ABSTRACT

REED, JAMES ALLEN. A Low Rank Approach to Computing Derivatives Using Automatic Differentiation. (Under the direction of Hany S. Abdel-Khalik).

This manuscript outlines a new approach for increasing the efficiency of applying automatic differentiation (AD) to large scale computational models. By using the principles of the Efficient Subspace Method (ESM), low rank approximations of the derivatives for first and higher orders can be calculated using minimized computational resources. The output obtained from nuclear reactor calculations typically has a much smaller numerical rank compared to the number of inputs and outputs. This rank deficiency can be exploited to reduce the number of derivatives that need to be calculated using AD. The effective rank can be determined according to ESM by computing derivatives with AD at random inputs. Reduced or pseudo variables are then defined and new derivatives are calculated with respect to the pseudo variables. Two different AD packages are used: OpenAD and Rapsodia. OpenAD is used to determine the effective rank and the subspace that contains the derivatives. Rapsodia is then used to calculate derivatives with respect to the pseudo variables for the desired order. The overall approach is applied to a few simple mathematical model problems and to MATWS, a simplified safety code for sodium cooled reactors.

A Low Rank Approach to Computing Derivatives Using Automatic Differentiation

by
James Allen Reed

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Nuclear Engineering

Raleigh, North Carolina

2012

APPROVED BY:

_____	_____
Steven Campbell	Paul Hovland

_____
Hany S. Abdel-Khalik

Committee Chair

BIOGRAPHY

James A. Reed was born on October 18th 1987 in Beaver, Pennsylvania. He attended Penn State University in State College, PA from 2006 to 2010 where he obtained Bachelor of Science degrees in Nuclear and Mechanical Engineering with honors in Nuclear Engineering. Upon graduation from Penn State, he continued his studies in Nuclear Engineering at North Carolina State University in Raleigh. While at NCSU, he started working on using automatic differentiation as a means to calculate derivatives from nuclear reactor codes for use in sensitivity analysis and uncertainty quantification. He collaborated with the Mathematics and Computer Science Division at Argonne National Laboratory and the work performed is presented here in this thesis.

ACKNOWLEDGMENTS

I would first like to thank my parents for their support. Without them, I would not have any chance to be in the position that I am today. I would also like to thank Dr. Abdel-Khalik for his guidance and for allowing me the opportunity to work under him at North Carolina State University. I must also thank Paul Hovland for sponsoring me during the summer at Argonne National Laboratory where essential work for this project was completed. I very grateful for the assistance I received from Jean Utke while at Argonne. Without his work on OpenAD and Rapsodia, none of this would be possible. Last but certainly not least, I am dearly thankful for the love and support from Casandra Niebel, who kept me sane during the entire process.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 Introduction

1.1 Applications of Automatic Differentiation

Because detailed nuclear reactor calculations involve large input and output streams, calculating derivative information can be a very time consuming task. Derivative information is required for many aspects of nuclear engineering and analysis, such as sensitivity analysis, design optimization, code-based uncertainty propagation, and data assimilation. One tool that can be used to facilitate the calculation of derivatives is automatic differentiation (AD) software. AD is a technique to reinterpret or completely transform a computer program implementing a numerical model with the goal to calculate the derivatives of specified output variables of the model with respect to specified input variables. The principal method of AD is the application of the chain rule to the given elemental decomposition of a mathematical function [1]. With continuing advances in computer power, the application of AD is becoming a more feasible and attractive option for the calculation of accurate derivatives [2] [3].

1.2 Uses of Derivatives

Derivative information is essential to sensitivity analysis, uncertainty quantification, design optimization, data assimilation, and surrogate modeling. Sensitivity analysis involves relating the changes in the outputs of a model to changes in the inputs. This is usually accomplished by using the Jacobian operator, a matrix of first order derivatives. Uncertainty

quantification can utilize model parameter sensitivities to estimate the potential error in outputs in order to validate model. Derivatives are used in design optimization problems in order to effectively tweak design parameters to obtain optimal performance. Data assimilation uses derivatives for statistical interpolation of given data in order to estimate the state of a given system. Surrogate modeling requires derivatives in order to build an approximate model of a system via a truncated Taylor expansion.

The means of obtaining the full derivatives that populate the Jacobian matrix for a sufficient number of model operating points is a very time and resource consuming task. AD can be used to obtain the derivative values to within machine precision. Other possible methods that are used for obtaining derivatives from computer codes of engineering models involve hand-coding derivatives and using finite differences, but some of the more advanced methods are the Generalized Perturbation Method and the Adjoint Method [4]. These methods generally allow the computer code to be run faster than the corresponding AD versions of the code, but the accuracy is generally lower and there is more room for error. Therefore, it is desirable to enhance the efficiency of using AD and maintain acceptable accuracy.

Two software package options for AD are OpenAD (www.mcs.anl.gov/OpenAD) and Rapsodia (www.mcs.anl.gov/Rapsodia). In OpenAD, the derivative evaluation is performed by a program resulting from the analysis and transformation of the original program that implements the mathematical function or model of interest [4]. While OpenAD relies on source transformation to accomplish derivative calculations, the Rapsodia tool uses operator

overloading as the vehicle of attaching derivative computations to the elementary operations provided by the programming language such as the arithmetic operators and intrinsic functions sin(x), exp(x), and so forth [5]. In our context, we use OpenAD with the so-called reverse mode providing for the efficient computation of gradients with respect to a large number of inputs. In contrast, Rapsodia implements higher order derivative computation in forward mode. It is normally efficient for a small number of input variables and the overloading overhead becomes negligible with higher derivative order. Both approaches are, in principle, capable of calculating higher order derivatives. Higher-order derivatives with source transformation by repeated application of the transformation tool increases complexity of the tool and the program size and has not been shown to yield large benefits when compared to operator overloading keeping in mind the expected small number of input variables.

In nuclear engineering, derivatives are mainly used for the purposes of sensitivity analysis and uncertainty quantification. Computational methods and uncertainties in input data are the main limitation of the calculations necessary to design nuclear reactor systems. Sensitivity analysis is often used to analyze the nuclear fuel cycle and the behavior of the fuel. In analyzing a nuclear fuel rod, the sensitivities of key variables (fuel centerline temperature, fission gas release, clad stress, etc.) to input parameters are found to be highly non-intuitive and strongly dependent on the fuel-clad gap status and the history of the fuel during the cycle [6]. The number of variables analyzed can be quite large, especially when highly dimensioned reactor wide calculations are performed. For example, the outputs of

interest could be the neutron flux, fuel temperature, moderator temperature, and void fraction at thousands of points throughout the reactor. The inputs of interest could be the complete set of neutron cross sections that quantify the probabilities of different reactions taking place in each type of material. It is easy to see that as the dimensions and details of the calculations increase, the number of derivatives increases as well. The increase in the number of higher order derivatives grows exponentially. For typical nuclear reactor calculations, the number of inputs *n* and outputs *m* are on the orders of 10^6 and 10^5, respectively. The numerical rank *r* of these problems is often orders of magnitude smaller than the size of the input and output data streams with *r* typically around 10^2 [7]. This fact can be used to reduce the effective dimensions of the problem and lessen the computational time and storage requirements.

1.3 Problem Reduction via the Efficient Subspace Method

The mathematical theory of efficient subspace methods (ESM) recognizes that the design and/or analysis of an engineering system is often judged by a few macroscopic metrics that capture the overall behavior of the system [8]. ESM exploits the ill-conditioning of the Jacobian matrix to reduce the number of code runs of the forward and reverse modes of AD to a minimum [7]. ESM utilizes various properties from linear algebra including orthogonality and the singular value decomposition (SVD) in order to identify the minimum information necessary to represent the overall response of the system. It can be shown by doing a rigorous sensitivity analysis of a system that variations in some inputs do not

contribute much to the overall behavior of the system relative to other inputs. These variables are deemed to be not as important, and by identifying them through ESM, their place in the overall analysis can be lessened or ignored in order to focus on the more important quantities and still capture the overall behavior of the system.

An example that illustrates the ideas behind ESM is in calculus for an integral quantity such as distance via velocity profiles. The same distance can be obtained by integrating different velocity profiles. Therefore, the question becomes "how can one identify the required modeling changes associated with the different physics that will lead to more accurate estimation of the macroscopic metrics of interest?" rather than "how can one enhance the accuracy of the different field solutions [8]?"

1.4 Multi-scale Phenomena Modeling

The large rank reduction that can be obtained in some reactor calculations is a result of the multi-scale phenomena modeling (MSP) strategy on which nuclear reactor calculations are based. Besides nuclear reactor calculations, many other engineering systems involve large variations in both time and length scales and are examples of applications of MSP. In fact, many of today's important engineering and physical phenomena are modeled via MSP, e.g. weather forecast, geophysics, materials simulation [7]. To accurately model the large time and scale variations, MSP utilizes a series of models varying in complexity and dimensionality [7]. First, high resolution (HR) microscopic models are employed to capture the basic physics and the short scales that govern system behavior. The HR models are then

coupled with low resolution (LR) macroscopic models to directly calculate the macroscopic system behavior, which is often of interest to system designers, operators, and experimentalists. The coupling between the different models results in a gradual reduction in problem dimensionality thus creating large degrees of correlations between different data in the input and output (I/O) data streams. ESM exploits this situation by treating the I/O data in a collective manner in search of the independent pieces of information. The term 'Degree of Freedom' (DOF), adopted in many other engineering fields, is used to denote an independent piece of information in the I/O stream. An active DOF denotes a DOF that is transferred from a higher to a lower resolution model, and an inactive DOF denotes a DOF that is thrown out. ESM replaces the original I/O data streams by their corresponding active DOFs. The number of active DOFs can be related to the numerical rank of the Jacobian matrix.

1.5 Thesis Contents

This manuscript presents a method for using OpenAD and Rapsodia to efficiently calculate first and higher order derivatives by reducing the size of the input stream according to the efficient subspace method (ESM). Chapter 2 of this work describes the computational methods and theory behind AD and the application of ESM. Chapter 3 presents a generalized description of how this approach can be applied to a computational model. Chapter 4 presents some simple examples of using the method as well as the results of applying this approach to MATWS, a safety code for sodium cooled fast reactors that

combines the SAS4A/SASSYS computer code with a simplified representation of the reactor heat removal system [9].

CHAPTER 2 Computational Methods

2.1 AD Theory

The quick, easy, intuitive (but inaccurate) way to calculate derivatives is by using a finite difference or divided difference approach. The first order derivative for a function $y = f(x)$ is given by:

$$\frac{dy}{dx} = f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{1}$$

This method requires very small values of *h* in order to obtain useful results. Automatic (or algorithmic) differentiation instead relies on the exact analytical expressions of the derivatives for the basic functions ($\sin x$, $\exp x$, $etc.$) that make up the overall function in question. The chain rule of differentiation is used in order to follow the differentiation calculation through the basic functions of the program. Equation (2) gives the chain rule for a function *f* that is a function of another function *g* which is a function of *x*. Thus, *f* is ultimately a function of *x* itself.

$$\frac{df}{dx} = f'\big(g(x)\big) = f'\big(g(x)\big)\, g'(x) \tag{2}$$

Equation (3) gives the chain rule in the case when $y = f(u)$ and $u = g(x)$.

