Automatic Implementation of Multidisciplinary Design Optimization Architectures Using $\pi$MDO

by

Christopher Marriage

A thesis submitted in conformity with the requirements for the degree of Masters of Applied Science Graduate Department of Aerospace Engineering University of Toronto

© Copyright by Christopher Marriage 2008
Abstract

Automatic Implementation of Multidisciplinary Design Optimization Architectures
Using πMDO

Christopher Marriage
Masters of Applied Science
Graduate Department of Aerospace Engineering
University of Toronto
2008

Multidisciplinary Design Optimization (MDO) provides optimal solutions to complex, coupled, multidisciplinary problems. MDO seeks to manage the interactions between disciplinary simulations to produce an optimum, and feasible, design with a minimum of computational effort. Many MDO architectures and approaches have been developed, but usually in isolated situations with little chance for comparison. πMDO was developed to provide a unified framework for the solution of coupled optimization problems and refinement of MDO approaches. The initial implementation of πMDO showed the benefits of a modular, object oriented, approach and laid the groundwork for future development of MDO architectures. This research furthered the development of πMDO by expanding the suite of available problems, incorporating additional MDO architectures, and extending the object oriented approach to all of the required components for MDO. The end result is a modular, flexible software framework which is user friendly and intuitive to the practitioner. It allows complex problems to be quickly implemented and optimized with a variety of powerful numerical tools and MDO architectures. Importantly, it allows any of its components to be reorganized and sets the stage for future researchers to continue the development of MDO methods.
Acknowledgements

I would like to thank Dr. Martins for his guidance and support throughout this project. I would also like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Toronto for providing the financial support to undertake this research. Finally, I would like to thank my family and my wonderful fiancee for providing the emotional support and encouragement required to complete this project.
# Contents

## List of Tables
- vi

## List of Figures
- vii

## List of Symbols
- ix

## 1 Introduction
1. Background .......................................................... 1
1.2 Motivation .......................................................... 2
1.3 Proposed Research ............................................... 3
1.4 Contributions ...................................................... 4

## 2 Theory
2.1 MDO Architectures ................................................ 6
   2.1.1 Multidisciplinary Design Feasible (MDF) .................. 8
   2.1.2 Individual Discipline Feasible (IDF) ...................... 9
   2.1.3 Simultaneous Analysis and Design (SAND) ................ 10
   2.1.4 Collaborative Optimization (CO) .......................... 11
   2.1.5 Concurrent Subspace Optimization (CSSO) ............... 14
   2.1.6 Bi-Level Integrated System Synthesis ........ ............ 17
2.2 Semi-Analytic Sensitivity Analysis ............................. 20
   2.2.1 Global Sensitivity Equations .............................. 21
   2.2.2 Adjoint Method ............................................. 22
2.3 Meta Model Approximations ...................................... 23
2.4 Reconfigurability of MDO problems ............................. 24

## 3 Methodology
3.1 $\pi$MDO ........................................................... 26
3.2 pyOpt ............................................................. 28
   3.2.1 pyOpt interface .......................................... 29
   3.2.2 Modifications to existing architectures ............... 30
3.3 pyMMT ............................................................. 31
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Comparison between original SNOPT and pyOptimization SNOPT implementations for the scalable problem.</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>Failure rates for various architectures on the scalable problem. Trials were considered failures if they did not converge to within $10^{-4}$ of the objective value at the optimum.</td>
<td>56</td>
</tr>
<tr>
<td>4.3</td>
<td>Function call comparison for CSSO with various approximation models. Note the increased computational overhead (convergence time) for a similar amount of function calls between CSSO and MDF.</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>Sensitivity Validation: Selected Semi-Analytic Sensitivities calculated for the Scalable Problem</td>
<td>63</td>
</tr>
<tr>
<td>4.5</td>
<td>Function call comparison for semi-analytic sensitivity methods. Note that this comparison is of the total analysis calls per discipline, including finite differencing.</td>
<td>65</td>
</tr>
<tr>
<td>4.6</td>
<td>Function call comparison for semi-analytic sensitivity methods. Note that this comparison is of the total analysis calls per discipline, including finite differencing.</td>
<td>72</td>
</tr>
</tbody>
</table>
## List of Figures

2.1 Representative MDO problem with 3 disciplines. Note that the problem is fully coupled, i.e. all disciplines require information from all other disciplines. ............................................ 7
2.2 MDF architecture .......................................................... 8
2.3 IDF Architecture .......................................................... 10
2.4 CO Architecture .......................................................... 12
2.5 CSSO Architecture ....................................................... 15
2.6 BLISS Architecture ....................................................... 17
3.1 \( \pi \)MDO Framework. A user provided input problem is translated and optimized using a variety of MDO approaches and tools ........................................... 27
3.2 Simplified UML diagram of \( \pi \)MDO’s interaction with the pyMMT and pyDOE meta modeling packages ................................................... 32
3.3 UML diagram for CO class ................................................. 33
3.4 UML diagram for CSSO class ............................................. 35
3.5 UML Diagram showing implementation of the semi-analytic sensitivity methods ................................................................. 38
3.6 UML diagram for OptProb class ......................................... 39
3.7 UML diagram for MdoProb class ....................................... 40
3.8 UML diagram for MDF class ............................................. 42
3.9 Simplified UML representation of the NEST architecture. Tightly coupled disciplines are clustered together in super disciplines to reduce system level coupling dimensionality. ........................................... 45
4.1 Convergence plot of the SNOPT and SLSQP optimizers on the scalable problem with the IDF architecture. .................................................. 51
4.2 Effects of constraint tolerance on convergence time and error for the modified CO architecture. For the Analytic problem tested, the optimum value is around \( 1 \times 10^{-7} \) .................................................. 54
4.3 Effects of constraint tolerance on function calls and objective function error for the modified CO architecture on the Analytic problem .................. 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Objective convergence of CO architecture with and without constraint relaxation. The relaxed form not only converges faster, but by avoiding the unstable regions near the optimum, converges to a better solution.</td>
</tr>
<tr>
<td>4.5</td>
<td>Performance of the original and constraint relaxation CO architectures for increasing dimensionality of design variables.</td>
</tr>
<tr>
<td>4.6</td>
<td>Performance of the new and original CO architectures for increasing design variables and random coupling between disciplines.</td>
</tr>
<tr>
<td>4.7</td>
<td>Performance of the original and constraint relaxation CO architectures for increasing dimensionality of coupling variables.</td>
</tr>
<tr>
<td>4.8</td>
<td>Performance of the CSSO architecture for an increasing design space. The response surface bounds were increased and the resulting performance degradation was measured.</td>
</tr>
<tr>
<td>4.9</td>
<td>Convergence plot for CSSO architecture and various meta models. Note MDF appears at the far left margin.</td>
</tr>
<tr>
<td>4.10</td>
<td>Calculation times for constraint sensitivities using the adjoint and direct methods.</td>
</tr>
<tr>
<td>4.11</td>
<td>Comparison of sensitivity calculation times using the original complex step differencing method and the new semi-analytic methods. Calculation times are normalized to the multi-disciplinary analysis performed by the MDF architecture.</td>
</tr>
<tr>
<td>4.12</td>
<td>Objective function convergence for Scalable problem with semi-analytic sensitivities. The adjoint version of MDF travels a similar path to the original, but converges an order of magnitude faster.</td>
</tr>
<tr>
<td>4.13</td>
<td>Performance of the semi analytic sensitivity methods for increasing dimensionality of local design variables.</td>
</tr>
<tr>
<td>4.14</td>
<td>Performance of the semi analytic sensitivity methods for increasing dimensionality of local design variables. Coupling between disciplines was randomly selected.</td>
</tr>
<tr>
<td>4.15</td>
<td>Performance of semi-analytic sensitivity methods with increasing coupling dimensionality on the scalable problem.</td>
</tr>
<tr>
<td>4.16</td>
<td>Problem reconfiguration for the NEST architecture. The MDF sub-optimization significantly reduces the coupling variables seen by the system level CO architecture.</td>
</tr>
<tr>
<td>4.17</td>
<td>Convergence plot for the NEST architecture, a hybrid of MDF and CO. Note that the nested formulation has performance between its two constituent architectures.</td>
</tr>
</tbody>
</table>
List of Symbols

Alphanumeric Symbols

\( R, r \) residual of a disciplinary analysis

\( \epsilon_f \) relative error of the objective function compared to the true value

\( \epsilon_{tolerance} \) error tolerance for constraint relaxation version of CO

\( \psi \) adjoint vector used in coupled adjoint method

\( c \) constraint, either disciplinary or global

\( f \) objective function or function of interest

\( J \) interdisciplinary compatibility constraint for CO architecture

\( x \) local design variable (affects only a single discipline)

\( x^{opt}, y^{opt} \) variables at the converged local optimum design point in BLISS

\( y \) output of a disciplinary analysis, or a coupling variable

\( y^{RS} \) approximation of non-local coupling variable, used by CSSO

\( z \) global design variable (affects more than one discipline)

\( z^*, x^*, y^* \) system target variables for CO architecture

\( z_{sys}, y_{sys} \) system level variables for BLISS architecture

Abbreviations

ix
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>πMDO</td>
<td>open object oriented framework for MDO</td>
</tr>
<tr>
<td>BLISS</td>
<td>bilevel integrated system synthesis, MDO architecture</td>
</tr>
<tr>
<td>CO</td>
<td>collaborative optimization, MDO architecture</td>
</tr>
<tr>
<td>CSSO</td>
<td>concurrent subspace optimization, MDO architecture</td>
</tr>
<tr>
<td>GSE</td>
<td>global sensitivity equation</td>
</tr>
<tr>
<td>IDF</td>
<td>individual discipline feasible, MDO architecture</td>
</tr>
<tr>
<td>MDF</td>
<td>multidisciplinary feasible, MDO architecture</td>
</tr>
<tr>
<td>MDO</td>
<td>multidisciplinary design optimization</td>
</tr>
<tr>
<td>NEST</td>
<td>nested sub-optimization MDO architecture</td>
</tr>
<tr>
<td>SAND</td>
<td>simultaneous analysis and design, MDO architecture</td>
</tr>
<tr>
<td>SQP</td>
<td>sequential quadratic programming</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

Multidisciplinary design optimization (MDO) is a relatively young field of study which is dynamically expanding. Its development was spurred by recent advances in computing technology and analysis software, allowing complex systems to be computationally modeled. This allows the optimization process, traditionally performed iteratively by human design teams, to be automated to provide faster and more optimal designs. MDO allows all the relevant disciplines, and design trade-offs, of a coupled problem to be considered simultaneously. It can therefore provide better solutions than those produced by sequential optimization, where individual disciplines are optimized one at a time. Additionally, because they consider the coupling between disciplines, MDO methods may find optimum designs that are not intuitively obvious to the designer. An obvious example is in aircraft design, where considering both aerodynamic and structural disciplines produces a different, and more efficient aircraft design than that produced by optimizing either the aerodynamics or structure of the aircraft independently. MDO, therefore, has the potential to be a valuable design tool for any field involving complex coupled problems that can be computationally modeled.
1.2 Motivation

Though research into MDO methods is ongoing, the majority of MDO implementations are programmed on a “one off” ad hoc basis. Practitioners typically adjust the structure of the MDO architecture to match their input problem and require problem specific data structures. This narrow focus makes comparison between MDO architectures and implementations difficult or impossible. Further, the difficulty of implementing multidisciplinary problems in this way constitutes a barrier to entry for new users hoping to find optimal solutions. The requirement for the practitioner to not only understand the specifics of the problem, but the underlying mathematics of the MDO architectures and optimization algorithms as well, impedes the adoption of MDO as a practical tool for engineering design.

For those developing improved optimization methods, standalone MDO implementations restrict the applications of their efforts. The lack of a single framework within which to compare new and improved MDO methods means subjective judgments must be made based on incomplete and problem specific data. This lack of generality also hampers the development of a unified test suite of problems for MDO research, analogous to those created for single discipline optimization XXX REF. Many potentially suitable test problems do exist, but if implemented in the traditional manner would require excessive programming time and would not be easily portable between architectures.

πMDO

πMDO is a universal framework for the implementation of MDO architectures [1]. It was developed to address the lack of a common standard for the comparison of approaches in MDO research. Written in Python [2], it is structured in an object-oriented fashion, and is designed to intuitively mimic the structure of the multidisciplinary problems it implements. The use of Python also allows analysis codes in other programming languages to be “wrapped”, so the framework can optimize problems using legacy codes on many platforms. πMDO implements several MDO architectures, both monolithic and hierarchic. It contains a simple quadratic response
surface, used by some architectures, and a sequential quadratic programming (SQP) non-linear optimizer with a python interface, pySNOPT \[2, 3\]. Finally, a set of simple test problems was implemented within the framework, allowing the comparison of architectures and sensitivity methods.

πMDO achieved many of its goals, but fell short in some areas. Not all of the common MDO architectures were successfully implemented within the framework, and the performance of others was hampered by the simple response surface. Further, the reconfiguration of problems and architectures was successfully demonstrated, but this adaptability could be further extended to multiple response surfaces, optimizers, and sensitivity methods. Finally, the test suite of problems allowed basic comparisons between architectures, but to truly match the capabilities of the “standard” optimization problem suites, many additional, and more complex, problems are required. This initial implementation, however, laid the groundwork for future development and demonstrated the promise of an object-oriented, truly general, optimization suite.

