non-viral license, NOSA, which allows non-open and proprietary plugins to be integrated into the framework.

To encourage the adoption of OpenMDAO by the widest possible audience, comprehensive software documentation has been created for OpenMDAO. This documentation has been integrated into OpenMDAO's development repository so that it is version controlled with the rest of the code base. This ensures that documentation for a specific OpenMDAO release, or even a particular OpenMDAO development revision, is always available. As the documentation is updated to reflect new capabilities and changes to the framework, users will still have access to older documentation for the version they are employing.

The example problem investigated here provides a detailed picture of what can be accomplished in terms of defining MDAO processes within the OpenMDAO framework. This example implements an EI algorithm which has been well examined in the literature. The algorithm includes initialization from a DOE and a nested iteration hierarchy that displays the versatility of the WorkFlow definition within OpenMDAO. Also displayed is the ability to have the same component in multiple workflows. The Branin test problem can easily be replaced with a real analysis component. In this way, OpenMDAO supports the development of MDAO processes which can be deployed for use on multiple problems. This capability lets users build up a library of processes to pick from whenever an analysis demands MDAO techniques.

Having developed this implementation, it would be trivial to swap out the component for the Branin test problem with a real analysis component and apply the same EI algorithm. Easily interchangeable components such as this will permit the construction of MDAO tools and process libraries, whose features may be individually called and applied to analyses demanding specific MDAO techniques. In this way, OpenMDAO will continue to support process development for use in all phases of the design cycle from conception through manufacturing. The open source nature of the framework will facilitate developer interaction from academia, government, and industry while advancing the future state of MDAO through broad-based community support and collaboration.

VI. Future Work

Development of the OpenMDAO framework is ongoing. One major effort for the next year will be the design and implementation of a cross-platform Graphical User Interface (GUI) to provide an alternative to the Python scripting interface. Additional efforts will include developing the ability to integrate component-provided analytical gradients into a framework-wide gradient calculation scheme, the continued development of an Application Programming Interface (API) to interface with geometry objects, and some code optimization to improve the efficiency of the framework infrastructure. Additionally, quite a few drivers and aerospace analysis components will be wrapped for use in OpenMDAO.

Acknowledgments

This work is supported by the Subsonic Fixed Wing project under NASA's Fundamental Aeronautics Program.

Thanks goes to Damon Rousis, of the Aerospace Systems Design Laboratory at the Georgia Institute of Technology, for his support in developing the Expected Improvement algorithms and implementing them within the OpenMDAO framework.

References

[1] Moore, K., Naylor, B., and Gray, J., "The Development of an Open-Source Framework for Multidisciplinary Analysis and Optimization," *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, AIAA, Victoria, Canada, August 2008, 2008-6069.

[2] "Trac," http://trac.edgewall.org/.

[3] "NASA Fundamental Aeronautics Program Multidisciplinary Analysis and Optimization (FAP MDAO) Software Requirements Specification," January 2008.

[4] "Buildbot," http://www.buildbot.com/.

[5] Fogel, C., *Producing Open Source Software*, O'Reilly, London, October 2005.

[6] "Pylons," http://pylonshq.com/.

[7] "Sphinx," http://sphinx.pocoo.org/.

[8] "Pyevolve," http://pyevolve.sourceforge.net/.

[9] "Enthought Traits," http://code.enthought.com/projects/traits/.

[10] "Scientific Python," http://dirac.cnrs-orleans.fr/plone/software/scientificpython/.

[11] Jones, D. R., Schonlau, M., and Welch, W. J., "Efficient Global Optimization of Expensive Black-Box Functions," *Journal of Global Optimization*, Vol. 13, 1998, pp. 455–492.

[12] Keane, A. J., "Statistical Improvement Criteria for Use in Multiobjective Design Optimization," *AIAA JOURNAL*, Vol. 44, 2006, pp. 879–891.