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx} \tag{3}$$

To show how the chain rule is used in AD, consider the following the example. Consider the following function where $y = f(x_1, x_2)$:

$$y = \left[\frac{\cosh\left(\frac{x_1}{x_2}\right)}{\sin\left(\frac{x_1}{x_2}\right)\exp\left(\frac{x_1}{x_2}\right)}\right]\frac{x_1}{x_2} \tag{4}$$

A computer will calculate *y* from x_1 and x_2 in much the same way that a human being would go about doing it by hand. Intermediate values will be calculated for the lowest level sub-functions (such as x_1/x_2 in this case), and then the values for these sub-functions will be used to calculate the values for higher level functions ($\cosh(x_1/x_2)$, $\sin(x_1/x_2)$, $\exp(x_1/x_2)$). This process is continued until the overall value for the function is computed. Most AD software packages follow this same control flow that is referred to as an evaluation trace. An evaluation trace is basically a record of a particular run of a given program, with particular specified values for the input variables, showing the sequence of floating point values calculated by a (slightly idealized) processor and the operations that computed them [1]. Table 2-1 shows an evaluation trace for Equation (4).

The v_i intermediate mathematic variables are different from normal program variables as they can normally not be assigned a value more than once [1]. A real computer program that models an actual engineering system will contain functions that are much more complex and have many more intermediates than the one given by Equation (4). To effectively follow variables through a program, a "computational graph" is often used to give a visual representation of an evaluation trace [1]. Figure 2.1 shows the computational graph for the evaluation trace in Table 2-1.

Table 2-1: Evaluation Trace for Equation (4)

v_{-1}	=	x_1	=	2.000		
v_0	=	x_2	=	1.000		
v_1	=	v_{-1}/v_0	=	2.000/1.000	=	2.000
v_2	=	$\cosh v_1$	=	$\cosh 2.000$	=	3.762
v_3	=	$\sin v_1$	=	$\sin 2.000$	=	0.9093
v_4	=	$\exp v_1$	=	$\exp 2.000$	=	7.389
v_5	=	$\frac{v_2}{v_3 v_4}$	=	$\frac{3.762}{0.9093(7.389)}$	=	0.5600
v_6	=	$(v_5 + v_1)v_1$	=	$(0.5600 + 2.000)2.000$	=	5.120
y	=	v_6	=	5.120		

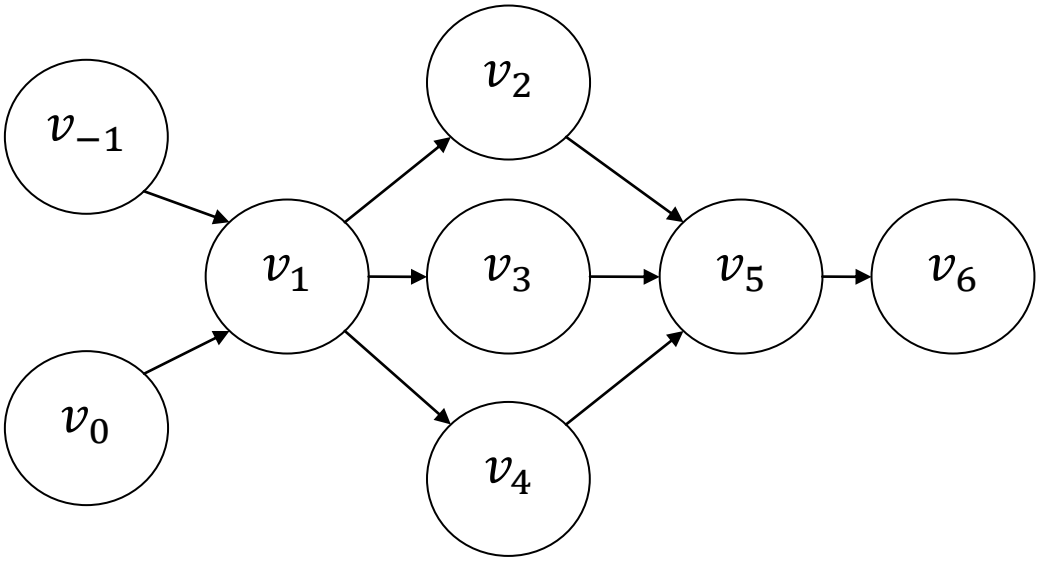

Figure 2.1: Computation Graph of Table 2-1

2.1.1 Forward Mode

Many AD packages feature two ways to follow the evaluation trace of the program. The first and most basic way that an AD package can operate is by calculating the derivatives of the inputs and following the evaluation trace to the outputs, calculating the derivatives of each intermediate along the way. In the example presented earlier, in order to calculate derivatives of *y* with respect to x_1, each intermediate variable must be differentiated with respect to x_1 and evaluated for the desired values of x_1 and x_2. This will create new intermediate derivative variables ($\dot{v}_i = \partial v_i/\partial x_1$) that will be associated with each corresponding intermediate variable. Starting with the first line in Table 2-1 it is easy to see that $v_{-1} = 1.000$ and $v_0 = 0.000$. Moving to v_1 gives the following:

$$\dot{v}_1 = (\partial v_i/\partial v_{-1})\dot{v}_{-1} + (\partial v_i/\partial v_0)\dot{v}_0 = \frac{(\dot{v}_{-1}v_0 - v_1\dot{v}_0)}{v_0^2}$$

$$= \frac{\big(1.000(1.000) - 0.0000(2.000)\big)}{1.000^2} = 1.000$$

Table 2-2 gives the full evaluation trace for the forward calculation of derivatives. If derivatives with respect to x_2 are desired, the process can be repeated but instead $\dot{x}_1 = 0.0000$ and $\dot{x}_2 = 1.0000$. For problems with multiple outputs, the forward mode can be used to obtain the derivatives of each desired output variable with respect to a single input variable in one run. This makes the forward mode more desirable to use when the number of outputs is larger than the number of inputs.

Table 2-2: Forward mode evaluation trace

v_{-1}	=	x_1	=	2.000		
$\dot{v}_{-1}$	=	$\dot{x}_1$	=	1.000		
v_0	=	x_2	=	1.000		
$\dot{v}_0$	=	$\dot{x}_1$	=	0.000		
v_1	=	v_{-1}/v_0	=	2.000/1.000	=	2.000
$\dot{v}_1$	=	$(\dot{v}_{-1}v_0 - \dot{v}_0 v_{-1})/v_0^2$	=	$\big(1.00(1.00) - 0.00(2.00)\big)/1.00^2$	=	1.000
v_2	=	$\cosh v_1$	=	$\cosh(2.000)$	=	3.762
$\dot{v}_2$	=	$\dot{v}_1 \sinh v_1$	=	$1.00(\sinh(2.00))$	=	3.627
v_3	=	$\sin v_1$	=	$\sin(2.000)$	=	0.9093
$\dot{v}_3$	=	$\dot{v}_1 \cos v_1$	=	$1.00(\cos(2.00))$	=	-0.4161
v_4	=	$\exp v_1$	=	$\exp(2.000)$	=	7.389
$\dot{v}_4$	=	$\dot{v}_1 \exp v_1$	=	$1.00(\exp(2.00))$	=	7.389
v_5	=	$\frac{v_2}{v_3 v_4}$	=	$\frac{3.762}{0.9093(7.389)}$	=	0.5600
$\dot{v}_5$	=	$\frac{(\dot{v}_2 v_3 v_4 - (\dot{v}_3 v_4 + \dot{v}_4 v_3)v_2}{(v_3 v_4)^2}$	=	$\frac{\big(3.62(0.90)(7.38) - \big(-0.41(7.38) + 7.38(0.90)\big)3.76\big)}{\big(0.90(7.38)\big)^2}$	=	0.2361
v_6	=	$(v_5 + v_1)v_1$	=	$(0.5600 + 2.000)2.000$	=	5.120
$\dot{v}_6$	=	$\dot{v}_1 v_5 + \dot{v}_5 v_1 + 2v_1\dot{v}_1$	=	$1.00(0.55) + 0.23(2.00) + 2(2.00)(1.00)$	=	5.032
y	=	v_6	=	5.200		
$\dot{y}$	=	$\dot{v}_6$	=	5.032		

2.1.2 Reverse Mode

In addition to the forward mode, some AD packages (including OpenAD) feature a reverse or adjoint mode. Instead of selecting an independent variable and calculating the derivatives of every intermediate variable with respect to that variable, a dependent variable is chosen and the derivatives of that variable with respect to each intermediate variable are calculated [1]. In order to perform an evaluation trace in the reverse mode, new adjoint variables will be defined. Let $\bar{v}_i = \frac{\partial y}{\partial v_i}$ (in a strict sense, $\bar{v}_i$ actually is defined to be $\frac{\partial y}{\partial \delta_i}$ where δ_i is a new independent variable added to the right-hand side of the equation defining v_i [1]). The evaluation trace starts from the final steps of the normal evaluation trace and works backwards, hence reverse mode. The desired dependent adjoint variable will be set to 1.000, which for this case of a single output means that $\bar{y} = 1.000$. Table 2-3 gives the normal evaluation trace followed by the evaluation trace for a reverse mode derivative calculation. Note that each line in the reverse mode calculation is lined up with the corresponding line of the model calculation above. The application of the chain rule in reverse mode can be confusing, so as an example, consider tracing backwards through the model calculation to the line $v_6 = (v_5 + v_1)v_1$. Here, v_6 depends on v_5 and v_1. The adjoint variables corresponding to this line are $\bar{v}_5 = \frac{dy}{dv_5} = \frac{dy}{dv_6}\frac{dv_6}{dv_5}$ and $\bar{v}_1 = \frac{dy}{dv_1} = \frac{dy}{dv_6}\frac{dv_6}{dv_1}$. Noting that $\frac{dy}{dv_6} = \bar{v}_6$ and evaluating the current expression for v_6 to get the other required derivatives gives $\bar{v}_5 = \bar{v}_6 v_1$ and $\bar{v}_1 = \bar{v}_6(v_5 + 2v_1)$. The next step is to follow the normal evaluation trace backwards to

the next step, $v_5 = \frac{v_2}{v_3 v_4}$, and repeat. For variables that appear multiple times in the trace (such as v_1), the previously calculated adjoint values are accumulated into the new calculation (as is shown in the lines $\bar{v}_1 = \bar{v}_1 + \bar{v}_4 \exp v_1$ and $\bar{v}_1 = \bar{v}_1 + \bar{v}_3 \cos v_1$). The value $\frac{\partial y}{\partial x_1} = \bar{x}_1 = 5.032$ obtained in the reverse mode trace agrees with what was obtained in the forward trace shown in Table 2-2. Also note that $\frac{\partial y}{\partial x_2}$ was calculated in the reverse mode trace with only a single extra calculation. This makes the reverse mode useful for models where the number of inputs is greater than the number of outputs. However, the reverse mode transformation can be difficult to implement due to the fact that the evaluation trace must be reversible which can be an issue for some model codes.