1.3 Proposed Research

It is proposed to further pursue the original goals of πMDO, to facilitate the implementation of a broad suite of multidisciplinary problems and allow the development of new and improved MDO approaches. Further, this research will leverage the generality of the πMDO framework to provide insight into and comparisons of the performance of MDO architectures for various problems.

Exploiting the modular character of πMDO and the capacity of the framework to handle user-provided sensitivities, semi-analytic sensitivity methods will be incorporated. These semi-analytic methods will allow more efficient calculation of the objective and constraint sensitivities while remaining general and applicable to any problem implemented within πMDO.

MDO architectures which were missing from the original implementation will also be incorporated or improved. The bi-level architectures Concurrent Subspace Optimization (CSSO), Bi level Integrated System Synthesis (BLISS), and Collaborative Optimization (CO) will be tested and benchmarked against the existing architectures.
Section 1.4. Contributions

Using the reconfigurable nature of architectures within \( \pi \text{MDO} \), the benefits of hybrid optimization approaches and combined architectures will also be explored.

To test all of these additions and compare their attributes, the existing suite of problems available to \( \pi \text{MDO} \) will be expanded. Problems will be modified and tailored to explore various aspects of each architecture, and new problems will be implemented from disparate fields of study.

Finally, extending the object-oriented philosophy of the framework to its constituent components, classes will be incorporated allowing multiple optimizers and meta models to be applied. This will complete the \( \pi \text{MDO} \) framework, allowing various multidisciplinary problems, optimizers, approximation methods, and sensitivity methods to be combined and reconfigured to find the best combinations.

As an important secondary objective of this research, it is the intention of the author to always proceed with those researchers who will follow in mind. To this end, the simplification, documentation, and thorough testing of all software is an important component of this effort. Making \( \pi \text{MDO} \) a robust, user-friendly, and powerful suite of MDO tools will allow future researchers to minimize the amount of time spent relearning old lessons, or comprehending the idiosyncrasies of a particular problem implementation. The fundamental aims of \( \pi \text{MDO} \) will be furthered by expanding the community of MDO practitioners and giving researchers a common point of comparison between architectures and approaches.

1.4 Contributions

The further development and refinement of the \( \pi \text{MDO} \) framework will provide future researchers with a setting to explore the practice of MDO.

Specifically, this research will demonstrate a method of applying more efficient semi-analytic sensitivity calculations while maintaining universal adaptability. It will also measure the benefits of this approach compared to the traditional sensitivity methods.

Additionally, it will expand the number of optimization architectures available to users of \( \pi \text{MDO} \), including the distributed bi level architectures CO, CSSO, and
BLISS. Comparisons of the performance of these architectures will be performed and will add to the sparse data set in this area. Further, the new optimizers and meta models incorporated in πMDO will facilitate the gathering of information on their relative performance.

This research will also investigate hybrid approaches that combine existing architectures to produce new methods that mix the characteristics of their constituent elements.

Finally, the generalization of the Optimizer class will also allow open source optimizers to be included with πMDO. With all of its major components, or at least simplified versions of them, publicly available, the software can be easily distributed for research and educational purposes. Once a larger community of users exists, errors in the framework can be corrected and improvements added through the open source model. This will accelerate innovation in MDO research and in the development of the test suite of MDO problems which accompanies πMDO.
Chapter 2

Theory

2.1 MDO Architectures

Multidisciplinary design optimization (MDO) architectures are a means of reorganizing a coupled multidisciplinary problem into the format of one or more single discipline optimization problems. In effect, the different architectures translate a coupled multidisciplinary problem to resemble one or more classical optimization problems. Two broad classes of architectures exist, monolithic and hierarchical (decompositional) architectures [4, 5].

A multidisciplinary problem consists of a coupled system of simulations where each simulation or discipline analysis requires some input from the other disciplines. A representation of the coupling interactions of a multidisciplinary problem with three fully coupled disciplines is shown in Figure 2.1.

The problem can be described using notation familiar to the classical optimization community. The optimization problem consists of minimizing some function $f$ that is dependent on multiple disciplinary analyses. The design variables directly affect the value of the system, either as global design variables, $z$, which affect more than one discipline, or local design variables, $x_i$, which affect a single discipline. The coupled system is also implicitly dependent on the outputs of each discipline analysis through the coupling variables, represented by $y$. These coupling variables are represented for discipline $i$ as in equation (2.1) where required inputs from other disciplines are
Section 2.1. MDO Architectures

Figure 2.1: Representative MDO problem with 3 disciplines. Note that the problem is fully coupled, i.e. all disciplines require information from all other disciplines.

represented as $y_j$. Note that the analysis of discipline $i$ may internally consist of solving for many more state variables, but for the purposes of problem decomposition only the coupling variables must be considered.

\begin{align}
    y_i(z, x_i, y_j), \quad i, j = 1, \ldots, n, \quad i \neq j
\end{align}

Finally, the problem is subject to constraints represented by $c_i$ which can be either local or global in nature. The problem statement for a multidisciplinary problem with $n$ disciplines is given in equation (2.2).

\begin{align}
    \text{minimize} \quad & f(z, x, y_i(z, x_i, y_j)) \\
    \text{with respect to} \quad & z, x_i \\
    \text{subject to} \quad & c(z, x, y_i(z, x_i, y_j)) \leq 0 \\
    \text{for} \quad & i, j = 1, \ldots, n, \quad i \neq j
\end{align}

Alexandrov et. al. showed that through the reconfigurability of MDO problems, all other MDO architectures can be derived from this basic problem statement [6] [7].

The common monolithic architectures, Multidisciplinary Design Feasible (MDF), Individual Design Feasible (IDF), and Simultaneous Analysis and Design (SAND)
reorganize the multidisciplinary problem to solve it as a single classical optimization problem. They are simpler and more robust than decompositional (bi-level) architectures, but because they are essentially a single optimization problem, their performance tends to decrease as the size of the problem increases.

2.1.1 Multidisciplinary Design Feasible (MDF)

The multidisciplinary design feasible [8] method, also known as All At Once (AAO) [4], is the simplest MDO architecture. It consists of solving a multidisciplinary analysis at each design point and is the most intuitive to engineers. The algorithm adjusts the design variables, selects a design point, then performs a fixed point iteration to converge the coupling variables to a multidisciplinary feasible solution. A representation of the MDF architecture is given in Figure 2.2.

![Figure 2.2: MDF architecture](image)

Mathematically, the MDF architecture is identical to the general problem statement given in equation (2.2). The fixed point iteration is performed by solving the system of $N$ coupled analyses given in equation (2.3) which drive the residuals of the discipline states to zero.
Section 2.1. MDO Architectures

\[ r_i (z, x_i, y_i(z, x, y_j)) = 0, \quad i = 1, \ldots, n. \]  \hspace{1cm} (2.3)

MDF is intuitive and robust, and generally will converge to a solution, but requires the entire coupled system to be solved at every iteration. As the size of the problem increases, the fixed point iteration encompasses more and more variables, making MDF impractical for large problems. Further, it provides no autonomy between disciplines, performing each discipline analysis an equal number of times regardless of difficulty. Finally, the sensitivity calculations for MDF become onerous if performed with finite differencing as they involve converging a multidisciplinary feasible solution for each design variable. For this reason the use of semi or fully analytic sensitivity calculations with MDF offers the potential for large performance gains.

2.1.2 Individual Discipline Feasible (IDF)

The Individual Discipline Feasible architecture removes the communication between disciplines, allowing them to be solved separately, and uses the optimizer to ensure interdisciplinary feasibility [8]. This is performed by adding the coupling output variables from each discipline to the design variable set as system targets. Each discipline analysis is performed using these system targets to represent the non-local coupling variables, allowing the analyses to be performed separately and potentially in parallel. Agreement between the system targets and actual discipline outputs is maintained through the use of equality constraints. A diagram of the variable flow for the IDF architecture is given in Figure 2.3.

The problem statement for IDF is given in equations (2.4) and (2.5), where \( y^* \) represents the target coupling variables.

\[
\begin{align*}
\text{minimize} & \quad f(z, x, y^*) \\
\text{with respect to} & \quad z, x, y^* \\
\text{subject to} & \quad c(z, x, y(z, y^*, z)) \leq 0 \\
& \quad y_i^* - y_i(x, y_j^*, z) = 0, \\
\end{align*}
\]  \hspace{1cm} (2.4)
The output states, $y_i$, of each discipline are determined by the discipline analysis such that

$$r_i(z, x_i, y_i(z, x, y^*_j)) = 0, \quad (2.5)$$

where the non-local coupling variables of the other disciplines, $y^*_j$ for $j \neq i$ are determined by the optimizer.

### 2.1.3 Simultaneous Analysis and Design (SAND)

Simultaneous Analysis and Design, like IDF, adds the output coupling variables into the optimization, but does it by setting the discipline analyses as equality constraints at the system level. Mathematically, SAND can be described as in equation \((2.6)\).
minimize \( f(z, x, y(z, x, u)) \)

with respect to \( z, x, y \) \hspace{1cm} (2.6)

subject to \( c(z, x, y(z, x, u)) \leq 0 \)

\[ r_i(z, x_i, y_i(z, x, y_j), u) = 0, \quad i = 1, \ldots, N, \]

The SAND approach has potential benefits in further relaxing the communication between disciplines, but becomes impractical for large scale optimization due to the excessive number of design variables required.

2.1.4 Collaborative Optimization (CO)

Collaborative Optimization (CO) is a bi-level, decentralized architecture with optimizations occurring at both the discipline and system level \([9]\). The problem is decomposed into a system level optimization and a series of independent optimizations at the discipline level. Local feasibility is maintained at the discipline level by the sub-system optimizations as they are responsible for satisfying the local discipline constraints. Inter-disciplinary feasibility is maintained by the use of compatibility constraints. These constraints minimize the discrepancy between the system level targets and the actual values of the coupling and design variables at the optimum. Global constraints are handled by the system level optimizer. The configuration of the CO architecture is shown in Figure 2.4.

Mathematically, the CO architecture can be described at the system level as in equation (2.7). The system level optimizer sets system targets for the global design variables \( z^* \), local design variables which directly affect the optimum \( x^{*_{obj}} \), and for the coupling variables for each discipline \( y^* \). The system level problem calculates the objective value using these targets, and is responsible for satisfying the global constraints \( c_{global} \), and one compatibility constraint for each discipline \( J_i \).
Section 2.1. MDO Architectures

minimize \[ f (z^*, x^*_{\text{obj}}, y^*) \]
with respect to \[ z^*, x^*_{\text{obj}}, y^* \]
subject to \[ c_{\text{global}} (z^*, x^*_{\text{obj}}, y^* (z, x)) \leq 0 \]
\[ J_i (z_i, z_i^*, x_i, y_j, y_j^*) = 0 \] (2.7)

The subsystem problem is defined as in equation (2.8). At the subsystem level, the objective is to minimize that particular discipline’s compatibility constraint, as given in equation (2.9), while satisfying the local constraints. The subsystem optimization has authority over the local and global design variables \( x_i \) and \( z_i \), and also over the coupling inputs from other disciplines \( y_j \). This formulation corresponds to the \( CO_2 \) formulation presented in [9] which uses a quadratic penalty function as the compatibility constraint.

minimize \[ J_i (z_i, z_i^*, y_j, y_j^*) \]
with respect to \[ x_i, z_i, y_j \]
subject to \[ c (x_i, z_i, y_i (x_i, y_j, z_i)) \leq 0 \] (2.8)

Figure 2.4: CO Architecture
Where the compatibility constraint for each discipline $J_i$ is calculated as:

$$J_i = \sum (z_i - z_i^*)^2 + \sum (y_i - y_i^*)^2 + \sum (y_{ji} - y_{ji}^*)^2$$

(2.9)

The classic formulation of collaborative optimizations runs into convergence difficulties as it approaches the optimum. These convergence difficulties were noted early on by Braun et. al. [10] and were found to cause degenerate solutions and excessive computational effort as the algorithm approached the optimum. Degeneracy in this context means that solutions to the sub-system optimization problems are unable to satisfy the necessary conditions for optimality at the system level [11] [12]. This is mathematically apparent from the formulation of the subsystem objective function, as the system level targets and local variables must be in agreement at the optimum. If they are in agreement, $x_i = x_i^*$, $z_i = z_i^*$, and the gradient of the objective goes to zero as the subsystem nears the optimum. This zero gradient necessitates the use of post-optimality sensitivity calculations for the system level constraint sensitivities. This leads to non-smoothness of the design space, further compounding the convergence difficulties of CO.

Several remedies for these issues have been proposed, including a modified formulation known as modified collaborative optimization (MCO) replacing the system level objective with a penalty function [13] or inexact penalty functions at the subsystem level [11]. These methods did not totally solve the convergence problems, however, and could produce local optima and a very non-smooth design space. The use of response surfaces to represent subsystem information was also proposed [14] but while effective, led to increased complexity of the problem and suffers the shortfalls of any approximation method. Braun, Gage, et al. proposed the use of slack variables with equality constraints as a means of smoothing the design space [10]. They also noted that due to the line search method used by SQP optimizers, formulating the problem with inequality, rather than equality, constraints at the system level produced better performance. Perez et. al. [15] [16], in reviewing the potential improvements to CO, found that a combination of slack variables and inequality constraints ameliorated the numerical instabilities and sped convergence, subject to a small loss of final design
point accuracy. This was accomplished by reformulating the system level compatibility constraints as inequalities subject to some error tolerance. For each discipline, this constraint becomes

\[ J_i \left( z_i, z_i^*, x_{i,\text{obj}}, x_{i,\text{obj}}^*, y_i, y_i^*, y_j, y_j^* \right) - \epsilon_{\text{tolerance}} \leq 0 \]  \hspace{1cm} (2.10)

with an error tolerance, \( \epsilon_{\text{tolerance}} \), that can vary based on the problem, but is usually in the range of \( 1 \times 10^{-6} \) to \( 1 \times 10^{-8} \). This method leads to a small reduction in the accuracy of the final optimum, subject to the error tolerance, but prevents the optimizer from entering the degenerate regions of the design space around the optimum.

### 2.1.5 Concurrent Subspace Optimization (CSSO)

Concurrent Subspace Optimization (CSSO) is a bi level architecture where an approximation of the non-local coupling variables is provided to each discipline by a meta model \[17\]. A diagram of the structure of CSSO is given in Figure 2.5. Essentially, CSSO populates a response surface or other approximation model with information on multidisciplinary feasible solutions across the design space. This approximation of the states of the system is then used by the system and sub-system level optimizations. The intention of the response surface is to provide a computationally inexpensive approximation of the output state variables for each discipline.

At the system level, the optimizer modifies the global and local design variables, \( z \) and \( x \), and uses information from the response surface to approximate the coupling variables \( y^{RS} \). The system problem can be stated as in equation (2.11). The objective function is the same as that of the multidisciplinary problem, but the values of the discipline outputs \( y^{RS} \) are provided by the approximation model.

\[
\begin{align*}
\text{minimize} & \quad f \left( z, x, y^{RS} \right) \\
\text{with respect to} & \quad z, x \\
\text{subject to} & \quad c \left( x, z, y^{RS} \right) \leq 0
\end{align*}
\]  \hspace{1cm} (2.11)
At the discipline level, each optimization can be stated as in equation (2.12). The objective function is the same as the system level objective, except the subsystem is given authority over a smaller set of design variables. For each discipline, the optimizer modifies the local design variables, $x_i$, and any globals assigned to that discipline, $z_i$, to minimize the objective function. For the sub-system analysis, the local coupling output variable, $y_i$, comes from the discipline analysis rather than the response surface. The non-local coupling information $y_j^{RS}$ comes from the response surface as in the system problem. Global and local design variables not controlled by the discipline level optimization, $x_j$ and $z_j$, are held constant at their system level values.
Section 2.1. MDO Architectures

\[
\begin{align*}
\text{minimize} & \quad f(z_i, z_j, x_i, y^{RS}) \\
\text{with respect to} & \quad z, x \\
\text{subject to} & \quad c(x, z, y^{RS}) \leq 0
\end{align*}
\] (2.12)

For the subsystem optimizations, the CSSO algorithm first distributes the local and global design variables to their respective disciplines. The global design variables are apportioned according to their relative influence on each discipline, with control over the variable given to the discipline optimization upon which it has the greatest impact. This can be decided subjectively, through the insight of the practitioner, or by measuring its influence on the Global Sensitivity Equations (GSE). By calculating the cross coupling derivatives as described by Sobieski [18], a measure of the influence of each design variable on the discipline outputs can be obtained. This global variable can then be assigned to the discipline upon which it has the largest influence on the objective function through the GSEs.

After the problem is decomposed into system and subsystem level optimizations, the response surface or meta model is populated. Based on a representative sample of points according to the appropriate design of experiments, a multidisciplinary analysis is performed at each point and the output coupling variable values for each discipline are stored. Using this sample, an approximation is built up which covers the design space of the problem. The system level optimizer minimizes the objective function using these approximate discipline outputs and records the optimal design point. Starting from this system optima, the individual subsystem optimizations then proceed. The subsystem optima with the lowest objective function value is selected and that design point is used to update the response surface and the system level design point. Due to the computationally inexpensive approximation, the impact of the speed of the sensitivity analysis at both the system and subsystem level is reduced.
2.1.6 Bi-Level Integrated System Synthesis

Bi-level Integrated System Synthesis (BLISS), is a bi-level architecture originally proposed by Sobieski et. al [19], as a means of coordinating a system level optimization problem while allowing individual discipline optimizations freedom within their design space. The BLISS algorithm is a logical offshoot of the Global Sensitivity Equations [18] with discipline optimizations weighted by their impact on the GSE. Significant effort has gone into developing BLISS and dealing with the complexity issues of a bi-level architecture. The current version, BLISS 2000 [20], goes beyond the original algorithmic formulation in the use of meta models to handle data exchange between disciplines. This allows for inexpensive function calls once the model is developed and also provides opportunities for distributed and/or parallel processing. A diagram of the BLISS algorithm is given in Figure 2.6.

![BLISS Architecture Diagram](image)

Figure 2.6: BLISS Architecture

The BLISS2000 algorithm separates the multidisciplinary problem into a system
optimization, and one sub-system optimization for each discipline. Initially, the sub-system optimizations are performed at a number of points distributed within the design space of the system level problem. To increase the performance of the meta model, these points should be distributed according to a formal Design of Experiments (DOE) methodology. A meta model is then derived which models the optimal outputs of the sub-system optimizations with respect to the system level design variables. Information on each discipline’s effect on the system objective function is modeled by the use of weighting factors, $w_i$, for each output variable, $y_i$, of a discipline. These weighting factors replace the GSE derived weightings of the original BLISS algorithm. Mathematically, the sub-system optimization problem for discipline $i$ can be described by equation \((2.13)\).

\[
\begin{align*}
given & \quad z_{sys}, y_{jsys}, w_i \quad \text{(from system level)} \\
\text{minimize} & \quad \sum w_i y_i \\
\text{with respect to} & \quad x_i, y_i \\
\text{subject to} & \quad c_{local}(x_i, z_{sys}, y_{jsys}, y_i) \leq 0 \\
& \quad r_i(z_{sys}, x_i, y_{jsys}, y_i) = 0
\end{align*}
\]  

(2.13)

The subsystem seeks to minimize an objective function composed of the sum of the discipline analysis outputs $y_i$ multiplied by their weighting factors $w_i$ as provided by the system level problem. The local optimizations are responsible for satisfying the discipline constraints $c_{local}$. The equality constraint $r_i$ represents the solution of the discipline analysis at each point, where the residuals are driven to zero. The given quantities $z_{sys}$, $y_{jsys}$, and $w_i$ are set based on the design of experiments when generating the discipline response surface. $z_{sys}$ represents the global design variables and $y_{jsys}$ represents the non-local coupling variables required as inputs to discipline $i$. The output from each discipline optimization is the variable set at the optimum, both the discipline analysis outputs $y_i^{opt}$ and the local design variables $x_i^{opt}$. Note that the value of the subsystem objective at the optimum is superfluous and is not returned to the system level or modeled by the response surface. The system level problem for BLISS2000 can be described as,
minimize \( f(z, x_i^{opt}, y_i^{opt}, w_i) \)

with respect to \( z_{sys}, y_{sys}, w_{sys} \)

subject to \( y_{sys} - y_i^{opt} = 0 \)

for \( i = 1, \ldots, n \) \hspace{1cm} (2.14)

In effect it is the application of the IDF architecture to the system level problem. The optimizer seeks to minimize the objective function of the original problem, \( f \), which is calculated using the optimal outputs queried from the subsystem response surfaces. The equality constraints \( y_{sys} - y_i^{opt} = 0 \) ensure that the system level targets are driven to match the optimal outputs of the discipline sub-optimizations.

Due to the error inherent in any approximation model, BLISS2000 requires multiple iterations between the system and subsystem level. Additionally, the performance of meta models is greatly improved when they are operating within narrow bounds, so some limitation of the initial design space at both the system and subsystem level is desirable. The use of a trust region technique to adjust the bounds of the system and subsystem design space after each system level optimization is recommended. This requires regeneration of the meta models when the bounds of the design space are reached.

The general procedure of the BLISS2000 architecture is to first perform an MDA to condition the design space, then generate meta model approximations for each of the subsystem optimizations. After the accuracy of these approximations is checked within their respective bounds, the system level optimization is performed using data drawn from these response surfaces. After each system optimization the accuracy of the approximation and relationship to the design space bounds are checked for each sub-optimization and adjusted or regenerated if necessary. If the accuracy is acceptable and the bounds have not been violated, the system optimum is checked against the convergence criteria and the optimization is concluded if they are met.

The BLISS2000 algorithm reduces some of the mathematical difficulties and complexity of the original BLISS formulation. The use of meta model approximations also inherently allows for parallel processing of the sub-system optimizations, which
can greatly improve performance. The dependence on a response surface or approximation model, however, is subject to the curse of dimensionality where the scales geometrically with the number of design variables. To make BLISS a competitive architecture, efficient approximation methods, involving the reuse of previous data, or parallel processing is required.

### 2.2 Semi-Analytic Sensitivity Analysis

Semi-analytic sensitivity methods exploit the structure of multidisciplinary problems to calculate the total derivatives without computationally costly finite differencing of the complete coupled system. They encompass the global sensitivity equations [18], and the closely related coupled adjoint methods [21, 22]. In both cases they combine and solve partial derivative terms from each discipline of a coupled system to calculate the total system level derivatives.

For a general multidisciplinary problem as described in equation (2.15), the objective function $f$ is directly dependent on the global and local design variables $z$ and $x$, and implicitly dependent on each discipline analysis $y(z, x)$. $c$ represents a constraint at either the discipline or global level, and $R$ represents the residuals of the governing equations for each discipline which should be driven to zero at a multidisciplinary feasible solution.

\[
\begin{align*}
\text{minimize} & \quad f(z, x, y_i(z, x, y_j)) \\
\text{with respect to} & \quad z, x_i \\
\text{subject to} & \quad c(z, x, y_i(z, x_i, y_j)) \leq 0 \\
& \quad \text{for } i, j = 1, \ldots, n, i \neq j
\end{align*}
\tag{2.15}
\]

The sensitivity of the objective function $f$ with respect to the design variables can be written, using the chain rule, as equation (2.16). For the sake of brevity $x_n$ represents the set of design variables $z, x$ at the system level.

\[
\frac{df}{dx_n} = \frac{\partial f}{\partial x_n} + \frac{\partial f}{\partial y_i} \frac{dy_i}{dx_n}
\tag{2.16}
\]
Martins \[23\] showed that this was analogous to equation (2.17) assuming a converged multidisciplinary solution (residuals are equal to zero). The semi-analytic methods applied within $\pi$MDO solve equation (2.17) or (2.16) and essentially differ only in the order in which they perform the required matrix multiplication.

$$\frac{df}{dx_n} = \frac{\partial f}{\partial x_n} + \frac{\partial f}{\partial y_i} \left[ \frac{\partial R}{\partial y} \right]^{-1} \frac{\partial R}{\partial x_n} \tag{2.17}$$

### 2.2.1 Global Sensitivity Equations

Sobieski \[18\] examined the use of discipline level partial derivatives to directly calculate system total derivatives. He showed mathematically that the partials $\frac{\partial R}{\partial y}$ in equation (2.17) were analogous to the partial derivative of the outputs of a discipline with respect to the state variables of the other disciplines in the system $\frac{\partial y_i}{\partial y_j}$. He proposed two versions of the global sensitivity equations, GSE1 and GSE2, where GSE1 uses the partial derivatives of the discipline residuals, and GSE2 uses the partial derivatives of discipline outputs.

The implementation of GSE1 and GSE2 is identical except for the partial derivatives used, so the GSE1, or direct method \[21\] is explained. GSE2 can be implemented by replacing the residual term $R$ with the discipline output $y_i$.

In the direct method, the total derivative $\frac{dy_i}{dx_n}$ is calculated by solving equation (2.18) and then multiplied by $\frac{\partial f}{\partial y_i}$ to find the system totals as in equation (2.16). The direct method requires the solution of $\frac{dy_i}{dx_n}$ once for every design variable, so it is more useful when the number of output functions $k$ is greater than the number of design variables $n$.

$$\frac{\partial R_i}{\partial R_j} \left[ \frac{dy_i}{dx_n} \right] = \frac{\partial R_j}{\partial x_n}, \quad j = 0..k, \; j \neq i \tag{2.18}$$

Within $\pi$MDO, any discipline governing equation which has an affect on another discipline is defined as a coupling variable, hence for GSE2 the partial derivatives of a discipline’s output variables replace the residual partials of GSE1. In situations with iterative or computationally expensive discipline analyses, GSE1 may have an advantage over GSE2. GSE1, however, requires the existence of separate residual
Section 2.2. Semi-Analytic Sensitivity Analysis

analysis code which must be provided by the user. GSE2 is therefore more widely applicable, whereas GSE1 may be faster in certain situations.

2.2.2 Adjoint Method

In the adjoint method, the adjoint vector $\psi$ or $\frac{\partial f}{\partial y_i} \left[ \frac{\partial R}{\partial y} \right]^{-1}$ is first calculated by solving equation (2.19). $\psi$ is then substituted into equation (2.17) to find the total derivative $\frac{df}{dx_n}$.

$$\frac{\partial R}{\partial y} \psi = -\frac{\partial f}{\partial y_i}$$  \hspace{1cm} (2.19)

This requires the vector $\psi$ to be calculated only once for each function of interest, it is therefore useful in situations where the number of design variables exceeds the number of system level functions. Further, as with the GSE2 method, Martins [23] contends that this formulation would also be valid using the partial derivatives of discipline outputs $\frac{\partial u_i}{\partial y_j}$ in place of the residual based formulation of equation (??).