2.1.3 Active Variables

The concept of an 'active variable' is important to AD. An 'active variable' is any variable in the model that comes into contact via assignment to the dependent/independent variables. For example, in a program that computes z from x in the following fashion:

$$y = ax \qquad z = y/b$$

and $\frac{dz}{dx}$ is the value desired by AD calculation, the active variables are $x, y,$ and z. The parameters a and b are viewed as 'passive variables' as they are not assigned values that depend on the independent variable x. It will be shown in the following sections that active

variables in OpenAD and Rapsodia require type changes from real, double precision, etc. to a custom defined active type in order to operate.

2.2 OpenAD

OpenAD uses association by address [5], that is an active type, as the means of augmenting the original program data to hold the derivative information. The usual activity analysis would ordinarily trigger the re-declaration of only a subset of common block variables. Because the access of the common block via the array enforces a uniform type for all common block variables to maintain proper alignment, all common block variables had to be activated. Furthermore, because the equivalence construct applied syntactically only to the first common block variable, the implicit equivalence of all other variables cannot be automatically deduced and required a change of the analysis logic for OpenAD to maintain alignment by conservatively overestimating the active variable set. Superficially this may seem a drawback of the association by address. The association by name [10], used in other AD source transformation tools will not fare better though. Shortening the corresponding loop for the name-associated and equivalenced derivative-carrying array is difficult for interspersed passive data and therefore one will resort to the same alignment requirement.

2.3 Rapsodia

Rapsodia is based on operator overloading for the forward propagation of univariate Taylor polynomials. All other operator overloading based AD tools have overloaded

Table 2-3: Reverse mode evaluation trace

$$v_{-1} = x_1 = 2.000$$
$$v_0 = x_2 = 1.000$$
$$v_1 = v_{-1}/v_0 = 1.000/2.000 = 2.000$$
$$v_2 = \cosh v_1 = \cosh 2.000 = 3.762$$
$$v_3 = \sin v_1 = \sin 2.000 = 0.9093$$
$$v_4 = \exp v_1 = \exp 2.000 = 7.389$$
$$v_5 = \frac{v_2}{v_3 v_4} = \frac{3.762}{0.9093(7.389)} = 0.5600$$
$$v_6 = (v_5 + v_1)v_1 = 5.120$$
$$y = v_6 = 5.120$$
$$\bar{v}_6 = \bar{y} = 1.000$$
$$\bar{v}_5 = \bar{v}_6 v_1 = 1.000(2.000) = 2.000$$
$$\bar{v}_1 = \bar{v}_6(v_5 + 2v_1) = 4.560$$
$$\bar{v}_4 = -\bar{v}_5 \frac{v_2}{v_3}\, v_4^{-2} = -0.1516$$
$$v_3 = -\bar{v}_5 \frac{v_2}{v_4}\, v_3^{-2} = -1.232$$
$$\bar{v}_2 = \bar{v}_5 \frac{1}{v_3 v_4} = 0.2977$$
$$\bar{v}_1 = \bar{v}_1 + \bar{v}_4 \exp v_1 = 3.440$$
$$\bar{v}_1 = \bar{v}_1 + \bar{v}_3 \cos v_1 = 3.953$$
$$\bar{v}_1 = \bar{v}_1 + \bar{v}_2 \sinh v_1 = 5.032$$
$$\bar{v}_{-1} = \bar{v}_1 \frac{1}{v_0} = 5.032$$
$$\bar{v}_0 = -\bar{v}_1 v_{-1} v_0^{-2} = -10.06$$
$$\bar{x}_2 = \bar{v}_0 = -10.06$$
$$\bar{x}_1 = \bar{v}_{-1} = 5.032$$

operators that are hand-coded, operate on Taylor coefficient arrays with variable length in loops with variable bounds to accommodate the derivative orders and numbers of directions needed by the application. In contrast, Rapsodia generates on demand a library of overloaded operators for a specific number of directions and a specific order. Thus, at compile time, the loops are already represented in (partially) unrolled code along with a fixed (partially flat) data structure that provides more freedom for compiler optimization. Because of the overall assumption that r, the reduced input dimension, is much smaller than m the higher order derivative computation in forward mode is feasible and appropriate.

Because overloaded operators are triggered by using a special (active) type for which they are declared it now appears as a nice confluence of features that OpenAD for the gradient computation already does the data augmentation via association by address, i.e. via an active type, and therefore the assumption could be made that one merely has to change the OpenAD active type to a Rapsodia active type to use the operator overloading library.

2.4 Efficient Subspace Method

Subspace methods are based on mathematical ideas in linear algebra. The key components are the vector spaces that exist in matrix representations of the inputs and outputs of a model in question. The goal of using subspace methods in relation to the method presented here is to determine a low rank approximation of the model using information gathered from the first order sensitivity (Jacobian) matrix. The effective rank that is desired corresponds to the active degrees of freedom (DOF) of the model. A DOF denotes an

independent piece of information in the input/output stream [8]. An active DOF denotes a DOF that is transferred from a higher to a lower resolution model, and an inactive DOF denotes one that is thrown out [8].

A very important tool used in subspace methods is the matrix decomposition. Examples of common matrix decompositions are the QR factorization and singular value decomposition (SVD). For an $m \times n$ Jacobian matrix $\mathbf{J}$ with rank $r < \min(m, n)$, the SVD is given by:

$$\mathbf{J} = \mathbf{U}\mathbf{S}\mathbf{V}^T \tag{5}$$

where $\mathbf{U} \in \mathbb{R}^{m \times r}$ and $\mathbf{V} \in \mathbb{R}^{n \times r}$ are full column rank orthonormal matrices that constitute orthonormal bases for the vector spaces $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively. $\mathbf{S} \in \mathbb{R}^{r \times r}$ is a nonsingular diagonal matrix whose elements correspond to the singular values (usually organized from largest to smallest) of $\mathbf{J}$.

The SVD is a so called 'rank revealing' decomposition because the number of non-zero singular values correspond to the numerical rank of the original matrix. In practice, all singular values will be non-zero, but the SVD still allows for the 'effective' rank to be determined. Only the largest singular values will count towards the effective rank. The cutoff criterion that determines which singular values count towards the effective rank can vary depending on the desired accuracy.

By determining an effective rank of a matrix which is lower than $\min(m,n)$, a low rank approximation can be constructed. After determining the effective rank by inspection of the SVD or by using the rank finding algorithm that will be introduced in the next section, a number of vectors corresponding to the size of the active subspace can be used in constructing a low rank approximation.

2.5 General Methodology

To start, a simple example of constructing a low rank approximation to a matrix operator will be considered. Let the matrix in question in be $\mathbf{A} \in \mathbb{R}^{m\times n}$. The elements of $\mathbf{A}$ are not known, but the ability to perform matrix vector products with $\mathbf{A}$ and $\mathbf{A}^T$ is available. The steps involved in determining a low rank approximation of $\mathbf{A}$ are as follows:

1. Use k random Gaussian input vectors $x^{(i)}$ to compute k matrix vector products: $y^{(i)} = \mathbf{A}x^{(i)}$
2. Perform a QR decomposition on the responses: $\left[y^{(1)} \ldots y^{(k)}\right] = \mathbf{QR} = \left[q^{(1)} \ldots q^{(k)}\right]\mathbf{R}$
3. Determine the effective rank r using the Rank Finding Algorithm (RFA):
    a. Choose a small integer k
    b. Choose a sequence of k random Gaussian vectors $\left\{\vec{w}_i\right\}_{i=1}^{k}$
    c. Calculate $y_i = (\mathbf{I} - \mathbf{Q}_y\mathbf{Q}_y^T)\mathbf{A}x_i$ for all i where $\mathbf{Q}$ is the $n\times r$ matrix identified in the steps above.

4. If $r < k$, continue. Otherwise, add more matrix vector products in step 1 and repeat steps 2 and 3
5. Calculate $p_i = \mathbf{A}^T q_i$ for all i
6. Using the p_i and q_i vectors, a low rank approximation of the form $\mathbf{A} = \mathbf{USV}^T$ can be calculated as shown in the appendix of [9].

It has been shown in other works [12] that the effective rank r can be determined with at least $1 - 10^{-k}$ probability when $\mathbf{Q}$ satisfies the following criterion:

$$\|(\mathbf{I} - \mathbf{QQ}^T)\mathbf{A}\| \leq \epsilon/(10\sqrt{2/\pi}) \tag{6}$$

where ϵ is the user specified error allowance. In real applications, these ideas can be applied by replacing the matrix operator with a computational model. Let the computational model of interest be described by a vector valued function:

$$y = \Theta(x)$$

where $y = \in \mathbb{R}^m$ and $x = \in \mathbb{R}^n$. The goal of this methodology is to compute the entire set of derivatives for a given order by reducing the dimensions of the problem and thus reducing the computational and storage requirements. First, the case where $m = 1$, i.e. a single-valued model, will be considered. A general function $\Theta(x)$ can be expanded around a

reference point x_0 as follows (without loss of generality, assume that $x_0 = 0$ and $\Theta(x_0) = 0$ in order to simplify the representation):

$$\Theta(\boldsymbol{x}) = \sum_{k=1}^{\infty} \sum_{j_1,..,j_l,...,j_k=1}^{n} \psi_1\left(\beta_{j_1}^{(k)T} x\right) ... \psi_l\left(\beta_{j_l}^{(k)T} x\right) ... \psi_k\left(\beta_{j_k}^{(k)T} x\right) \tag{7}$$

where $\{\psi_l\}_{l=1}^{\infty}$ can be any kind of scalar functions. The outer summation over the variable k goes from 1 to infinity. Each term represents one order of variation, e.g. $k=1$ represents the first order term; $k=2$, the second order terms. For the case of $\psi_l(\Theta) = \Theta$, the k^{th} term reduces to the k^{th} term in a multi-variable Taylor series expansion. The inside summation for the k^{th} term consists of k single valued functions $\{\psi_l\}_{l=1}^{\infty}$ that are multiplying each other. The arguments for the $\{\psi_l\}_{l=1}^{\infty}$ functions are scalar quantities representing the inner products between the vector $\boldsymbol{x}$ and n vectors $\left\{\beta_{j_l}^{(k)}\right\}_{j_l=1}^{n}$ which span the parameters space. The superscript (k) implies that a different basis is used for each of the *k*-terms, i.e. one basis is used for the first-order term, another for the second-order term and so on.

Any input parameter variations that are orthogonal to the range formed by the collection of the $\left\{\beta_{j_l}^{(k)}\right\}$ vectors will not produce changes in the output response, i.e. the value of the derivative of the function will not change. If the $\left\{\beta_{j_l}^{(k)}\right\}$ vectors span a subspace of dimension *r* as opposed to *n*, then effective number of input parameters can be reduced

from *n* to *r*. The mathematical range can be determined by using only the first-order derivatives.