Both the adjoint and direct methods have the potential to reduce the computational expense of finding the total derivatives of a multidisciplinary problem. Because only partial derivatives are required by these methods, they avoid the completion of a system level analysis for every design variable or function. Further, the practitioner has considerable flexibility in calculating the partial derivatives required by these methods. Finite differencing, complex step [24], automatic differentiation [25] [26], or even hand derived analytic methods can be used. $\pi$MDO allows users to define the partial derivatives required or can automatically calculate them using finite differencing or complex step methods. The availability of several variants of semi-analytic method maximizes their flexibility and the potential gains. For example, the direct method can be used to calculate the numerous system level constraints of a problem while the adjoint method is used for the single objective function.
2.3 Meta Model Approximations

A meta model is a mathematical approximation of a black box system. Meta models use various techniques to approximate the response of this black box to stimulus, but share some broad features. Meta models are usually constructed from a sampling of points in the design space according to the formal methodology of design of experiments. Using these distributed points, a mathematical surrogate for the actual analysis is created, and responses from this surrogate model are then used for the optimization process. These approximations are subject to some error, and while their evaluation for a given point is usually much faster than the actual analysis, they are subject to a computational cost proportional, often exponentially, to the number of dimensions being modeled. This is known as the “curse of dimensionality” and has mostly limited the application of meta models, and the MDO architectures that depend on them, to small test problems.

Various methods have been proposed to provide as accurate a model as possible for a given function while minimizing the number of point evaluations required. The simplest method is the classical tactic of polynomial regression, modeling the function as a linear combination of polynomials. Typically the polynomial used is second order, known as least squares regression, but the number of points required is proportional to the number of design variables squared. Polynomial regression, also known as a response surface, was the meta model implemented in the original version of πMDO \cite{27}. Remedies for the poor scaling of polynomial regression have been suggested, including the idea of reusing information from previous points to reduce the number of function evaluations overall. Lee et. al. \cite{28} proposed a method modeled on the classic Broyden-Fletcher-Goldfarb-Shanno (BFGS) optimization algorithm to approximate the cross quadratic terms of the response surface. By using information from successive iterations, this method reduces the minimum number of points required for a given accuracy, and therefore potentially improves the dimensional scaling of the meta model.

The kriging meta model \cite{29} adds stochastic processes to a polynomial regression to produce a more accurate response on a global scale \cite{30}. Though it has some
advantages over polynomial regression, it still requires point evaluations on the order of the number of design variables squared. The Radial Basis Function method is another meta model designed for more accurate approximations over a large design space. It constructs an approximation using multipolynomial functions expressing the Euclidean distance of the design point from a given data point. Because it combines multiple polynomial approximations in a single hypersurface it can model widely distributed points in several dimensions while remaining fairly straightforward to implement [31].

\[ \pi \text{MDO has access to a variety of meta models through integration with pyMMT and pyDOE, the meta modeling libraries within pyACDT [32]. These libraries contain multiple design of experiment functions in pyDOE as well as the Polynomial Regression, kriging, and Radial Basis Function metamodels within pyMMT. Through the contributions of Thomson and Perez [33], these meta models share a common interface and can be readily interchanged with each other. In addition, pyMMT contains a sophisticated progressive validity trust region method to control the bounds of the metamodel and increase the validity of the model as it approaches the optimum [34].} \]

\[ 2.4 \text{ Reconfigurability of MDO problems} \]

Bi-level decompositional architectures were developed in part to allow local autonomy to MDO practitioners in optimizing a given disciplinary system. The performance of all these bi-level architectures, CO, CSSO, and BLISS, is adversely affected by the coupling variables which must be passed between disciplines. In collaborative optimization, the presence of the coupling variables in the system level optimization problem means CO scales very poorly with increasing coupling [5]. The dependence of CSSO and BLISS on meta model approximations means they are subject to the curse of dimensionality, and complexity of their meta models scales geometrically with the number of coupling variables in the system. Sobieski pointedly recommends in [20] that variable condensation techniques be used to reduce this effect in the BLISS2000 algorithm.

The work of Alexandrov and Lewis [6] [7] demonstrated that MDO problems could
Section 2.4. Reconfigurability of MDO problems

be reconfigured and decomposed from the MDF or AAO formulation into a variety of architectures \[35\]. \(\pi\)MDO takes advantage of this reconfigurability to derive many optimization architectures from a single input format. The object oriented nature of \(\pi\)MDO means that not only are the disciplines objects which can be easily reshuffled and reconfigured, but the architectures themselves are also reconfigurable. The opportunity exists within \(\pi\)MDO to readily create hybrid or reconfigured architectures to address the shortfalls of the bi level architectures.
Chapter 3

Methodology

The original structure of πMDO is extensively described in [1] and [36]. A general overview is given below, and the new or extensively modified elements of πMDO are described in further detail.

3.1 πMDO

πMDO was created as a unified framework for multidisciplinary design optimization, a vehicle to both decrease the difficulty of implementation and support the further development of MDO methods. The general structure of πMDO is shown in Figure 3.1. Initially described as a unified framework for the implementation of MDO architectures [36], it provided the object oriented structure to decompose MDO problems and optimize them.

The initial implementation of πMDO provided a structure to translate a user defined problem into a single discipline optimization format. The object oriented layout mimicked the format of the MDO problems themselves, incorporating discipline objects which contained the analysis and constraint functions and information on inter-disciplinary connections. πMDO also made extensive use of inheritance and code reuse, where attributes from the optimization problem instance were inherited by the multidisciplinary problem instance and finally the optimization architectures themselves. Once decomposed, the non-linear SQP solver SNOPT [2] was used to
optimize the problems given an objective and constraint function. A variety of optimization architectures were implemented, including the monolithic architectures Multidisciplinary Design Feasible (MDF), Individual Discipline Feasible (IDF), and Simultaneous Analysis and Design (SAND). Several bi-level decomposition architectures were also attempted, including versions of Collaborative Optimization (CO), Concurrent Subspace Optimization (CSSO), and bi-level integrated system synthesis (BLISS). Sensitivity analysis could be performed by conventional finite differencing or the complex step method [24], or the user could provide their own derivatives in the SNOPT format. As a proof of concept, the initial version of πMDO performed admirably. It demonstrated the flexibility of the object oriented approach and allowed the comparison of several MDO architectures on a variety of simple problems [37]. It also flattened the learning curve for the implementation of MDO methods, allowing problems to be solved without algorithms being explicitly programmed. The monolithic architectures performed robustly on most problems implemented in the framework, and two of the bi level architectures, CSSO and CO, were tested on
simple problems. There were a few shortcomings, however, to this initial implementation. While CO and CSSO appeared to converge for simple problems, they broke down when the number of design variables was increased beyond 5 or 10 variables. Additionally, the quadratic response surface implemented scaled poorly with increasing complexity and adversely affected the performance of CSSO. The Collaborative Optimization implementation, while not dependent on response surfaces, also scaled extremely poorly with increasing complexity, and failed to converge for problems with more than a few variables. Finally, though attempts were made to implement versions of the BLISS architecture, they were not successful. The opportunity also existed to improve πMDO in a number of areas, including the incorporation of alternate optimizers, response surfaces, and sensitivity analysis methods. The addition of more automated data logging and performance measures also had the potential to allow more complete comparisons between architectures.

3.2 pyOpt

One limitation on the initial version of πMDO was the availability of only one optimizer, SNOPT [2]. To improve the utility and the accessibility of πMDO, the integration of many optimizers with different algorithms was required. Optimization frameworks with similar philosophies to πMDO had been developed before, for example the DAKOTA [38] framework developed by Sandia National Laboratories, but none exactly suited the requirements of πMDO. Concurrent with the development of πMDO, the pyACDT framework was being developed for aircraft conceptual design [32], and also required access to a suite of optimizers with a common interface. Due to language commonality, and the ability to influence the development of this suite of optimizers, the new pyOpt framework was integrated into πMDO.

PyOpt provides an object oriented common interface to a variety of third party optimization algorithms. Similar to the structure of the OptProb class in πMDO, pyOpt contains a base class for optimization problems, Optimization, which holds all the information required to perform a single discipline optimization. This is inherited by the Optimizer and Solution objects which represent various opti-
mizers, or a mechanism for storing the solution of each optimization. pyOptimization currently contains several gradient based and gradient free optimizers, and with the common interface, as more optimizers are added to the pyOptimization suite, they become immediately available to πMDO. Especially important for the eventual goals of πMDO, pyOptimization contains several open source optimizers, for example pySLSQP [39], that can be freely distributed.

In order to interface with the pyOptimization framework, it was necessary to modify the equivalent levels of code in πMDO. In this regard, the OptProb level saw the bulk of the changes, with a few modifications to the call structure of each architecture within πMDO.

3.2.1 pyOpt interface

The original πMDO implementation combined initialization of the optimizer with problem setup and solution in a single routine called optimize. pyOpt decouples these two steps, reducing the computational overhead and allowing for time saving modifications to be made to most architectures. To maintain compatibility with problems written for the earlier version, the optimize function is maintained, but augmented with routines to initialize a pyOpt optimization problem, set_prob, and initialize an optimizer init_optimizer.

set_prob

This routine translates the variable, objective, and constraint information contained within the OptProb class of πMDO to the standard used by pyOpt. πMDO was originally designed around the SnoptB format [2], which calculates the objective function and constraints in two separate steps. The majority of optimizers available use a different call structure, as typified by the SnoptC format, so pyOpt adopts this standard. In the new call structure, the objective and constraints are calculated simultaneously in a single routine. This can greatly reduce sensitivity calculation times using finite difference as one perturbation affects both the objective and constraints and therefore eliminates redundant function calls. set_prob translates the old format
into the pyOpt format by using a dummy function to call the objective and constraints in sequence. An empty function, `fun_comb`, was also added to the `OptProb` class that uses the pyOpt format that to allow users or architectures to operate using the new structure. The output of the `set_prob` function is a persistent optimization problem instance compatible with pyOpt stored as `OptProb.opt_instance`. When passed to a properly initialized optimizer instance, this problem is optimized and the solution is returned.

`init_optimizer`

When given the name of the desired optimizer, this function imports the properly wrapped optimizer and initializes it with certain default settings. Because of the common standard for the pyOpt interface, no changes are required to the problem when switching between optimizers. πMDO does customize some settings between optimizers, in particular modifying the file output routines of SNOPT and adjusting convergence tolerances, etc. `init_optimizer` passes back an instance of the optimizer which is not associated with a particular optimization problem and will solve whatever problem is passed to it. Exploiting this decoupling, the user can modify the variable values of an existing problem then pass it back to the optimizer to be solved again. This becomes very valuable for particle based optimization algorithms, design space surveys, and multi-objective optimization.

### 3.2.2 Modifications to existing architectures

Due to the change in input formats to maintain commonality with pyOpt, several modifications to increase the performance of the existing architectures became possible. For the MDF architecture, an MDA was being performed for the calculation of both the objective and constraints. This MDA was unnecessary with the new structure, and was eliminated when the objective and constraint functions were combined. This significantly reduced the computational cost of both the MDF and MDF adjoint architectures. A similar change was applied to the CO architecture, also reducing the number of function calls expended for sensitivity analysis. For any of the multilevel
architectures, adoption of the pyOpt format reduced the amount of computational overhead significantly. The original version of πMDO would initialize an instance of SNOPT every time a sub-optimization was performed, leading to potentially thousands of unnecessary initializations. The new format initializes one version of the optimizer for each discipline sub-optimization, then simply modifies the optimization problem instance being passed to it to perform sub-optimization at each design point.

3.3 pyMMT

The pyMMT class contains a variety of approximation models and related tools to allow computationally inexpensive representations of the functions within MDO problems. The major components of this class are a design of experiments module, DOE, and a meta model module, MMT. A simplified UML diagram of these modules is given in Figure 3.2 below.

The procedure for initializing a meta model is to first use the design of experiments module to select a cloud of points spanning the design space. This distributed set of design points is then passed to the meta model class along with the function of interest to be modeled. Using a variety of approximation methods, from kriging to classic least squares fits, the meta model routine then builds an approximation of the function over the given design space. Once created, this meta model provides computationally inexpensive function evaluations through the eval point method. Based on the measured accuracy of the approximation, the meta model can also be regenerated while reusing as many points as possible. This is achieved by calling the __init__ routine and also providing the old meta model instance.

Provision is made within πMDO to call the meta modeling routines either from within specific architectures, such as CSSO, or from the OptProb level directly for single discipline problems. The specific design of experiment and meta model are chosen either by specifically calling that class, or by the use of attribute strings at the architecture level. CSSO is the primary architecture within πMDO that uses the meta model classes. Within CSSO, the specific meta model can be selected through the CSSO.meta_choice string, as well as a boolean dictating whether or not to use
Figure 3.2: Simplified UML diagram of piMDO’s interaction with the pyMMT and pyDOE meta modeling packages.

BFGS response surface updates CSSO.meta model.BFGS. The meta model class also supports Thomson’s progressive validity trust region optimization \[34\] to control the bounds of the design space and increase the performance of the meta models.