Differentiating Eq. (7) with respect to *x* gives:

$$\vec{\nabla}\Theta(\boldsymbol{x}) = \sum_{k=1}^{\infty} \sum_{j_1,..,j_l,...,j_k=1}^{n} \psi_1\left(\beta_{j_1}^{(k)T}x\right) \ldots \psi_{l-1}\left(\beta_{j_{l-1}}^{(k)T}x\right)\psi_l'\left(\beta_{j_l}^{(k)T}x\right)\psi_{l+1}\left(\beta_{j_{l+1}}^{(k)T}x\right) \ldots \psi_k\left(\beta_{j_k}^{(k)T}x\right) \ldots \quad (8)$$

where $\psi_l'\left(\beta_{j_l}^{(k)T}x\right)\beta_{j_l}^{(k)}$ is the derivative of the term $\psi_l\left(\beta_{j_l}^{(k)T}x\right)$. Eq. (3) can be reinterpreted to show that the gradient of the function is a linear combination of the $\{\beta_{j_l}^{(k)}\}$ vectors:

$$\vec{\nabla}\Theta(x) = \sum_{k=1}^{\infty} \sum_{j_1,..,j_l,...,j_k=1}^{n} \chi_{j_l}^{(k)}\beta_{j_l}^{(k)} = \begin{bmatrix}\beta_1^{(1)} & \ldots & \beta_{j_l}^{(k)} & \ldots\end{bmatrix}\begin{bmatrix}\alpha_1^{(1)} \\ \vdots \\ \alpha_{j_l}^{(k)} \\ \vdots\end{bmatrix} = \mathbf{B}\chi \quad (9)$$

where

$$\chi_{j_l}^{(k)} = \psi_1\left(\beta_{j_1}^{(k)T}x\right) \ldots \psi_{l-1}\left(\beta_{j_{l-1}}^{(k)T}x\right)\psi_l'\left(\beta_{j_l}^{(k)T}x\right)\psi_{l+1}\left(\beta_{j_{l+1}}^{(k)T}x\right) \ldots \psi_k\left(\beta_{j_k}^{(k)T}x\right) \quad (10)$$

In a typical application, the **B** matrix will not be known beforehand. It is only necessary to know the rank *r* of **B** which can be accomplished using the rank finding algorithm (RFA).

After determining the effective rank, it can be seen that the function only depends on *r* effective dimensions and can be reduced to simplify the calculation. The reduced model only requires the use of the subspace that represents the range of **B**, of which there are infinite possible bases.

This concept will now be expanded to a multi-response model. The q^{th} response of the model and its derivative are given by:

$$\Theta_q(x) = \sum_{k=1}^{\infty} \sum_{j_1,..,j_l,...,j_k=1}^{n} \psi_1\left(\beta_{j_{1,q}}^{(k)T} x\right) ... \psi_l\left(\beta_{j_{l,q}}^{(k)T} x\right) ... \psi_k\left(\beta_{j_k,q}^{(k)T} x\right) \tag{11}$$

$$\vec{\nabla}\Theta_q(x) = \sum_{k=1}^{\infty} \sum_{j_1,..,j_l,...,j_k=1}^{n} \psi_1\left(\beta_{j_{1,q}}^{(k)T} x\right) ... \psi_l'\left(\beta_{j_{l,q}}^{(k)T} x\right) ... \psi_k\left(\beta_{j_k,q}^{(k)T} x\right) \beta_{j_l,q}^{(k)} \tag{12}$$

The active subspace of the overall model must contain the contributions of each individual response. The matrix **B** will contain the $\left\{\beta_{j_l,q}^{(k)}\right\}$ vectors for all orders and responses. To determine a low rank approximation, a pseudo response $\Theta^{pseudo}(x)$ will be defined as a linear combination of the *m* responses:

$$\Theta^{pseudo}(x) = \sum_{q=1}^{m} \gamma_q \sum_{k=1}^{\infty} \sum_{j_1,..,j_l,...,j_k=1}^{n} \psi_1\left(\beta_{j_{1,q}}^{(k)T} x\right) \ldots \psi_l\left(\beta_{j_{l,q}}^{(k)T} x\right) \ldots \psi_k\left(\beta_{j_k,q}^{(k)T} x\right) \quad (13)$$

where γ_q are randomly selected scalar factors. The gradient of the pseudo response is:

$$\vec{\nabla}\Theta^{pseudo}(x) = \sum_{q=1}^{m} \gamma_q \sum_{k=1}^{\infty} \sum_{j_1,..,j_l,...,j_k=1}^{n} \psi_1\left(\beta_{j_{1,q}}^{(k)T} x\right) \ldots \psi_{l-1}\left(\beta_{j_{l-1,q}}^{(k)T} x\right) \psi_{l+1}\left(\beta_{j_{l+1,q}}^{(k)T} x\right) \ldots \psi_l'\left(\beta_{j_l,q}^{(k)T} x\right) \beta_{j_l,q}^{(k)} \quad (14)$$

Calculating derivatives of the pseudo response as opposed to each individual response provides the necessary derivative information while saving considerable computational time for large models with many inputs and outputs.

CHAPTER 3 Implementation

3.1 Implementation on a General FORTAN Code

The computational model that is executed in the computer code of interest will be assumed to be of the following form:

$$y = \Theta(x) \tag{15}$$

where *y* is an $m \times 1$ vector whose components correspond to the *m* outputs of interest, *x* is an $n \times 1$ vector whose components correspond to the *n* inputs of interest, and Θ corresponds to the unknown operator acting on the inputs to produce output. No previous information of Θ is needed; only the ability to obtain ***y*** from ***x*** is necessary for this method to work. The next step is to define the pseudo response $\tilde{y}$:

$$\tilde{y} = \sum_{i=1}^{m} \gamma_i y_i \tag{16}$$

where y_i are the individual components of the response vector *y*. This will be done with an invasive definition in the computer code, preferably in the highest level routine. Once the pseudo response has been defined, the reverse mode of OpenAD will be utilized to obtain derivatives of the pseudo response with respect to each input variable. Once an executable

form of the code is created to calculate the $n \times 1$ vector $\frac{d\tilde{y}}{dx}$, the active subspace of the model will be found by generating a set of derivative vectors, which span the union of all single-responses active subspace. The code will be run using a random set of inputs, $x^{(1)}, \ldots, x^{(k)}$. The responses will then be collected into the columns of an $n \times k$ matrix **G**:

$$\mathbf{G} = \left[\left(\frac{d\tilde{y}}{dx}\right)_{x^{(1)}} \left(\frac{d\tilde{y}}{dx}\right)_{x^{(2)}} \ldots \left(\frac{d\tilde{y}}{dx}\right)_{x^{(k)}}\right] \tag{17}$$

Now, the effective rank of **G** will be found via the Rank Finding Algorithm (RFA).

$$\mathbf{G} = \mathbf{QR} = [\mathbf{Q}_r \mathbf{Q}_2]\mathbf{R} \tag{18}$$

where the sub-matrix $\mathbf{Q} \in \mathbb{R}^{n \times r}$ contains only the first *r* columns of **Q**. The rank is selected in the RFA to satisfy a user defined error metric such that

$$\|(\mathbf{I} - \mathbf{Q}_r \mathbf{Q}_r^T)\mathbf{G}\| \leq \epsilon \tag{19}$$

where $\|\cdot\|$ is the l_2 norm. The columns of the $\mathbf{Q}_r$ matrix will be used to define the pseudo inputs that will be coded into a version of the code that will be used with Rapsodia to calculate derivatives of the output responses with respect to the pseudo inputs of a desired order. The pseudo inputs are defined as:

$$\tilde{x} = \mathbf{Q}_r^T x \tag{20}$$

In order to satisfy the logical order of derivative calculation by means of the chain rule of differentiation $\left(\frac{dy}{dx}=\frac{dy}{d\tilde{x}}\frac{d\tilde{x}}{dx}\right)$, x will be defined in the code in terms of $\tilde{x}$ and the columns of $\mathbf{Q}_r$:

$$x = \mathbf{Q}_r \tilde{x} \tag{21}$$

This line is what must be inserted into the code, preferably in the top level routine. Now Rapsodia will be used to create an executable version of the code that calculates $\left[\frac{d^{(O)}y}{d\tilde{x}_1^{(o_1)}...d\tilde{x}_n^{(o_n)}}\right]$ where O is the desired derivative order and $o_1+...+o_n=O$. For first order calculations, the full derivatives can be recovered utilizing the following relation that comes from the chain rule of differentiation:

$$\frac{dy_i}{dx_j}=\sum_{k=1}^{r}\frac{dy_i}{d\tilde{x}_k}\frac{d\tilde{x}_k}{dx_j}=\sum_{k=1}^{r}\frac{dy_i}{d\tilde{x}_k}q_{jk} \tag{22}$$

The following matrix operation can also be used: $\mathbf{J}=\tilde{\mathbf{J}}\mathbf{Q}_r^T$ where $\tilde{\mathbf{J}}$ is the matrix of derivatives with respect to the pseudo variables. From Eq.(20), it can easily be determined that $\frac{d\tilde{x}_k}{dx_j}=q_{jk}$. For orders greater than 1, the mixed derivatives play a part in the reconstruction. The following equation shows how the second order derivatives are reconstructed on an element by element basis for a non-mixed derivative:

$$\frac{d^2 y_i}{dx_j^2} = \sum_{k,l} \frac{d^2 y_i}{d\tilde{x}_k d\tilde{x}_l} q_{jk} q_{jl} \tag{23}$$

The following matrix operation can also be used: $\mathbf{H}_i = \mathbf{Q}_r \tilde{\mathbf{H}}_i \mathbf{Q}_r^T$ where $\tilde{\mathbf{H}}_i$ is the matrix of second order derivatives of response *i* with respect to the pseudo variables. For example, the second derivative of output $i = 1$ with respect to input $j = 1$ in a problem with an effective rank of $r = 2$ would be recovered by $\frac{d^2 y_1}{dx_1^2} = \frac{d^2 y_1}{d\tilde{x}_1^2} q_{11}^2 + 2\frac{d^2 y1}{d\tilde{x}_1 d\tilde{x}_2} q_{11} q_{12} + \frac{d^2 y_1}{d\tilde{x}_2^2} q_{12}^2$. Mixed derivatives would be recovered by:

$$\frac{d^2 y_i}{dx_j dx_g} = \sum_{k,l} \frac{d^2 y_i}{d\tilde{x}_k d\tilde{x}_l} q_{jk} q_{gl} \tag{20}$$

In the same case as before ($r = 2$), $\frac{d^2 y_1}{dx_1 dx_1} = \frac{d^2 y_1}{d\tilde{x}_1^2} q_{11} q_{12} + \frac{d^2 y_1}{d\tilde{x}_1 d\tilde{x}_2}\left(q_{11} q_{22} + q_{12} q_{21}\right) + \frac{d^2 y_1}{d\tilde{x}_2^2} q_{12} q_{22}$.