Experimenting with single discipline optimization problems through the OptProb class, Thomson found that the polynomial regression model was the fastest method as the dimensionality of the problem increased \[33\]. Polynomial regression combined with the progressive validity trust region methods is therefore set as the default model for the CSSO architecture. The ability to select any model within pyMMT is retained to allow comparison between methods.
3.4 Collaborative Optimization (CO)

Several variations of the Collaborative Optimization architecture exist. Two versions of the CO2 architecture with a quadratic objective function were implemented, differing in their handling of the inter-disciplinary compatibility constraints. The first version was the “standard” CO2 implementation with the compatibility constraints handled as equality constraints at the system level. This version was originally implemented by Tedford [27] and converged for small scale problems, but was found to quickly degenerate for larger numbers of variables. The source of this degeneracy was traced to an error in assigning design variables and system targets to the respective sub-system optimizations. As discussed in the results section, when this deficiency was fixed the architecture achieved convergence, but was very slow and required many more function calls than the monolithic architectures.

The second version, based upon investigations by Perez [15], implements these constraints as inequality constraints subject to some user defined error tolerance. The second version trades slight inaccuracy of the optimum result for faster convergence, and the amount of inaccuracy can be adjusted by altering the error tolerance. A UML diagram showing the implementation of the CO architecture in πMDO is shown in Figure 3.3.

![UML diagram for CO class](image)

Figure 3.3: UML diagram for CO class

To modify the equality constraints of the original formulation, a new attribute, `self.J_tol`, was added to the CO class to allow the user to specify a tolerance for the compatibility constraints, $J$. Based on the work of Perez [15] and some investigation
of the trade off between convergence time and accuracy of the final solution, this constraint is set to $1 \times 10^{-7}$ by default. Based upon engineering intuition, the user can set this tolerance to a suitable value for each discipline.

```python
__.calc_con
```

This function serves as the constraint calculation for the system level optimization. Initially, it populates the sub-optimization problems with the current system level design and coupling variables. Then each discipline optimization is converged to an optimal solution. Currently this is done sequentially, but is an area where parallel processing would be relatively easy to implement. After the sub-optimizations have converged, the value of their objective function is the system level compatibility constraint for this design point. Depending on which version of CO is being run, the constraint values are either returned directly to the system level, or the J_tol error tolerance is applied before they are returned.

```python
__.calc_con_sensitivity
```

__.calc_con_sensitivity calculates the sensitivity of the global and compatibility constraints for the system level problem. Because the exact formula for the compatibility constraints is known, their sensitivities are calculated analytically through post optimality sensitivity analysis. The global constraint sensitivities are first calculated through finite differencing, complex step, or user provided derivatives. Then the post-optimality sensitivities for each discipline are calculated analytically. The post-optimal sensitivities are appended to the constraint sensitivity array and returned to the system level optimizer.

```python
__.eval_objective
```

This method serves as the objective function for the sub-system optimization problems. It takes design variable inputs from the optimizer but also from design variable targets set at the system level. It performs a local discipline analysis based on the self.current_discipline flag and retrieves the relevant discipline output.
Using the locally calculated discipline outputs and the non-local coupling and design variable information provided by the sub-system optimizer, the value of the compatibility constraint is calculated. Note that the system level targets are held constant through this process and only used in the calculation of the discipline compatibility constraint. This value is then returned to the optimizer as the objective function value at that design point.

```python
_eval_constraints
```

`_eval_constraints` calculates the value of the local discipline constraints at a given design point. Before these are calculated, it performs a discipline analysis based on the `self.current_discipline` flag to update the local coupling variables. The calculated constraint values are then passed to the sub-system optimizer for the relevant discipline.

### 3.5 Concurrent Subspace Optimization (CSSO)

A UML diagram showing the implementation of the CSSO architecture in πMDO is shown in Figure 3.4. The CSSO class inherits from `MdoProb` and contains unique functions for the subsystem analyses and constraints.

```
CSSO(MdoProb)
- dvar_groups:Dict
- cp_groups:Dict
- obj_discipline:String
- mpi:Bool

init(dvar_groups,cp_groups,obj_discipline)
- optimize()
- eval_objective(vars)
- calc_con(vars)
- eval_objective(vars)
- eval_constraints(vars)
```

Figure 3.4: UML diagram for CSSO class

The CSSO algorithm also instantiates the `MetaApprox` class to create an approximation model and query values from it.
Section 3.5. Concurrent Subspace Optimization (CSSO)

__eval_objective__

This function is a hidden function contained within the CSSO class which is used as the objective for the subsystem optimization problems. Based on the flag `self.current_discipline` set at the system level, `__eval_objective` calculates the global objective function. Local information from the current discipline is used as well as approximate values of the non-local coupling variables obtained from the meta model approximation.

__eval_constraints__

Like the subsystem objective function, this method calculates the local constraints for a given discipline. The local discipline is indicated by the `self.current_discipline` flag set at the system level. The local constraints are evaluated using local and locally controlled global design variables with non-local information being queried from the approximation model.

optimize

The optimize method for CSSO first restructures the problem into local and discipline level OptProb instances. The design variables are distributed according to their influence on the subsystem optimization problems. Global design variables that affect more than one discipline are distributed to the discipline upon which they have the greatest effect. This influence can be user provided through the `vars.influence` attribute at the Variable level. For example, if variable $z_1$ has a much larger impact on the value of discipline 1 than discipline 2, `vars["x1"].influence = "Discipline 1"`. These influence factors can also be determined automatically through the use of cross-coupling sensitivity derivatives. Once the design variables have been grouped into their subsystem optimizations, they are initialized with `__eval_objective` and `__eval_constraints` as their objective and constraints respectively. Once all the function and variables are properly assigned and initialized, the actual CSSO algorithm begins. Initially, the response surface or meta model approximation is populated by performing multidisciplinary analyses (MDA)s at a series of design points.
These points are selected according to the design of experiments module available within pyMMT. Once the meta model has been initialized and verified, it is used to provide non-local coupling variable information to each of the discipline level sub-optimizations in turn. The objective value for each of these sub-optimizations is compared, and the design point with the best value is used as the starting point for the system level optimization. An MDA is also performed at the best design point and the resulting information is used to retrain the meta model. Once the system level optimization has converged, the accuracy of the meta model is checked against the exact objective function at that point. If accuracy is outside of a certain tolerance, the meta model is rebuilt until that tolerance is achieved. Finally, the objective function value at the system level optimum is compared to previous iterations. If the accuracy of the meta model is acceptable and the change between successive iterations is below the convergence tolerance, the algorithm terminates.

3.6 Bi-level Integrated System Synthesis (BLISS)

Due to time constraints and performance difficulties with the meta models within πMDO, a functioning version of BLISS was not achieved. The primary issue was with the poor scaling of the meta models implemented within pyMMT. Experience with CSSO, which uses only a single meta model instance, showed that performance was extremely poor for problems of more than a few variables. Due to the structure of BLISS, which uses one meta model for the system level and for each of the disciplines, this scaling with dimensionality would be much worse. Additionally, the introduction of weighting variables further exascerbates this effect by increasing the dimensionality of each approximation. As the maturity of the approximation methods within pyMMT improves, efforts should be made to implement and test the BLISS algorithm against the other architectures. Particularly if parallelization of the meta models is implemented, it may be competitive with the other bi-level architectures.
Section 3.7. Semi-Analytic Sensitivity Analysis

As for all the components of $\pi$MDO, the semi-analytic sensitivity functionality was programmed in an object oriented fashion to maximize code reuse and flexibility. A UML diagram showing the structure of the semi-analytic methods is given in Figure 3.5.

Exploiting the object oriented nature of $\pi$MDO, the semi-analytic methods work by overwriting the $\text{obj}_\text{sensitivity}$ and $\text{con}_\text{sensitivity}$ methods at the $\text{OptProb}$ level with their own version of a semi-analytic sensitivity routine using the common methods within the $\text{MdoProb}$ level.

For the specific MDF case, the intention is to provide sensitivities of the objective and constraint functions with respect to the design variables in an identical format to that produced by SNOPT’s finite differencing. To this effect the method at the
MDF level calls the disciplinepartials function which steps through disciplines retrieving the required partial derivatives. After this, the assemble_adjointmatrix method creates the $\frac{\partial R}{\partial y}$ and $\frac{\partial R}{\partial x}$ matrices. Finally, the function at the MDF level solves for either the adjoint or direct vector and uses it to calculate the total sensitivities. Descriptions of the major components are given below.

### 3.7.1 OptProb level

At the OptProb level, the methods used for the semi-analytic sensitivities are obj_sensitivity and constraint_sensitivity. These methods are designed to be overwritten to pass user provided sensitivities directly to the optimizer. Both expect array inputs in the same format used by SNOPT [2] for its objective and constraint gradients. Due to the object oriented nature of πMDO, one or both of these methods can be overwritten to pass the semi-analytic total sensitivities directly to the Optimization class and pySNOPT.

### 3.7.2 MdoProb level

At the MdoProb level, several functions common to the semi-analytic sensitivity methods are encapsulated. These are shown in the UML diagram of the MdoProb class.
in Figure 3.7. As MdoProb is inherited by all the MDO architectures implemented within πMDO, these basic functions are available for use by any of these architectures. They comprise functions to gather the partial derivatives required by the semi-analytic methods and assemble them into properly formatted matrices. Individual functions are described below.

**discipline_partials**

When provided with the vars dictionary for the current design point and the name of desired discipline, **discipline_partials** retrieves or calculates the required partial derivatives for the semi-analytic methods. First the function **getpartials** at the discipline level is called for the given discipline. The expected output from **getpartials** is a dictionary of sensitivities in the same format as the outputs of **discipline_partials**. If this does not return anything, or provides only partial sensitivities, **discipline_partials** calculates the missing entries using the user specified sensitivity method, either finite differencing or complex step.

Based on the user provided **use_residuals** flag for this discipline, the partials are calculated using either the analysis or residual code for the given discipline by calling **calc_partial** or **calc_residuals**. If the residual code is requested but does not exist, the method recovers from the error and sets the **use_residuals** flag to false. In this way **discipline_partials** minimizes the computational effort required, using...
as many user provided (presumably more efficient) sensitivities as possible and using
the most efficient differencing scheme possible (usually residual based).

\texttt{calc\_partial} and \texttt{calc\_partialresid}

Both \texttt{calc\_partial} and \texttt{calc\_partialresid} are simple differencing methods that
can use either complex or real (forward) differencing to calculate the partial deriva-
tives of a function. Given a particular function and input variable, they perturb the
input variable and observe the change in the output variables or residuals of the disci-
pline. \texttt{calc\_partialresid} only differs from \texttt{calc\_partial} in being set up to provide
the \texttt{state\_vars} input required by the residual code. Both methods can handle either
scalar, 1D array, or dictionary inputs and outputs.

\texttt{assemble\_adjointmatrix}

Provided with the partial derivatives from \texttt{discipline\_partials}, this method
creates empty arrays of the proper dimensions for the $\frac{\partial R}{\partial y}$ and $\frac{\partial R}{\partial x}$ matrices. Using a
consistent ordering scheme, \texttt{assemble\_adjointmatrix} fills these arrays with appro-
priate partials and returns the completed arrays. This ordering scheme, based on the
discipline keys, remains constant whether the residual based or residual free formula-
tions of the semi-analytic methods are used. Further, this method is called identically
whether the direct or adjoint method is being used at the architecture level.

\textbf{3.7.3 Architecture Level}

At the architecture level, the common functions from the \texttt{MdoProb} and \texttt{OptProb}
levels are used to calculate total derivatives exploiting the specific mathematical struc-
ture of the architecture. Based on the architecture, there may or may not be an
advantage to using the semi-analytic methods, so implementation in this way allows
specific approaches to be tailored to each situation. To illustrate usage, the specific
implementation for the MDF architecture is shown in Figure , and described below.
Section 3.7. Semi-Analytic Sensitivity Analysis

MDF(MdoProb)

| dvar_groups:Dict |
| cp_groups:Dict |
| obj_discipline:String |

init(dvar_groups,cp_groups,obj_discipline)
optimize()
eval_objective(vars)
calc_con(vars)
adjoint_obj_sens(vars)
constraint_sens(vars,state_vars=False)

Figure 3.8: UML diagram for MDF class

__adjoint_obj_sens

Operating at the architecture level, in this case MDF, this method uses all of the functions at the MdoProb level to assemble the required partial derivatives in the proper format. It then performs the linear solution from equation (2.19) to calculate the adjoint vector, and finally, calculates the total derivatives of the objective function with respect to the design variables. The arrangement of the linear solution, and that used by assemble_adjointmatrix, is carefully selected to output a 1D array of the total sensitivities in the same format as that used by OptProb.calc_sens. In this way, during problem initialization, OptProb.obj_sensitivity can be overwritten by __adjoint_obj_sens without the optimizer being affected. For a single function of interest, the adjoint method will always be more efficient than the direct method for calculating sensitivities. As πMDO is not currently set up to handle multi objective optimization, the adjoint method is always used for the objective sensitivities.

__constraint_sens

This method operates in a very similar fashion to __adjoint_obj_sens by overwriting the constraint_sensitivity method at the OptProb level. __constraint_sens steps through the global and local constraints in the same order as MDF.__calc_con and gathers the required partial derivatives. Again, the ordering scheme is carefully selected to output a 2D array of the total constraint sensitivities in the format expected by the optimizer. As the number of constraints relative to design variables
may vary from problem to problem, the user is given the option of using the adjoint or direct method to calculate the constraint derivatives. If the user does not select one or the other, πMDO estimates the relative expense of each method by comparing the number of design variables and constraint functions. `_constraint_sens` uses the direct method by default.