It can be inferred that for derivatives of order *O*, the non mixed derivatives can be recovered using the following expression:

$$\frac{d^{(O)} y_i}{dx_j^{(O)}} = \sum_{k,l,\ldots o} \frac{d^{(O)} y_i}{d\tilde{x}_k d\tilde{x}_l \ldots d\tilde{x}_z} q_{jk} q_{jl} \ldots q_{jz} \tag{21}$$

where the total number of different indices *k, l,..z* is *O* and each index runs from 0 to *n* . The mixed derivatives are:

$$\frac{d^{(O)}y_i}{dx_j \ldots dx_g} = \sum_{k,l \ldots o} \frac{d^{(O)}y_i}{d\tilde{x}_k \ldots d\tilde{x}_z} q_{jk} \ldots q_{gz} \tag{22}$$

The algorithm detailed in this section can be summarized as follows:

1. Define the pseudo responses in the top level routine (Eq. 11)
2. Compile an executable using OpenAD to compute $\frac{d\tilde{y}}{dx}$
3. Run the $\frac{d\tilde{y}}{dx}$ calculation *k* times and assemble the output into a matrix **G** (Eq. [12])
4. Determine the effective rank *r* using the RFA
5. Calculate a QR decomposition of $\tilde{\mathbf{G}}$ and keep the first *r* columns of **Q** in $\mathbf{Q}_r$ (Eq. [15])
6. Define the inputs in terms of the pseudo inputs and $\mathbf{Q}_r$ in the top level routine (Eq. [17])
7. Reconstruct the full derivatives (Eq.[18])

The steps in the RFA are:

1. Calcuate the SVD of **G**: $\mathbf{G} = \mathbf{USV}^T$
2. Set the smallest singular value to 0 and keep the remaining *r* singular values: $\sigma_p = 0$
3. Calculate an approximation of **G**: $\tilde{\mathbf{G}} = \mathbf{US}_r\mathbf{V}^T$

Repeat steps 2 and 3 until $\frac{\|\mathbf{G} - \tilde{\mathbf{G}}\|}{\|\mathbf{G}\|} < \varepsilon$ is no longer satisfied

CHAPTER 4 Examples and Test Cases

4.1 Two-minute AD Examples

To quickly introduce the two AD tools used in this method, 'Two-minute' examples will be shown to illustrate the basics of how to use each tool on a subroutine containing an easily differentiated function. The examples follow the basic style given in the examples used in the OpenAD/F Manual [10] and Rapsodia Manual [5]. The manuals should be consulted for a more in depth presentation of the inner workings of each tool.

4.1.1 OpenAD/F Example

A simple subroutine will be used to illustrate the calculation of derivatives using OpenAD. Figure 4.1 shows the subroutine that will be used in this example. The function that the subroutine models consists of two outputs (`y`) and three inputs (`x`). The parameters `a`, and `b` are simply scalar factors used in the calculation. The only additions to this code that are required by OpenAD are the tags identifying the independent (`!$openad INDEPENDENT(x)`) and dependent variables (`!$openad DEPENDENT(y)`). The goal is to calculate the derivatives of each output with respect to each input. A main program called `driver` will be used with subroutine `head` in order to extract the derivatives. Figure 4.2 shows the main program `driver`.

```
subroutine head(x,y)
  double precision, dimension(3) :: x
  double precision, dimension(2) :: y
  double precision :: a,b

  a=2.0d0
  b=3.0d0

!$openad INDEPENDENT(x)
  y(1)=a*x(1)**2+sin(b*x(2))
  y(2)=log(a*x(1))+cos(b*x(2))+1/x(3)
!$openad DEPENDENT(y)

end subroutine
```

Figure 4.1: Simple subroutine for 'Two-minute' forward OpenAD example

```
program driver
  use OAD_active
  implicit none
  external head
  type(active) :: x(3), y(2)
  x(1)%v=1.0D0
  x(2)%v=2.0D0
  x(3)%v=3.0D0

     x(1)%d=1.0D0
     x(2)%d=0.0D0
     x(3)%d=0.0D0
     call head(x,y)
     print *, 'driver running for x =',x%v
     print *, '            yields y =',y%v,' dy/dx =',y%d

     x(1)%d=0.0D0
     x(2)%d=1.0D0
     x(3)%d=0.0D0
     call head(x,y)
     print *, 'driver running for x =',x%v
     print *, '            yields y =',y%v,' dy/dx =',y%d

     x(1)%d=0.0D0
     x(2)%d=0.0D0
     x(3)%d=1.0D0
     call head(x,y)
     print *, 'driver running for x =',x%v
     print *, '            yields y =',y%v,' dy/dx =',y%d

end program driver
```

Figure 4.2: Main program used in the 'Two-minute' forward OpenAD example

The active type declaration shown in line 5 must be used in this top level routine for the active variables. The active type will give each active variable two parts, a normal value (indicated by %v) and a derivative part (%d). Lines 6, 7 and 8 initialized the value parts of `x`. Before the subroutine is called, one input will be chosen for derivative calculation. This variable's derivative part will be 'seeded' with a value of 1 while all others are assigned 0. For the first call of `head`, this can be viewed as starting with $\frac{dx_1}{dx_1} = 1$, $\frac{dx_2}{dx_1} = 0$ and $\frac{dx_3}{dx_1} = 0$ as is shown in Table 2-2: Forward mode evaluation trace. For subsequent calls of `head`, the seed will change to the other inputs. The basic call to invoke the OpenAD tool for a forward transformation is `openad -m f [file name]` which will create a new transformed file. After transforming and compiling all the necessary files for the example given above, the following output given in Figure 4.3: Output for the 'Two-minute' forward OpenAD example is returned.

The forward mode allows for the calculation of derivatives of each output with respect to one input at a time. As a quick check, the first set of derivative values given are $\frac{dy_1}{dx_1}$ and $\frac{dy_2}{dx_1}$. The derivatives evaluated for $x_1 = 1.0$ can easily be found analytically: $\frac{d}{dx_1}[2x_1^2 + \sin(3x_2)] = 4x_1 = 4(1.0) = 4.0$ and $\frac{d}{dx_1}\left[\ln(2x_1) + \cos(3x_2) + \frac{1}{x_3}\right] = \frac{1}{x_1} = 1.0$. An example that requires more precision is $\frac{dy_1}{dx_2} = 3\cos(3x_2)$. When the analytically derived expression is evaluated in Matlab for $x_2 = 2.0$, the first 14 digits match exactly.

The `Makefile` shown in Figure 4.4: `Makefile` contents for compiling and linking the 'Two-minute' forward OpenAD example contains all the steps for transforming and

```
driver running for x =   1.00000000000000   2.00000000000000   3.00000000000000
            yields y =   1.72058450180107   1.98665080054364
               dy/dx =   4.00000000000000   1.00000000000000

driver running for x =   1.00000000000000   2.00000000000000   3.00000000000000
            yields y =   1.72058450180107   1.98665080054364
               dy/dx =   2.88051085995110   0.838246494596778

driver running for x =   1.00000000000000   2.00000000000000   3.00000000000000
            yields y =   1.72058450180107   1.98665080054364
               dy/dx =   0.000000000000000E+000 -0.111111111111111
```

Figure 4.3: Output for the 'Two-minute' forward OpenAD example

```
ifndef F90C
F90C=ifort
endif
RTSUPP=w2f__types OAD_active
driver: $(addsuffix .o, $(RTSUPP)) driver.o head.prepped.pre.xb.x2w.w2f.post.o
       ${F90C} -o $@ $^
head.prepped.pre.xb.x2w.w2f.post.f90 $(addsuffix .f90, $(RTSUPP)) : toolChain
toolChain : head.prepped.f90
       openad -c -m f $<
%.o : %.f90
       ${F90C} -o $@ -c $<
clean:
       rm -f ad_template* OAD_* w2f__*  iaddr*
       rm -f head.prepped.pre* *.B *.xaif *.o *.mod driver driverE *~
.PHONY: clean toolChain
# the following include has explicit rules that could replace the openad script
include MakeExplRules.inc
```

Figure 4.4: `Makefile` contents for compiling and linking the 'Two-minute' forward OpenAD example

compiling all the necessary files for the example shown above. After running the `Makefile`, an executable named 'driver' is created. The Fortran compiler used is ifort.

To operate in the reverse mode, the prepped subroutine does not require changes. The main program `driver` will require some slight modifications. Figure 4.5 contains the reverse mode `driver`.

The main difference between the forward and reverse mode main programs is that the seeding is done with the outputs in the reverse mode. The derivatives of one output with respect to all inputs are calculated at once in the reverse mode. The reverse mode flag must be used when executing the OpenAD tool (`openad -m rj [file name]`). Figure 4.6 gives the output for the reverse mode example.

The outputs exactly match those given by the forward mode. When using OpenAD on more complex codes, the same basic methods presented in these examples are used.

4.1.2 Rapsodia Example

Rapsodia will now be used to calculate first, second, and third order derivatives of the function in the subroutine from the previous example. The subroutine itself does not require many changes. Rapsodia does not require the tags that OpenAD used, but the active variables must be changed to the Rapsodia active type in both the subroutine and main program. The Rapsodia prepped subroutine is shown in Figure 4.7.

```
program driver
  use OAD_active
  use OAD_rev
  implicit none
  external head
  type(active) :: x(3), y(2)
  x(1)%v=1.0D0
  x(2)%v=2.0D0
  x(3)%v=3.0D0

  y(1)%d=1.0D0
  y(2)%d=0.0D0
  our_rev_mode%tape=.TRUE.
  call head(x,y)
  print *, 'driver running for x =',x%v
  print *, '            yields y =',y%v,' dy/dx =',x%d

  y(1)%d=0.0D0
  y(2)%d=1.0D0
  our_rev_mode%tape=.TRUE.
  call head(x,y)
  print *, 'driver running for x =',x%v
  print *, '            yields y =',y%v,' dy/dx =',x%d

end program driver
```

Figure 4.5: Example reverse mode main program

```
driver running for x =  1.00000000000000  2.00000000000000   3.00000000000000
          yields y =  1.72058450180107  1.98665080054364
             dy/dx =  4.00000000000000  2.88051085995110  0.000000000000000E+000

driver running for x =  1.00000000000000  2.00000000000000   3.00000000000000
            yields y =  1.72058450180107  1.98665080054364
               dy/dx =  1.00000000000000  0.838246494596778 -0.111111111111111
```

Figure 4.6: Reverse mode example output

```
subroutine head(x,y)
  INCLUDE 'RAinclude.i90'
  TYPE(RARealD) :: x(3),y(2)
  double precision :: a,b

  a=2.0d0
  b=3.0d0

  y(1)=a*x(1)**2+sin(b*x(2))
  y(2)=log(a*x(1))+cos(b*x(2))+1/x(3)

end subroutine
```

Figure 4.7: Rapsodia prepped subroutine for the ‘Two-minute’ example

The main program logic that is required for running Rapsodia on this subroutine is more involved than OpenAD, but it can mostly be reused easily for any general subroutine or program. Figure 4.8 gives the complete main program that was used to run the Rapsodia example.

Lines 2-9 contain the Rapsodia specific variable declarations. These variables are initialized in lines 17-23 by calling Rapsodia specific routines that are generated and must be compiled with the program. The desired independent variables are set in the loops of lines 31-35 with the call to RAset. The derivatives are extracted and output in lines 45-58, which are looped over for each output. The call to RAget sets the dependent variables. A `Makefile` is shown in Figure 4.9 which contains the instructions for generating the necessary files from the Rapsodia library and linking them to the subroutine and main program. The line `${RAPSODIAROOT}/Generator/generate.py -d 3 -o 1 -f $(GEN_DIR)` must be set with the appropriate number of directions (d) and derivative order (o). For first order derivatives of this example, the values are 3 and 1, respectively. Again, an executable named 'driver' is created to run the program.

Figure 4.10 through Figure 4.12 contain the results for first through third order derivatives. The only changes required to obtain a different order of derivatives are rebuilding the program after changing the order number in the main program and `Makefile` and changing the number of directions in the `Makefile`. The results are printed with the appropriate multi-index of the derivative order. For example, `[1][2][0]` corresponds to a

```
PROGRAM DRIVER
  INCLUDE 'RAinclude.i90'
  USE higherOrderTensorUtil
  IMPLICIT NONE
  INTEGER :: O, DIRS, max_order
  TYPE(higherOrderTensor) :: T
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: SeedMatrix
  REAL(KIND=RAdKind), DIMENSION(:,:), ALLOCATABLE :: TaylorCoefficients
  REAL(KIND=RAdKind), DIMENSION(:), ALLOCATABLE :: CompressedTensor

  TYPE(RARealD) :: x(3),y(2)
  integer :: j,k,i,n

  n=3
  O=3

  CALL setNumberOfIndependents(T, n)
  CALL setHighestDerivativeDegree(T, O)
  DIRS = getDirectionCount(T)
  ALLOCATE(SeedMatrix(n, DIRS))
  CALL getSeedMatrix(T, SeedMatrix)
  ALLOCATE(TaylorCoefficients(O, DIRS))
  ALLOCATE(CompressedTensor(DIRS))

  WRITE (*,'(A,I3.1)') 'Number of directions = ', DIRS

  x(1)=1.0D0
  x(2)=2.0D0
  x(3)=3.0D0

  DO j=1,n
    DO i = 1, DIRS
      CALL RAset(x(j), i, 1, REAL(SeedMatrix(j, i), KIND=RAdKind))
    END DO
  END DO

  CALL head(x,y)

   DO i=1,2
     WRITE(*,*) 'y(',i,') = ', y(i)%v
   END DO

  DO k=1,2

   DO i = 1, O
     DO j = 1, DIRS
       CALL RAget(y(k), j, i, TaylorCoefficients(i, j))
     END DO
   END DO
   CALL setTaylorCoefficients(T, TaylorCoefficients)
   CALL getCompressedTensor(T, O, CompressedTensor)
   DO i = DIRS, 1,-1
     WRITE (*,'(A)',ADVANCE='NO') 'Y'
     DO j = 1, n
       WRITE (*,'(A,I1,A)',ADVANCE='NO') '[', SeedMatrix(j, i), ']'
     END DO
     WRITE (*,'(A,E25.17E3)') ' = ', CompressedTensor(i)
   END DO

  END DO

 DEALLOCATE(CompressedTensor)
 DEALLOCATE(TaylorCoefficients)
 DEALLOCATE(SeedMatrix)

END PROGRAM DRIVER
```

Figure 4.8: Main program for the ‘Two-minute’ Rapsodia example

```
ifndef RAPSODIAROOT
 $(error "environment variable  RAPSODIAROOT undefined")
endif
include ${RAPSODIAROOT}/Makefile.inc

default: driver
	./$^

GEN_DIR=RALib

RA_EXTRAS=${RAPSODIAROOT}/hotF90
IPATH+=-I$(GEN_DIR) $(MODSEARCHFLAG)$(GEN_DIR) $(MODSEARCHFLAG)$(RA_EXTRAS)

OBJS= \
$(addprefix $(RA_EXTRAS)/, $(addsuffix .o,$(HOTF90NAMES))) \
driver.o

driver: sources $(OBJS)
	$(F90C) $(FFLAGS) $(IPATH) -o $@ $(OBJS) $(GEN_DIR)/libRapsodia.a

sources : FORCE
	${RAPSODIAROOT}/Generator/generate.py -d 3 -o 1 -f $(GEN_DIR)
	cd $(GEN_DIR) && $(MAKE)

FORCE:

clean:
	rm -rf $(GEN_DIR) *.o *.mod driver driver.out

.PHONY: default sources clean
```

Figure 4.9: `Makefile` contents for compiling and linking the 'Two-minute' Rapsodia example (first order)

derivative that is first order in x_1 and second order in x_2.

Like the OpenAD example, these same basic steps are used in the examples and test cases that follow.

4.2 Matrix-Vector Product Example

This example will now implement the reduction method on a pre-constructed low rank matrix operator. To start, consider a random $m \times m$ matrix **A**. The matrix will be modified so that it is rank deficient by zeroing $m - r$ singular values. To visualize a rank deficient matrix, consider a 2D plane in a 3D space. Consider m different vectors that live inside the plane. Since the plane is 2 dimensional, any vector inside it can be expressed as a linear combination of two vectors only (any two independent vectors that live inside the plane). In this case the matrix **A** would have rank equal to 2 only and not m. In order to reduce this problem, one needs to find any two vectors that live in that plane. The simplest way to do this is to run the reverse mode twice, once for $\frac{dy_1}{dx}$ and once for $\frac{dy_2}{dx}$.

However, this does not guarantee that $\frac{dy_1}{dx}$ and $\frac{dy_2}{dx}$ are linearly independent vectors. To get around that, a new 'pseudo' response will be defined. Let $\tilde{y}_j = \sum_{i=1}^{m} \gamma_{ij} y_i,\ j = 1,2$. The new variables are simply weighted (the weights γ_{ij} can be picked randomly) sums of the original responses. Now differentiating the pseudo response gives:

```
Number of directions =   3
 y( 1 ) =    1.7205845018010741
 y( 2 ) =    1.9866508005436445

Y[1][0][0] =  0.40000000000000000E+001
Y[0][1][0] =  0.28805108599510980E+001
Y[0][0][1] =  0.00000000000000000E+000

Y[1][0][0] =  0.10000000000000000E+001
Y[0][1][0] =  0.83824694596777758E+000
Y[0][0][1] = -0.11111111111111110E+000
```

Figure 4.10: Rapsodia example first order results

```
Number of directions =   6
 y( 1 ) =    1.7205845018010741
 y( 2 ) =    1.9866508005436445

Y[2][0][0] =  0.40000000000000000E+001
Y[1][1][0] =  0.00000000000000000E+000
Y[1][0][1] =  0.00000000000000000E+000
Y[0][2][0] =  0.25147394837903327E+001
Y[0][1][1] =  0.00000000000000000E+000
Y[0][0][2] =  0.00000000000000000E+000

Y[2][0][0] = -0.10000000000000000E+001
Y[1][1][0] =  0.00000000000000000E+000
Y[1][0][1] =  0.00000000000000000E+000
Y[0][2][0] = -0.86415325798532940E+001
Y[0][1][1] =  0.00000000000000000E+000
Y[0][0][2] =  0.74074074074074070E-001
```

Figure 4.11: Rapsodia example second order results

```
Number of directions =  10
 y( 1 ) =    1.7205845018010741
 y( 2 ) =    1.9866508005436445

Y[3][0][0] =  0.0000000000000000E+000
Y[2][1][0] =  0.46678715293069217E-006
Y[2][0][1] =  0.0000000000000000E+000
Y[1][2][0] =  0.37342972254439388E-005
Y[1][1][1] =  0.46678715304171448E-006
Y[1][0][2] =  0.0000000000000000E+000
Y[0][3][0] = -0.25924587937029660E+002
Y[0][2][1] =  0.37342972269982511E-005
Y[0][1][2] =  0.46678715293069217E-006
Y[0][0][3] =  0.0000000000000000E+000

Y[3][0][0] =  0.19999992437660694E+001
Y[2][1][0] = -0.15225116500872105E-006
Y[2][0][1] = -0.28675537055988798E-006
Y[1][2][0] =  0.10506924744690949E-005
Y[1][1][1] =  0.10116055770836851E-006
Y[1][0][2] = -0.25341172271708956E-007
Y[0][3][0] = -0.75442155987740112E+001
Y[0][2][1] =  0.10880373606525495E-005
Y[0][1][2] =  0.14650791824166731E-006
Y[0][0][3] = -0.74074046065409974E-001
```

Figure 4.12: Rapsodia example third order results

$$\frac{d\tilde{y}_j}{dx} = \frac{d}{dx}\sum_{i=1}^{m}\gamma_{ij}y_i = \sum_{i=1}^{m}\gamma_{ij}\frac{dy_i}{dx} \tag{24}$$

Equation (24) is simply a random sum of the rows of the matrix **A**. If the weights are selected randomly, there is a high probability that $\frac{d\tilde{y}_1}{dx}$ and $\frac{d\tilde{y}_2}{dx}$ will be independent.

This approach can be generalized for a $m \times n$ matrix with rank r. Let the model be described by $y = \mathbf{A}x$. For now assume that the matrix is random and constructed to be rank deficient with known rank $r < \min(m, n)$. A pseudo response will be constructed as shown above. Using reverse mode OpenAD, r sets of derivatives of the pseudo response will be taken. Figure 4.13 shows the code for a matrix vector product with the pseudo response definition.

The $n \times r$ collection of derivatives that are obtained, $\tilde{\mathbf{J}}_1 = \left[\frac{d\tilde{y}_1}{dx} \quad ... \quad \frac{d\tilde{y}_r}{dx}\right]^T$, are now used to define pseudo inputs of the form $\tilde{x} = \tilde{\mathbf{J}}_1 x$. In order to keep the logical progression of variables for the OpenAD evaluation trace in the code, x will be defined in terms of $\tilde{x}$: $x = \tilde{\mathbf{J}}_1^{\mathrm{T}}\tilde{x}$. Forward mode OpenAD will then be used to obtain $\tilde{\mathbf{J}}_2 = \left[\frac{dy_1}{d\tilde{x}} \quad ... \quad \frac{dy_r}{d\tilde{x}}\right]^T$ Figure 4.14 shows the code for a matrix vector product with the pseudo input definition.

The full derivatives $\frac{dy_i}{dx_j}$ (which in this case are the same as the **A** matrix) can be recovered by multiplying the reverse results by the forward results: $J = J_2 J_1^T = \begin{bmatrix} \frac{dy_1}{d\tilde{x}}^T \\ \vdots \\ \frac{dy_r}{d\tilde{x}}^T \end{bmatrix} \left[\frac{d\tilde{y}_1}{dx} \quad ... \quad \frac{d\tilde{y}_r}{dx}\right]$.

```
!$openad INDEPENDENT(x)

  do i=1,m
    do j=1,n
      y(i)=y(i)+A(i,j)*x(j)
    end do
  end do

  do j=1,r
    do i=1,m
        y_pseudo(j)=y_pseudo(j)+y(i)*gamma(i,j)
    end do
  end do

!$openad DEPENDENT(y_pseudo)
```

Figure 4.13: Matrix-vector product code with pseudo response definition

```
!$openad INDEPENDENT(x_pseudo)
  do j=1,r
    do i=1,n
        x(i)=x(i)+x_pseudo(j)*J_pseudo(i,j)
    end do
  end do

do i=1,m
   do j=1,n
      y(i)=y(i)+A(i,j)*x(j)
   end do
 end do
!$openad DEPENDENT(y)
```

Figure 4.14: Matrix-vector product code with pseudo input definition

In realistic problems, the rank will not be absolute, i.e. all singular values will be non-zero, but the magnitudes of the singular values will be distributed such that an effective rank can be determined. Another random rank deficient matrix will be used to demonstrate how the rank finding algorithm works. A 600×500 random matrix with the singular value distribution shown in Figure 4.15 will be considered for this example.