### 3.7.4 Support for user-provided sensitivities

The semi-analytic sensitivity routines within πMDO make the process of passing in sensitivities both easier and more generally applicable. Rather than having to overwrite a specific `calc_sens` method, the user can simply provide a dictionary of the sensitivities of a function with respect to the design and coupling variables. This query is implemented within the `_calc_partial` and `_calc_partialresid` routines at the `MdoProb` level. In Python format the dictionary is keyed by the discipline name, sensitivity type, the denominator, and finally the numerator of the partial derivatives. This format is as follows,

```python
{'dir': {'x1': {'y1': 3.14, 'y12': 3.45}, 'x2': {'y1': 3.76, 'y12': 2.875}}
```

corresponding to a dictionary of sensitivities for the outputs $y_{11}$ and $y_{12}$ of a given discipline with respect to the design variables $x_1$ and $x_2$. In order from left to right, the entries represent the sensitivities:

$$\frac{\partial y_{11}}{\partial x_1} = 3.14, \quad \frac{\partial y_{12}}{\partial x_1} = 3.45, \quad \frac{\partial y_{11}}{\partial x_2} = 3.76, \quad \frac{\partial y_{12}}{\partial x_2} = 2.875$$  \hspace{1cm} (3.1)

Note that the format of this dictionary may seem counterintuitive, but serves to reduce the workload for the partial derivative calculation scheme. When finite differencing or complex step is performed to find these partials, the design variable is first perturbed, then the effect on the coupling variable is measured. This means that perturbing $x_1$, for example, the partials $\frac{\partial y_{11}}{\partial x_1}$ and $\frac{\partial y_{12}}{\partial x_1}$ are both available for the
same amount of computational effort. Therefore this structure serves to reduce the workload of the \_\_calc\_partial routine.

3.8 Reconfigurability

In aerospace applications of MDO, difficulty is encountered in solving large problems with many disciplines and variable degrees of coupling between these disciplines. Bi-level architectures like Collaborative Optimization perform well for problems with low-fidelity coupling between disciplines and relatively complex discipline level routines. If several disciplines are highly coupled, however, the performance of CO is reduced at a geometric rate. For tightly coupled problems with few disciplines, monolithic architectures such as MDF, especially with semi-analytic sensitivities, perform well.

3.8.1 NEST Architecture

The proposed nested optimization (NEST) architecture combines the attributes of these two architectures to produce a formulation that works for widely distributed problems with clusters of high local coupling dimensionality. In effect, it extends and automates the original philosophy of collaborative optimization, allowing the practitioner to optimize each discipline in the most efficient way.

Structurally, the NEST architecture exploits the object oriented nature of πMDO. As the variables, disciplines, and even architectures within the framework all function as individual entities, they can be reordered and recombined with a minimum of effort. The essential tactic of the NEST architecture is to modify the discipline structure of the original problem by creating new “super” disciplines that contain instances of the MDF architecture. Note that the problem statement input to πMDO requires no changes except for a dictionary indicating which disciplines should be clustered together. In future, this choice can be automated based on ratios of coupling to design variables and interpretation of the global sensitivity equations. A simplified UML diagram demonstrating reconfiguration for a sample problem is shown in Figure 3.9.
The new super disciplines encapsulate an MDF architecture with a slightly modified objective function. This sub-optimization uses the multidisciplinary analysis and constraint calculations from the MDF formulation with a modified objective function taken from CO. The sub-optimization therefore attempts to minimize the discrepancy between its output coupling variables and the system level targets while satisfying multidisciplinary feasibility and discipline level constraints for the sub disciplines. More robust convergence is obtained by using the modified CO formulation with relaxed constraints proposed by Perez [15]. Functionally, this is accomplished by overwriting the `eval_objective` function of MDF with the hidden `__eval_objective` function taken from the CO architecture.

Compared to a normal CO formulation, many coupling variables are removed from the system level problem, speeding convergence. Compared to a monolithic
MDF formulation, there is now autonomy between most discipline analyses, and the fixed point iteration encompasses fewer disciplines and variables. Using the CO subsystem objective function as the objective of the MDF sub-optimizations allows the unmodified post-optimality sensitivity methods from the CO architecture to be used. This speeds implementation and avoids problem dependent calculation of second order derivatives, required when multilevel optimization problems are formulated [40]. Because the new architecture inherits from the MdoProb class, the semi-analytic sensitivity methods are also available to it, and can be used to speed the convergence of the nested MDF.

3.9 Example Problems

Several example problems were used to test the performance of the MDO architectures and sensitivity methods. A full description of the test-suite of MDO problems is available at the πMDO website. Some selected problems are described in more detail below.

3.9.1 Analytic

The Analytic problem was implemented in the initial framework implementation and originally formulated by Sellar et al. [17]. Mathematically, it can be defined as in equation (3.3).

\[
\begin{align*}
\text{minimize} \quad & x_1^2 + z_2 + y_1 + e^{-y_2} \\
\text{with respect to} \quad & z_1, z_2, x_1 \\
\text{subject to} \quad & y_1/3.16 - 1 \geq 0 \\
& 1 - y_2/24 \geq 0 \\
& -10 \leq z_1 \leq 10 \\
& 0 \leq z_2 \leq 10 \\
& 0 \leq x_1 \leq 10 
\end{align*}
\]
where $y_1$ and $y_2$ are the state variables of the two disciplines, and are given by,

\begin{align}
  y_1(z_1, z_2, x_1, y_2) &= z_1^2 + x_1 + z_2 - 0.2y_2 \quad (3.3) \\
  y_2(z_1, z_2, y_1) &= \sqrt{y_1} + z_1 + z_2. \quad (3.4)
\end{align}

The Analytic example represents one of the simplest practical multidisciplinary problems with which to test the framework, while maintaining the characteristics of much larger problems. There are two disciplines, each with one state variable, $y_1$ and $y_2$ that is also a coupling variable. There are two global design variables, $z_1$ and $z_2$ and one local design variable for discipline 1, $x_1$. The problem exhibits nonlinear coupling between disciplines as well as within the objective function. There is one local constraint associated with each discipline. With the divisors of the constraint functions formulated as 3.16 and 24, the global optimum of this problem is $(z_1, z_2, x_1) = (1.9776, 0, 0)$ \cite{41}, where the objective value is 3.18339 and Discipline 1’s constraint is active. The Analytic example is very useful when proving new concepts within πMDO as it is quick to converge and has a known (analytic) optimum to compare results. One caveat is that the problem is so simple that it may not reveal bugs which will only become apparent with more complex implementations.

### 3.9.2 Scalable Dimensional

The scalable dimensional problem is an extension of the scalable problem \cite{1} \cite{37} to allow greater flexibility in setting the problem attributes. It can generate a multidisciplinary problem of an arbitrary dimensionality in number of disciplines, global and local design variables, and coupling variables. The coupling between disciplines can be set by the user, or randomly assigned based on the degree of interdependence desired. To allow the investigation of non-uniform dimensionality in problems, the number of local design and output coupling variables can also be randomly assigned or set by the user. The mathematical objective, analysis, and constraint functions can all be modified and global constraints can be added if the user requires them. Finally, the computational expense of the discipline analysis with respect to the residual codes
can also be modified to test the efficiency of residual based MDO architectures.

The default instance of the scalable problem has a quadratic objective function, linear discipline analyses, and linear constraints for each discipline. A mathematical representation of one possible version of the scalable dimensional problem is given in equations (3.5) and (3.6). It has three disciplines that are fully coupled (i.e each discipline has the outputs from all other disciplines in its inputs).

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^{3} z_i^2 + \sum_{i=1}^{3} y_i^2 \\
\text{with respect to} & \quad z, x_1, x_2, x_3, \\
\end{align*}
\]

(3.5)

where \( y_i \) represents the output coupling variables given by

\[
\begin{align*}
y_1(z, x_1, y_2, y_3) &= -\frac{1}{C_{y_1}} (C_{z_1} z + C_{x_1} x_1 - C_{y_2} y_2 - C_{y_3} y_3) \\
y_2(z, x_2, y_1, y_3) &= -\frac{1}{C_{y_2}} (C_{z_2} z + C_{x_2} x_2 - C_{y_1} y_1 - C_{y_3} y_3) \\
y_3(z, x_3, y_1, y_2) &= -\frac{1}{C_{y_3}} (C_{z_3} z + C_{x_3} x_3 - C_{y_1} y_1 - C_{y_2} y_2) \\
\end{align*}
\]

(3.6)

s.t \quad 1 - \frac{y_1}{C_1} \leq 0 \quad 1 - \frac{y_2}{C_2} \leq 0 \quad 1 - \frac{y_3}{C_3} \leq 0,

When the scalable problem is first initialized, it generates random coefficients for the discipline analyses and constraints. These coefficients and the problem attributes are stored so successive trials can be run with the identical setup. Finally, the results of each optimization are stored in a series of log files to allow analysis and comparison of the results.
Chapter 4

Results and Discussion

4.1 pyOpt

The implementation of the new optimizer class and common problem structure dramatically improved the performance of several of the architectures within the πMDO framework. While the performance of CO, CSSO, and the semi-analytic sensitivity version of MDF are examined in greater detail later, it is worth comparing the general performance gain from interfacing with the pyOptimization framework.

As the comparison in table 4.1 between each method shows, most architectures display a decrease in function calls and overall convergence time. A version of the scalable problem with 5 disciplines, 96 design variables, and 21 coupling variables was optimized for six different architectures with complex step sensitivities. The same problem was then optimized with the pyOptimization implementations of each architecture, and the results were collated. Note that pySNOPT \cite{42} was the optimizer in both cases, though the pyOptimization trial uses the SnoptC rather than SnoptB format \cite{2}.

The small increase in function calls for the IDF and SAND architectures is explained by the increase in computational overhead incurred by translating the problem at the OptProb level to match the pyOptimization format. In future, this effect could be reduced by harmonizing the underlying problem format between πMDO and pyOptimization. Interfacing with pyOptimization, however, allowed redundant function
### Table 4.1: Comparison between original SNOPT and pyOptimization SNOPT implementations for the scalable problem.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Version</th>
<th>Function Calls</th>
<th>Convergence Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDF</td>
<td>pyOpt</td>
<td>11,225</td>
<td>3.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11,235</td>
<td>3.65</td>
</tr>
<tr>
<td></td>
<td>pyOpt</td>
<td>11,815</td>
<td>4.57</td>
</tr>
<tr>
<td>SAND</td>
<td>pyOpt</td>
<td>11,805</td>
<td>4.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11,815</td>
<td>104.4%</td>
</tr>
<tr>
<td></td>
<td>pyOpt</td>
<td>313,450</td>
<td>88.22</td>
</tr>
<tr>
<td>MDF</td>
<td>pyOpt</td>
<td>13,470</td>
<td>4.12</td>
</tr>
<tr>
<td>MDF Adjoint</td>
<td>pyOpt</td>
<td>7,445</td>
<td>67.9%</td>
</tr>
<tr>
<td>CO original</td>
<td>pyOpt</td>
<td>39,895</td>
<td>9.57</td>
</tr>
<tr>
<td>CO const. red.</td>
<td>pyOpt</td>
<td>20,662</td>
<td>8.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13,587</td>
<td>88.3%</td>
</tr>
</tbody>
</table>

The difference in convergence speed can be partially explained by the file output routine for SLSQP as opposed to SNOPT. Within \(\pi\)MDO, SNOPT only records the convergence data for system level objective evaluations. Currently, SLSQP records this data at every function call, including those required for finite differencing, resulting in many more file writing operations. This issue is being remedied as SLSQP, and all the optimizers within pyOpt, are more tightly integrated into \(\pi\)MDO. As more
optimizers are added to the pyOpt framework, more comprehensive comparisons can be performed and the best optimizer can be selected. Currently, SNOPT is the default optimizer within πMDO, and SLSQP is the default open source optimizer for the public release.

Figure 4.1: Convergence plot of the SNOPT and SLSQP optimizers on the scalable problem with the IDF architecture.

4.2 Collaborative Optimization (CO)

The original CO implementation in πMDO converged for small problems of a few design variables [27]. This implementation, however, broke down and would not converge when the dimensionality of the problem was increased. This degeneracy was traced to inadequate freedom of the sub-system level design space caused by over constraining the sub-system problem. Greater sub-space freedom was produced by granting the sub-system level optimizations authority over the non-local coupling variables, while maintaining inter-disciplinary feasibility through the system level
compatibility constraints. This modification produced a CO2 algorithm which would converge or come very close to the optimum for much larger problems with several hundred design variables.

Even with the modifications this CO implementation was extremely expensive in terms of function calls and optimization time, even for problems where bi-level architectures would be assumed to have an advantage. The original implementation was dramatically slower than even the MDF architecture and did not scale well with increasing complexity when tested on the scalable problem. Further, though the final design point would agree with the optimum achieved by the other architectures, the algorithm would rarely detect a successful convergence. In the majority of cases, due to the degeneracy issues discussed in chapter two, the optimizer would get extremely close to the converged solution but exit due to numerical infeasibility or by hitting the iteration limit. This was extremely inefficient as the algorithm spent the majority of the computational effort trying to bridge the very small improvements around the optimum. Additionally, the practitioner was not given confidence that a legitimate converged solution had been reached as the exit message from the optimizer usually complained of infeasible solutions. Without the optimal results of other architectures to compare the solution to, the user had no confidence that the optimum had in fact been reached, which made the potentially faster solution from CO unreliable.