The singular value distribution shows that of the 500 singular values, only about 60 actually contribute significantly. A safe estimate of the effective rank would be 100. Obtaining the unreduced derivatives will yield the original matrix. By doing an SVD on this output and taking the first 50 columns of the $\mathbf{V}$ matrix, $\mathbf{V}_r$, pseudo inputs can be defined as shown above. The 600×100 output of the derivatives of the responses with respect to the pseudo inputs will be multiplied by $\mathbf{V}_r$ to obtain the reduced approximation of $\mathbf{A}$. Using an effective rank of 100 yields a maximum matrix element relative error of 0.130%.

To advance the demonstration of this method further towards actual engineering codes, the next examples will demonstrate when the rank must be determined by means of sampling the derivatives and using the rank finding algorithm.

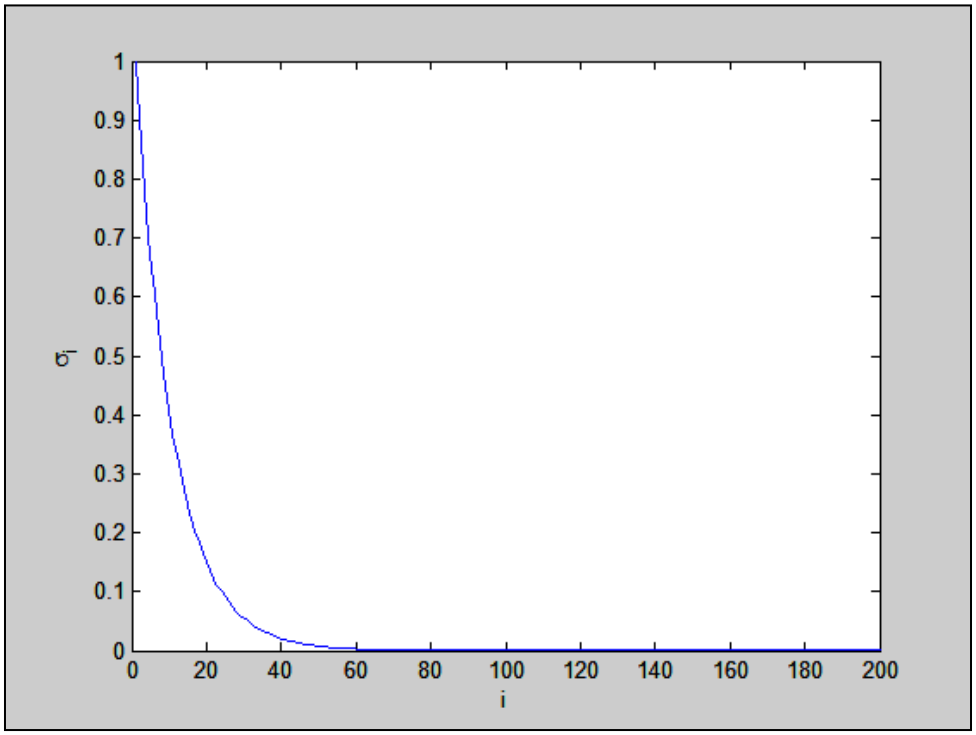

Figure 4.15: Singular value distribution for the matrix-vector product example

4.3 Scalar-valued Model Example

First, a scalar valued function will be considered. The model that will be considered is:

$$y = a^T x + (b^T x)^2 + \sin(c^T x) + \frac{1}{1 + e^{d^T x}} \tag{25}$$

where *x, a, b, c,* and *d* are $n \times 1$ vectors, making *y* a scalar valued function. A simple subroutine named 'head' was written to calculate this model along with a simple main program called a 'driver' that calls the subroutine and is used to extract the derivatives. In order to prepare the code for use with OpenAD, tags must be inserted to identify the independent and dependent variables (typically formal parameters of 'head') to the code analysis. Within 'driver' the corresponding inputs and outputs, passed as actual parameters to 'head', have to be declared with the OpenAD active type to carry the derivative values. After the code is prepared in this fashion for use with OpenAD, it generates a transformed version of 'head', which then is compiled together with 'driver' and calculates the first order derivatives of *y* with respect to the vector *x*.

A Python script was written to execute the subspace identification algorithm with the compiled executable code. The script takes a guess *k* for the effective rank and runs the code for *k* random input vectors *x*. Within the Python script, the responses are collected into a matrix **G** of dimension $n \times k$. **G** is then QR factorized and the first *r* columns of **Q** are used to evaluate if the error criterion is met. Additional samples of the derivatives are used also to

evaluate the error. If the error criterion is met, the first *r* columns of **Q** are written to a file to be used as input for the higher-order derivative computation with Rapsodia. Figure 4.16 shows the Python script used with this model. For the model above with $n = 50$ and random input vectors with 8 digits of precision for *a, b, c,* and *d* with an error criterion of $\varepsilon = 10^{-6}$, the effective rank was found to be $r = 3$.

A process similar to the one for OpenAD is used to prepare the program for use with Rapsodia. The active variables are identified via a similar manual type change in the 'driver' and the source transformation capabilities of OpenAD can be used to perform the type change in the 'head' subroutine. Because of the additional steps to determine the propagation directions and compute the derivative tensors, the logic in the 'driver' has additional steps for Rapsodia, but they follow a simple recipe and can be transferred between such driver programs with relative ease. For efficiency, the number of directions that the derivatives are calculated for along with the desired derivative order must be provided to the library generator implemented by Rapsodia. Once the library is generated, the type-changed 'head', the 'driver', and the library can be compiled and linked. For first order calculations, the number of directions is simply the number of input variables for which derivatives are calculated. Once the derivatives $\frac{dy}{d\tilde{x}}$ are calculated, the full derivatives can be reconstructed by multiplying the Rapsodia results by the $\mathbf{Q}_r$ matrix used as input. Using an effective rank of $r = 3$, the reconstructed derivatives were found to have relative errors on the order of 10^{-13} compared to results obtained from an unreduced Rapsodia calculation.

```
while (z>e):
       while (i<k):
              #run the forward driver
              status,output=commands.getstatusoutput('./driver ' + str(n) )
              print status
              #input to G from output of driver
              G[:,i]=numpy.genfromtxt('der_out.txt')
              j=0
              while (j<n):
                     In[j,5]=10*random.random()
                     j=j+1

              numpy.savetxt('Rinput.in.txt',In)
              i=i+1

       #calculate QR of G
       Q,R=linalg.qr(G)
       Qr=Q[:,0:k]
       #generate additional samples for error testing
       Gadd=numpy.zeros((n,k))
       p=0
       while (p<k):
              j=0
              while (j<n):
                     In[j,5]=10*random.random()
                     j=j+1

              numpy.savetxt('Rinput.in.txt',In)
              #run the forward driver
              status,output=commands.getstatusoutput('./driver ' + str(n) )
              print status
              #input to Gadd from output of driver
              Gadd[:,p]=numpy.genfromtxt('der_out.txt')
              p=p+1

       z=linalg.norm(  dot( numpy.eye(n)-dot(Qr,Qr.T),Gadd)  )
       print z
       k=k+1
       Gnew=numpy.zeros((n,k))
       Gnew[:,0:k-1]=G
       numpy.savetxt('Gnew',Gnew)
       G=numpy.zeros((n,k))
       G=Gnew
       numpy.savetxt('G',G)

#output Q to file for input
numpy.savetxt('Q.in.txt',Qr)
```

Figure 4.16: Python script used for the scalar-valued model example

Using Rapsodia to calculate second order derivatives simply involves changing the derivative order (o = 2) and the number of directions (d = 6) to regenerate the library and recompiling the code. The output can then be constructed into a matrix **H** of size $r \times r$ and the full derivatives can be recovered by: $\mathbf{Q}_r \mathbf{H} \mathbf{Q}_r^T$ which results in an $n \times n$ symmetric matrix. When the second order derivatives are calculated for the example above, there are only 6 directions required for an effective rank of 3 as opposed to 1275 directions for the full problem. The relative errors of the reduced derivatives are on the order of 10^{-12}.

Third order derivatives were also calculated using this example. The unreduced problem would require 22,100 directions while the reduced problem only requires 10. Relative errors were much higher for this case but still at a reasonable order of 10^{-6}. The relative errors for each derivative order are summarized in Table 4-1. The maximum unreduced relative error is the maximum relative error between the unreduced Rapsodia calculations and analytical results. The average unreduced relative errors are on the order of 10^{-15}. Note that for the third order calculations, not all values for the unreduced case were calculated due to the difficulty of obtaining all values at once. The full codes for this example are given in Appendix A.

Table 4-1: Results for the scalar-valued model example

Derivative Order	Unreduced Directions	Reduced Directions	Reduced Relative Error	Maximum Unreduced Relative Error
1	50	3	10^{-13}	1.49%
2	1275	6	10^{-12}	4.21%
3	22,100	10	10^{-6}	2.24%

4.4 Vector-valued Model Example

It is important to note here that in practice the derivatives are employed to construct a surrogate model that approximates the original function. Therefore, it is much more instructive to talk about the accuracy of the surrogate model employed rather than the accuracy of each derivative. This is illustrated in the MATWS test case by using an engineering model.

Problems with multiple outputs require a slightly different approach when determining the subspace. The following example will be considered:

$$y_1 = a^T x + (b^T x)^2 \quad y_2 = \sin(c^T x) + \frac{1}{1 + e^{d^T x}} \quad y_3 = (a^T x)(e^T x) \tag{26}$$

$$y_4 = 2^{e^T x} \quad y_5 = \frac{1}{(d^T x)^3}$$

In the OpenAD version of the subroutine, the responses will be combined into a pseudo response defined by the following:

$$\tilde{y} = \gamma_1 y_1 + \gamma_2 y_2 + \gamma_3 y_3 + \gamma_4 y_4 + \gamma_5 y_5 \tag{27}$$

where γ_i are randomly generated factors that are unique for each execution of the code. The derivatives that OpenAD will calculate are $\frac{d\tilde{y}}{dx}$, which is an $n \times 1$ vector. Following the same procedure for the single output case, the subspace identification script was run for n = 50 and $\varepsilon = 10^{-6}$. The effective rank was found to be r = 5.

The Rapsodia version of the code is executed in essentially the same manner with a loop over each output. The first, second, and third order derivatives were recovered for this example with errors that were of the same magnitudes as the single output case. For first order derivatives, the number of directions required in the unreduced problem is 50 for each output, making the total number of derivatives 250. The reduced case only requires 5 derivatives for each output, 25 in total. The total number of derivatives for the unreduced second and third order calculations were 6,375 and 110,500, respectively, compared to 15 and 35 for the reduced problem. The errors were on the same order as the scalar-valued model test case. The results are shown in Table 4-2.