4.2.1 New Implementation

After the constraint relaxation proposed by Perez [15] had been implemented on the CO2 algorithm, the degeneracy issues near the optimum were avoided by allowing the optimizer to converge before reaching the numerically treacherous regions. Significantly, by modifying the feasibility tolerance $J_{\text{tol}}$ the precision of the final optimum could be traded against faster convergence. This tolerance could be modified for each discipline individually, or for the system as a whole. To measure the effect of changes in the constraint relaxation tolerance, $J_{\text{tol}}$, a study was performed using the Analytic example. As shown in Figure 4.2, the convergence time dramatically increases as the constraint tolerance is tightened.
When the function calls are included in this study, the result is even more dramatic. When the optimum $J_{\text{tol}} = 1 \times 10^{-7}$ is selected, 4462 function calls are required for an accuracy of $6.8 \times 10^{-6}$. Decreasing the tolerance to $= 1 \times 10^{-13}$ increases the function calls to 27428 without any appreciable increase in the accuracy of the final solution. A plot of the effects of $J_{\text{tol}}$ on function calls is given in Figure 4.3. Clearly there exists an optimum trade off between accuracy of the final solution and computational expense, but this may vary depending on the problem. This is also likely influenced by the constraint tolerance of the optimizer, in this case $= 1 \times 10^{-6}$. Based on the investigations of Perez, and the studies performed here, a default parameter of $= 1 \times 10^{-7}$ was selected for πMDO. The user has the ability, however, to vary this tolerance both globally, and for individual disciplines.

A convergence plot of the original and modified implementations of CO is shown in Figure 4.4 using a $J_{\text{tol}}$ of $1 \times 10^{-7}$.

The convergence is shown for a version of the scalable problem with 5 disciplines, 5 global design variables, 90 local and 10 coupling output variables per discipline. In this particular case the original method could only converge the solution to within $1 \times 10^{-3}$ of the MDF optimum objective value. The constraint relaxation method resulted in a relative error of only $1 \times 10^{-14}$, almost machine zero.

### 4.2.2 Scaling

To test the performance of collaborative optimization against the existing monolithic architectures, a series of test cases was generated using the scalable problem described in Section (3.5). The basic problem consisted of 5 disciplines with 5 global design variables and an arbitrary number of local design and output coupling variables per discipline. In the first study, shown in Figure 4.5 the number of local design variables was held constant while the coupling variables increased. The problem had 5 global design variables and 10 coupling variables per discipline.

The implementation of constraint relaxation clearly improves the performance of the CO architecture, not only in terms of performance, but in the consistency of the results. As the plot shows, the new CO implementation converges faster than MDF in
Section 4.2. Collaborative Optimization (CO)

Figure 4.2: Effects of constraint tolerance on convergence time and error for the modified CO architecture. For the Analytic problem tested, the optimum $J_{tol}$ value is around $1 \times 10^{-7}$.

Figure 4.3: Effects of constraint tolerance on function calls and objective function error for the modified CO architecture on the Analytic problem.
all cases and exhibits smooth and predictable scaling with increasing design variables. The original version of CO, even with the performance benefits of the pyOpt suite, performs inconsistently and often slower than MDF. Further, only successful trials are plotted in Figure 4.5 and the original CO implementation fails to converge within $1 \times 10^{-4}$ of the optimum in a large percentage of trials. This is shown in table 4.2.2, where the original CO implementation would fail up to 70% of the time.

The same trial was run again, but with the coupling between disciplines allowed to randomly vary. Each discipline was coupled to a minimum of 2 other disciplines, but could be connected to anywhere between 2 and 4 other disciplines, altering the variable flow of the problem. The plot of the randomly coupled scalable problem for an increasing number of design variables is shown in Figure 4.6. Because the coupling is random, the results are fairly noisy, but the constraint relaxation formulation clearly converges faster, and more consistently, than the original version. The CO architec-
Section 4.2. Collaborative Optimization (CO)

Figure 4.5: Performance of the original and constraint relaxation CO architectures for increasing dimensionality of design variables.

<table>
<thead>
<tr>
<th>Case</th>
<th>No. Trials</th>
<th>IDF</th>
<th>SAND</th>
<th>CO C.R.</th>
<th>CO Orig.</th>
<th>MDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Coupling</td>
<td>81</td>
<td>4</td>
<td>10</td>
<td>5</td>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>Failure Rate</td>
<td>4.9%</td>
<td>12.3%</td>
<td>6.2%</td>
<td>70.4%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Random Coupling</td>
<td>53</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>Failure Rate</td>
<td>7.5%</td>
<td>7.5%</td>
<td>7.5%</td>
<td>64.2%</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Failure rates for various architectures on the scalable problem. Trials were considered failures if they did not converge to within $10^{-4}$ of the objective value at the optimum.
Section 4.2. Collaborative Optimization (CO)

ture should theoretically perform well on problems with varying coupling like this, as it is a decentralized approach where computational effort can be expended only where required. The implementation of constraint relaxation fixes the convergence difficulties of the original approach and allows this advantage to become clear.

As an inherent function of its design, the CO architecture does not perform very well for problems with high levels of inter-disciplinary coupling. This effect can be clearly seen in Figure 4.7, where a series of trials was performed with an increasing number of coupling variables for each discipline. There was full coupling between disciplines in this case, meaning each discipline required the outputs of all other disciplines as input variables. In this case, the performance of the original and new CO formulations was very similar, with both scaling poorly for increasing coupling.

Overall, the CO architecture with constraint relaxation represents a significant improvement over the original implementation, both in terms of performance and robustness. The introduction of inequality constraints and a user defined error tol-
erance provides greater freedom to the optimizer, smoothing the design space and allowing faster convergence. With appropriate selection of the error tolerance, the performance of the architecture can be greatly improved, both in terms of function calls and convergence time.

4.3 Concurrent Subspace Optimization (CSSO)

By integrating πMDO with pyMMT, a variety of meta models are available to the CSSO architecture. Due to the common framework, these meta models can be varied during the problem setup phase by specifying which model to use with a simple input string. To measure the performance of the CSSO architecture, it was compared with the MDF architecture using three different meta models.

First, to illustrate the importance of limiting the bounds of the design space for meta model approximations, a study was performed using the scalable problem. A
problem with three disciplines and two design variables per discipline was optimized over an increasing design space. The results are shown in Figure 4.8 below.

![Figure 4.8: Performance of the CSSO architecture for an increasing design space. The response surface bounds were increased and the resulting performance degradation was measured.](image)

As the bounds of the response surface were increased, the approximation becomes poorer and requires more function evaluations. This also increases the convergence time, and highlights the importance of limiting the bounds of the approximation model to maintain the performance of the CSSO architecture. In practice, a trust region method is used to improve the performance of this architecture by constructing a bounding box over which the approximation is modeled. As the optimizer moves within the limits of the bounding box, it is moved, expanded, or shrunk as necessary to maintain the accuracy of the meta model in the local region. The trust region method used in this implementation of CSSO is the progressive validity trust region method proposed by Thomson [33] and integrated into the pyMMT modeling suite. With the trust region method in place, the various meta models were tested using the analytic problem, and the results are presented in table 4.3 and Figure.

The function calls and convergence times between meta models indicate some
Section 4.3. Concurrent Subspace Optimization (CSSO)

Table 4.3: Function call comparison for CSSO with various approximation models. Note the increased computational overhead (convergence time) for a similar amount of function calls between CSSO and MDF.

<table>
<thead>
<tr>
<th>Case Architecture</th>
<th>Meta Model</th>
<th>Function Calls</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analytic Problem</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSSO</td>
<td>Kriging</td>
<td>1241 1247 - -</td>
<td>48.04 s</td>
</tr>
<tr>
<td></td>
<td>Polynomial</td>
<td>1457 1439 - -</td>
<td>2.66 s</td>
</tr>
<tr>
<td></td>
<td>RBF</td>
<td>934 1646 - -</td>
<td>9.37 s</td>
</tr>
<tr>
<td>MDF</td>
<td></td>
<td>362 362 - -</td>
<td>0.064 s</td>
</tr>
<tr>
<td><strong>Scalable Problem</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSSO</td>
<td>Kriging</td>
<td>- - - - -</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Polynomial</td>
<td>8491 8385 8437 -</td>
<td>148.63 s</td>
</tr>
<tr>
<td></td>
<td>RBF</td>
<td>- - - - -</td>
<td>-</td>
</tr>
<tr>
<td>MDF</td>
<td></td>
<td>7797 12011 13871 12941</td>
<td>4.55 s</td>
</tr>
</tbody>
</table>

interesting trends. Of the three methods, polynomial regression allowed for the fastest convergence and had the lowest computational overhead with an average of 0.00092 s/function call. Kriging had the slowest convergence time, but required fewer function evaluations. The computational overhead for kriging, at 0.019 s / function call, was much higher than polynomial regression, but for situations with very slow discipline analyses, the overhead may be justified. The radial basis function method showed performance in the middle of the range, requiring slightly more function calls than polynomial regression, but converging much faster than kriging. The convergence of these three meta models when combined with the CSSO architecture is given in Figure 4.9.

The plot clearly shows the speed difference between the three methods. Note that even for this simple problem with only three design variables and two coupling variables, CSSO is not competitive with any of the other architectures. MDF is typically the slowest architecture to converge, but it is barely visible on the left hand axis of this plot. Also visible in this plot is the relative speed of each of the meta
models - Notably, the higher overhead and lower speed of kriging and radial basis functions made them impractical for the second trial on the scalable problem. For this trial, the scalable problem with three disciplines, a total of nine design variables, and six coupling variables was optimized. Even for such a relatively simple problem, the meta models began to break down. Only the polynomial regression version of CSSO would converge, and the convergence time was orders of magnitude greater than MDF.

One caveat is that to maintain a common basis between architectures, all of these trials were performed using a single processor. In practice, the use of parallel processing would dramatically improve the convergence speed of CSSO, and is really the advantage of using meta models in the first place. The pyMMT module is capable of running in parallel through MPI XXX REF MPI. Even with this speed increase, however, it remains to be seen if CSSO can be a competitive architecture. In tests on the scalable problem using the fastest of the meta models, polynomial regression,
to model quadratic analysis functions, CSSO still broke down for more than half a
dozan design variables. Without more mature approximation methods, or the use of
parallel processing, CSSO

This does, however, prove the implementation of approximation methods with a
common interface within the $\pi$MDO framework. In future, new modeling techniques
implemented within this standard can be quickly tested on the full suite of $\pi$MDO
test problems. Further, the availability of meta models allows the development and
implementation of response surface based MDO architectures, especially BLISS and
CO with response surfaces.

4.4 Semi-analytic Sensitivity Analysis

4.4.1 Validation

To validate the sensitivities calculated by the semi-analytic methods, they were
compared to the results achieved by complex step and finite differencing for a rep-
resentative problem. Table 4.4.1 shows some representative calculated sensitivities
for the scalable problem and the agreement between the original and semi-analytic
results. In this case, the complex step sensitivities calculated over the whole problem
were used as the exact reference. With a sufficiently small step size, i.e the $1E - 20$
$\pi$MDO uses as a default, these can be considered accurate to machine zero [24] and
are equivalent to those obtained by automatic differentiation [25]. The semi-analytic
results were found to be accurate to within $1E - 14$ when the complex step method
was used to calculate the partial derivatives and $1E - 7$ when finite differencing was
used. This greater error in the finite difference results is most likely the result of
less accurate partial derivatives due to subtractive cancellation. For this particular
problem, the residual based (GSE1) and residual free (GSE2) methods had the same
order of accuracy but this may vary depending on problem implementation and the
underlying residual codes.

One interesting note is that the accuracy of the final sensitivities for the MDF
architecture is heavily dependent on the solution of the multidisciplinary solution
Section 4.4. Semi-analytic Sensitivity Analysis

<table>
<thead>
<tr>
<th>Sensitivity</th>
<th>Original Complex Step</th>
<th>Semi-Analytic Complex Step</th>
<th>Finite Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{df}{dz_1} )</td>
<td>1.5011619793190241</td>
<td>1.5011619793190372</td>
<td>1.5011618970987384</td>
</tr>
<tr>
<td>( \frac{df}{dz_2} )</td>
<td>8.8643743255674643</td>
<td>8.8643743255674483</td>
<td>8.8643745004156376</td>
</tr>
<tr>
<td>( \frac{df}{dx_1} )</td>
<td>8.3048445951781069</td>
<td>8.3048445951781247</td>
<td>8.3048452336145004</td>
</tr>
<tr>
<td>( \frac{dc}{dz_1} )</td>
<td>0.0346907368144533</td>
<td>0.0346907368144535</td>
<td>0.0346907368695189</td>
</tr>
<tr>
<td>( \frac{dc}{dx_1} )</td>
<td>-0.0693803453125928</td>
<td>-0.0693803453125925</td>
<td>-0.0693803456558495</td>
</tr>
<tr>
<td>( \frac{dc}{dx_2} )</td>
<td>0.0526057938095800</td>
<td>0.0526057938095803</td>
<td>0.0526057937144702</td>
</tr>
</tbody>
</table>

Max Difference: 1.8e - 015, 7.68e - 008

Table 4.4: Sensitivity Validation: Selected Semi-Analytic Sensitivities calculated for the Scalable Problem

(MDA) at each iteration. If the individual discipline analyses or residual codes are not converged tightly enough, the final calculated sensitivities are significantly affected. To counter this effect, a residual convergence check was added to the MdoProb.mda method to ensure individual residuals are less than \( 1E - 14 \) before the MDA exits. It is left to the practitioner to adjust this convergence tolerance and trade the accuracy of final sensitivities against the computational cost of more MDA iterations.

The relative performance of the direct and adjoint methods are problem dependent and directly related to the number of design variables and constraints of the input problem. A study was performed to measure the performance of the adjoint and direct methods for different ratios of design variables to constraints. As \( \pi \)MDO is presently set up to handle only single objective optimization problems, the adjoint method will clearly be the preferred choice for the objective sensitivities. Depending on the number of constraint calculations and input design variables, the faster method, whether adjoint or direct, is automatically selected for the constraint sensitivities. This can also be determined manually by the user. A series of trials was run on a scalable mathematical problem with varying numbers of constraints and design variables. The results were calculated using the GSE2 and Adjoint2 methods (residual free), and are displayed in Figure 4.10

Based on this study, the constraint sensitivity routines were modified to automatically select the most efficient calculation method based on the number of constraints
4.4.2 Performance

The performance of the semi-analytic methods was characterized by running several test cases on the Analytic and Scalable problems. Initially, the Analytic problem was optimized with the MDF architecture in both its original form, and its semi-analytic form using the adjoint method for the objective sensitivities and the direct method for the constraints. To avoid biasing the results, the residual free versions of both methods were used. These methods were also applied to a version of the Scalable problem with three disciplines, three global design variables, 10 local design variables per discipline, and 10 coupling variables per discipline. The results in terms of function calls are shown in table ???. The advantage of the semi-analytic methods increases with the dimensionality of the problem. For the Analytic example, with three design variables and two coupling variables, the semi-analytic methods reduce the required function calls by ≈ 37%. The Scalable example, with 33 design variables and 30 coupling variables, sees a reduction in function calls of ≈ 94%. This can be attributed to the fact that the number of sensitivity calculations for the objective
function gradients scales with $n \times n_{MDA}$ and the number of constraint gradient calculations scales with $n \times n_{con} \times n_{MDA}$. By reducing the constraint and objective gradient calculations to only one MDA per iteration, the semi-analytic methods significantly reduce their computational cost.

A comparison was also made to the sensitivity calculation times using complex step differencing. This is shown in Figure 4.11 for one instance of the scalable problem, with three disciplines and twenty local design variables and twenty coupling variables per discipline. Whether the partial derivatives are gathered with the finite difference or complex step method, the improvement is dramatic. The majority of the improvement is due to a reduction in the number of multidisciplinary analyses (MDAs) that must be converged for each iteration. With finite difference or complex step sensitivities, one MDA is performed for each design variable at every iteration. With the semi-analytic methods, despite the higher computational overhead of the matrix multiplication, only one MDA is required per iteration. This results in vast reduction in computational time and function calls.

The convergence of the Scalable problem was plotted for each of the sensitivity methods. As shown in Figure 4.12, the path followed by the optimizer is identical
Section 4.4. Semi-analytic Sensitivity Analysis

Figure 4.11: Comparison of sensitivity calculation times using the original complex step differencing method and the new semi-analytic methods. Calculation times are normalized to the multi-disciplinary analysis performed by the MDF architecture.

in each case, but the semi-analytic methods significantly reduce the expense of each iteration, both in terms of function calls and computational time. As indicated by the function call results, the use of semi-analytic sensitivities greatly reduces the number of iterations for the problem. The complex step method reduces the number of function calls and the convergence time for both the semi-analytic and original sensitivities, but this effect is more visible for the original formulation due to the scale of the graph.

The performance of the semi-analytic methods was also compared to the single discipline architectures for increasing complexity of the scalable problem. Results are shown in Figure 4.13 for an increasing dimensionality of the local design variables with a constant number of coupling variables. In each case, the multidisciplinary problem was composed of five disciplines with five global design variables. The fixed number of variables in each test case was ten variables for each discipline, either local or coupling.

The incorporation of semi-analytic sensitivity calculations clearly improves the performance of the MDF architecture in this case. Not only do these methods sig-
Figure 4.12: Objective function convergence for Scalable problem with semi-analytic sensitivities. The adjoint version of MDF travels a similar path to the original, but converges an order of magnitude faster.

significantly improve the convergence speed of MDF, they actually outperform IDF and SAND in the majority of cases. Prior to these tests, IDF and SAND were the fastest, best performing architectures for the scalable problem yet found. Figure 4.14 shows the results of the same trial, except the coupling between disciplines was randomly selected during problem definition. At each point, every discipline was coupled to at least two other disciplines, but the overall number of coupling variables for each discipline was allowed to vary between two and four (full coupling). This introduces more noise into the resulting plots, but allows the performance of each architecture to be measured for problems with variable coupling dimensionality.

Finally, results are shown in Figure 4.15 for an increasing number of coupling variables per discipline with full coupling.

Though the semi-analytic MDF formulation no longer approaches the performance of IDF, it is still significantly improved from the original implementation. Incorpor-
Figure 4.13: Performance of the semi analytic sensitivity methods for increasing dimensionality of local design variables.

rating the coupled adjoint or direct methods clearly has a significant effect upon total convergence time of the MDF architecture.

In terms of time performance, the semi-analytic sensitivity methods are “embarrassingly parallel” in that the gathering of the required partial derivatives lends itself to parallel computing. Due to the object-oriented nature of πMDO, implementation of parallel finite differencing or complex step would be simply achieved by overwriting the calcpartials and calc_partialresid methods at the MdoProb level.

### 4.4.3 Other architectures

Because all of the necessary methods for the semi-analytic sensitivity methods are at the MdoProb level in πMDO, they are available to all of the architectures. Depending on the structure of the architecture, however, these methods may not provide a performance gain large enough to justify their increased complexity. In particular, examination of the IDF and SAND formulations, shows that their sensitivities are already calculated in a decoupled, and highly efficient way. The implementation of
Figure 4.14: Performance of the semi analytic sensitivity methods for increasing dimensionality of local design variables. Coupling between disciplines was randomly selected.

coupled sensitivity analysis for the system level derivatives would therefore increase the computational overhead of the architecture with no gain in performance.

One way a small performance gain could be achieved on these monolithic architectures is by reducing the number of unnecessary derivative calculations by exploiting the information on discipline inputs contained in the framework. Using the calc\_partial or calc\_partialresid methods, derivatives could be gathered from a discipline only if that design variable appeared in the discipline.inputs list. If the design variable did not appear, a simple 0 derivative could be substituted. This could reduce the number of function calls slightly and speed the optimization. This benefit would be only due to more efficient finite differencing, rather than the implementation of coupled sensitivity analysis, but would exploit some routines developed for the coupled methods.
4.5 Reconfigurability

A version of the scalable problem was tailored to explore the advantages of the nested optimization architecture. This version has five disciplines, two of them tightly coupled with a large number of state variables. A smaller number of state variables from these two disciplines are used in the objective function, and as coupling variables with the remaining disciplines. The structure of this test problem before and after reconfiguration is shown in Figure 4.16 below.

The function calls required for each of the MDF, CO, and NEST architectures to converge to the optimum are given below in Figure 4.5.

The function calls show the new architecture allocating computational effort properly to the most highly coupled disciplines. In this initial implementation, the modified MDF architecture used for the tightly coupled disciplines does not make use of the semi-analytic sensitivity methods. Once the semi-analytic methods are implemented on the sub-optimization, the performance of the NEST architecture should
Section 4.5. Reconfigurability

Figure 4.16: Problem reconfiguration for the NEST architecture. The MDF sub-optimization significantly reduces the coupling variables seen by the system level CO architecture.

improve dramatically. Further, the effect of reducing the coupling dimensionality on the system level CO problem is clearly shown - the NEST architecture has fewer function calls for disciplines three, four, and five than the CO architecture. When the number of coupling variables at the system level is increased, the performance of CO and MDF will deteriorate at a faster rate than that of the nested optimization. The relative error in the objective function for all three architectures is plotted in Figure below.

The NEST architecture clearly improves upon the MDF formulation in both function evaluations and convergence time overall. Though the CO architecture performs better for this example, the function evaluation comparison shows that the NEST architecture reduced the computational time being spent on the loosely connected disciplines. Should the semi-analytic sensitivity methods be applied to the nested
Section 4.5. Reconfigurability

![Table 4.6: Function call comparison for semi-analytic sensitivity methods. Note that this comparison is of the total analysis calls per discipline, including finite differencing.](image)

MDF optimization, the results would improve dramatically, and potentially be competitive with CO. Further, these results clearly show the potential of the concept of nesting architectures within πMDO. The new hybrid architecture, developed and implemented with very little effort through inheritance and code reuse, successfully combined the CO and MDF architecture. The resulting method, NEST, displayed performance that was a blend between the characteristics of its two constituent components. For certain classes of problems, especially with semi analytic sensitivities, this hybrid architecture could be competitive and reduce the amount of time and computational resources required for optimization.
Figure 4.17: Convergence plot for the NEST architecture, a hybrid of MDF and CO. Note that the nested formulation has performance between its two constituent architectures.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

The functionality and utility of the πMDO framework was significantly extended, and greater insight was gained into the dynamics of multidisciplinary design optimization. In particular, the coupled adjoint and direct semi-analytic sensitivity methods were incorporated into the πMDO framework, dramatically improving the performance of the MDF architecture.

Further, significant improvements to the CO architecture were made by incorporating constraint relaxation, and the modified architecture was extensively tested. The new architecture was found to be faster and more robust than the original implementation, and also provide flexibility in modifying the required convergence tolerance between disciplines.

πMDO was also greatly enhanced by incorporating and interfacing with object oriented frameworks for optimization algorithms, pyOpt, and meta model approximations, pyMMT. This open framework approach allows optimizers and meta models to be easily exchanged and the best combination to be chosen for a particular situation.

Finally, taking the object oriented philosophy to its logical conclusion, the MDO architectures within πMDO were themselves reconfigured and combined to produce new hybrid architectures. One hybrid design combined the functionality of the CO
and MDF architectures to more effectively handle problems with highly localized coupling between disciplines.

The ability to simply and efficiently choose and combine MDO architectures, optimizers, and meta models demonstrates the utility of the open framework approach. In addition to providing simple and quick implementation of MDO architectures, πMDO can now serve as the basis for future research in meta models, optimization algorithms, and MDO approaches.

5.2 Future Work

Though significant improvements were made to the πMDO framework, more work remains.

Overall, though the software itself is constantly improving, there is room for many efficiency gains through effective programming and simplification of code. In particular, a thorough review of bottlenecks in the framework and subsequent mitigation efforts could dramatically reduce the computational overhead πMDO introduces.

In line with reducing the overhead of πMDO, the current version implements all of its internal math and memory access in complex variable format. This allows use of the complex step method throughout, but handicaps the speed of the framework if only finite differencing is desired. Implementing two parallel branches of πMDO, one complex and one real, would allow the extra overhead to be eliminated for real calculations. Further, time consuming sections of code could be written in a faster language than python, e.g. Fortran or C, and wrapped as python libraries.

While parallelization is implemented in a limited way for the pyMMT meta model classes, this should be extended to the πMDO framework as a whole. Architectures such as IDF, CO, CSSO, and BLISS which are “embarrassingly parallel” should be implemented and tested with parallel processing, ideally through the existing MPI for Python toolkit [43]. While this is being performed, an effort should be made to coordinate and standardize the interface and backend of πMDO, pyACDT, pyMMT, and the new MPI modules. Some duplication still remains between the πMDO and pyACDT standards for the description of variables, disciplines, and optimization
problems.

Further, though many architectures are implemented in πMDO, there remain many architectures for which no comparisons exist. In particular, now that the meta model options within the framework are more mature, BLISS 2000 [20] should be implemented. As an intermediate stage while the maturity of the pyMMT module improves, it could be implemented with the disciplines themselves serving as an “approximation” of the discipline responses with zero error. Though this may be slower than a mature approximation model, it would remove much of the complexity of the implementation and allow the architecture itself to be tested without waiting for more efficient meta models. Other decomposition algorithms, such as analytic target cascading [44, 45] and the augmented lagrangian method proposed by Tosserams et. al. [46] should also be implemented. Last, the modification of the CO architecture known as enhanced collaborative optimization [47] should be implemented.

As one of the major goals of πMDO is to provide a suite of common test problems, these problems must be sought out and implemented in the proper format. One excellent opportunity created through integration with the pyACDT framework is the availability of many common aircraft analysis modules. These modules should be used in various configurations to produce innovative aircraft design problems exploring unconventional aerodynamics, engine emissions, etc. In particular, a variable fidelity problem with tightly coupled aerostructural interactions and some loosely coupled disciplines like controls or weights should be produced to test collaborative optimization and the NEST architecture.

Finally, to ensure the propagation and survival of πMDO as a software project, efforts should be made to complete the transition to a fully open source and user supported distribution model. To this end, the user interface should be further improved to encourage new practitioners to adopt πMDO. The existing visualization and performance measurement functions within πMDO should be coherently integrated into the framework, rather than simply included as stand-alone scripts. Most importantly, the community of users of πMDO must be encouraged to assist in the debugging and development of the software framework, and the test problems implemented within it.
References


References

of the 34th AIAA Aerospace Sciences Meeting and Exhibit, Reno, NV, January 1996.