Table 4-2: Results for the vector-valued model example

Derivative Order	Unreduced Directions	Reduced Directions	Reduced Relative Error
1	50	5	10^{-13}
2	1275	15	10^{-12}
3	22,100	35	10^{-6}

4.5 Important Directions

The next two examples will highlight some unique features of using this method for derivative calculations. First, the property of derivatives not changing for inputs that are orthogonal to the model subspace will be shown in an example. In order to visualize this concept, consider two three-dimensional vectors a and b that are the basis for a model subspace, shown in Figure 4.17. Derivatives are then calculated with a vector w as input. It can be shown that the derivatives are the same when calculated using a vector p that is a projection of w onto the plane spanned by a and b. This means that the components of w that are perpendicular to the plane spanned by a and b do not factor into the derivative calculation and can be removed as is done with the reduction techniques employed in the previous example problems.

A larger example will now be run to verify that these ideas hold true for a general problem space. The following function using 100 inputs and five vectors as the basis for the problem subspace.

$$y = (a^T x)^2 + (b^T x)(c^T x) + \frac{\sin(d^T x)}{e^T x} \tag{28}$$

Using Matlab, *a*, *b*, *c*, d, *e* and *w* where created to be random 100×1 single precision vectors with *w* used as the input vector for the initial derivative calculation using OpenAD. A vector *p* was then calculated as a projection of *w* onto the space spanned by *a*, *b*,

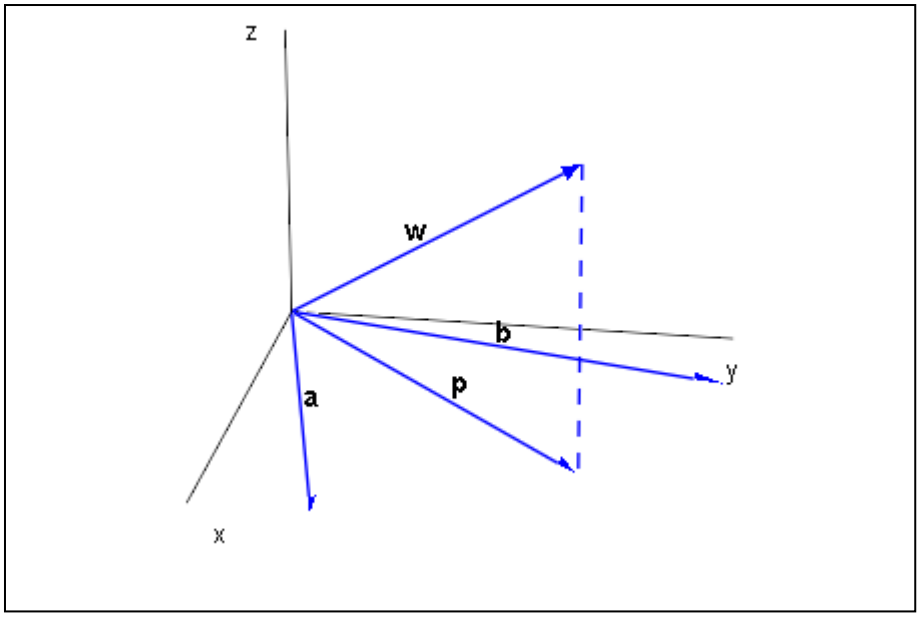

Figure 4.17: Vector projection onto a plane

c, d, and e and used as a second input vector. The resulting derivative values were on the orders of 10^1 and 10^2 with the maximum absolute difference between the two sets of derivatives being 1.901×10^{-6}. As expected, the two sets of derivatives are essentially the same.

One concern that can arise with this method is the fact that the rank is determined via a collection of random samples of the model in question. Some may have concerns that using random samples can risk not capturing the important directions in a model's output and can lead to highly inaccurate reduced output. This could prove to be an issue when applied to models that do not have sufficient benchmarking data to check the accuracy of reduced output.

A model with two directions will be considered. To be able to show what is happening visually in a geometric sense, only two inputs will be considered for now. The model that will be discussed is as follows:

$$y = e^{-(a^T x)} \cos(a^T x) + e^{-(b^T x)} \sin(b^T x) \tag{29}$$

where x, a, and b are 2×1 vectors. The vectors a and b will be defined so that their components are far enough apart to give the model two distinct directions. $a = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and $b = \begin{bmatrix} 8 \\ 6 \end{bmatrix}$ are shown graphically in the x_1, x_2 plane in Figure 4.18.

When the derivatives of the function are sampled, only inputs that are near the directions of a and b (approximated by the red circles in Figure 4.18) will yield useful information that captures these directions. Any samples not in this area will not contribute significantly to determining the rank of the problem. When these directions are not known beforehand, it must be insured that inputs used to sample the derivatives contain enough of the components of these directions in order to be useful.

This idea will be expanded and tested with a model consisting of 100 inputs and three orthogonal directions. The model will be similar to the one above.

$$y = e^{-(a^T x)} \cos(a^T x) + e^{-(b^T x)} \sin(b^T x) + e^{-(c^T x)} \tan(c^T x) \tag{30}$$

It is clear that when the rank finding algorithm is applied to this model, the effective rank should be three. First, the rank finding algorithm will be applied but instead of completely random inputs, random inputs that are near orthogonal to the problem space will be used. As expected, the resulting sets of derivatives are essentially equal. This type of result will not work well in the error criterion step ($\|(\mathbf{I} - \mathbf{Q}\mathbf{Q}^T)\mathbf{G}\|$) of the RFA. Table 4-3 gives results of the error criterion evaluated for effective rank estimations of $r = 1, \ldots, 5$.

It can be seen from the table that the orthogonal inputs do not correctly evaluate an effective rank of three. Now, the RFA will be run as it is intended, with completely random inputs and a cutoff error criterion of 10^{-8}. The results are given in Table 4-4.

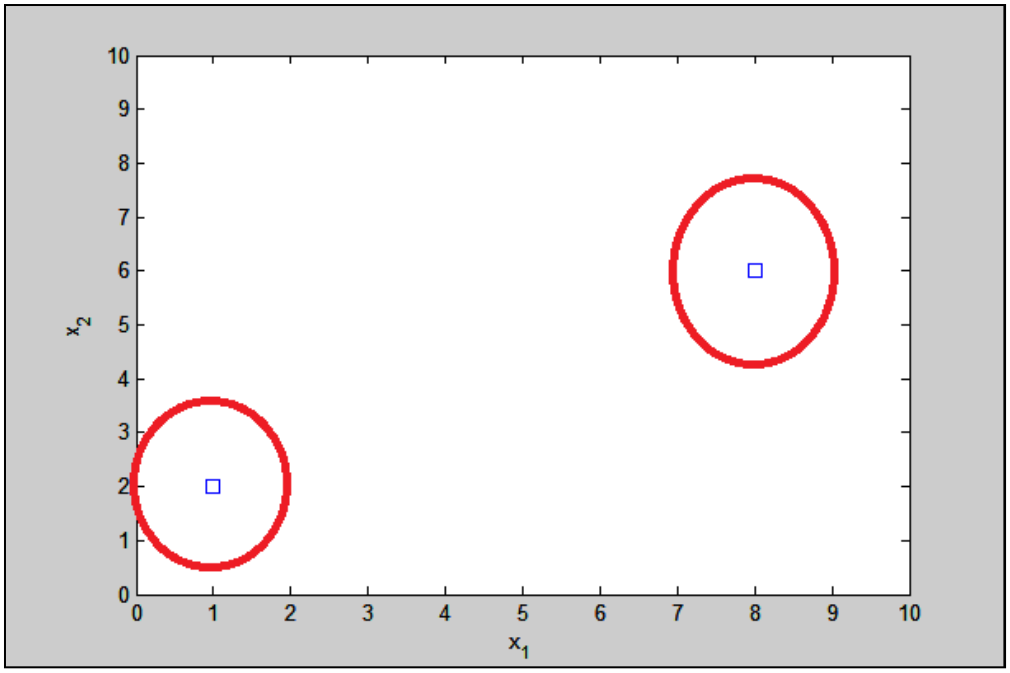

Figure 4.18: Graphical representation of the problem directions with the possible sampling areas indicated

Table 4-3: Error criterion evaluations with random orthogonal inputs

Effective rank estimation	Error criterion
1	5.807×10^{-16}
2	8.595×10^{-16}
3	1.158×10^{-15}
4	1.364×10^{-15}
5	1.324×10^{-15}

Table 4-4: Error criterion evaluations with random inputs

Effective rank estimation	Error criterion
1	7.406
2	0.4403
3	9.376×10^{-10}

The RFA now correctly selects an effective rank of three. The error criterion evaluations exhibit a clear drop between each estimation of the rank with a very large drop when the expected value of three is chosen. This example showcases that random inputs are necessary to obtain proper results in the RFA.

4.6 Third Order function Approximated with Second Order derivatives

By being able to efficiently calculate higher order derivatives, detailed surrogate models can be constructed that allow for faster calculations than using the full model. The surrogate models that will be discussed here involves using a Taylor expansion to approximate the model. The basic form that will be used is:

$$y = y_0 + \frac{dy}{dx}\Delta x + \frac{d^2y}{dx^2}\Delta x^2 + \cdots + \frac{d^oy}{dx^o}\Delta x^o \tag{31}$$

where o is the highest order of derivative used. For this example, first and second order derivatives will be used to construct a surrogate model for a third order function. It will be shown that sufficient accuracy can be achieved by not only using reduced derivative calculations, but by using derivatives up to an order less than the actual model order. The model that will be used in this example is:

$$y = (a^Tx)^3 + (b^Tx)(c^Tx)(d^Tx) + 10^4\frac{1}{(e^Tx)^3} \tag{32}$$

where all vectors are 100×1. Since this model uses vectors as inputs, the second order Taylor series approximation used for the surrogate model will be modified as follows:

$$y = y_0 + \mathbf{J}\Delta x + \Delta x^T \mathbf{H} \Delta x \tag{33}$$

where **J** is the 1×100 Jacobian and **H** is the 100×100 Hessian. This expression also highlights another benefit of trying to only use up to second order derivatives. Using higher order derivative tensors will lead to more complex expressions for surrogate models. If the model does require higher order derivatives, they can still certainly be used, but the second order approximation involves only simple matrix vector products.

When the RFA with a cutoff error criterion of 10^{-8} is applied to a code evaluating the model in Equation (32), the effective rank is found to be five, as expected. Reduced inputs were defined in a Rapsodia version of the code as was done in the previous examples. The resulting reduced derivatives were collected in Matlab and a surrogate model was constructed. Table 4-5 gives the values calculated by the actual model and the surrogate model for inputs perturbed from a vector consisting of all ones.

The table shows that depending on the desired relative error, the surrogate model evaluated around one point can be used to estimate the model values for a range of inputs around that point. Figure 4.19 shows the surrogate model plotted against the actual model

Table 4-5: Surrogate model error evaluation around one point

Input Perturbation	Actual Value	Surrogate Model Value	Relative Error
-5%	-2.145135×10^3	-2.077070×10^3	3.173%
-2%	-2.214260×10^3	-2.185020×10^3	1.320%
-1%	2.239183×10^3	2.224228×10^3	0.6679%
+1%	-2.2918567×10^3	-2.307481×10^3	0.6817%
+2%	-2.319609×10^3	-2.351526×10^3	1.376%
+5%	-2.408547×10^3	-2.493336×10^3	3.520%

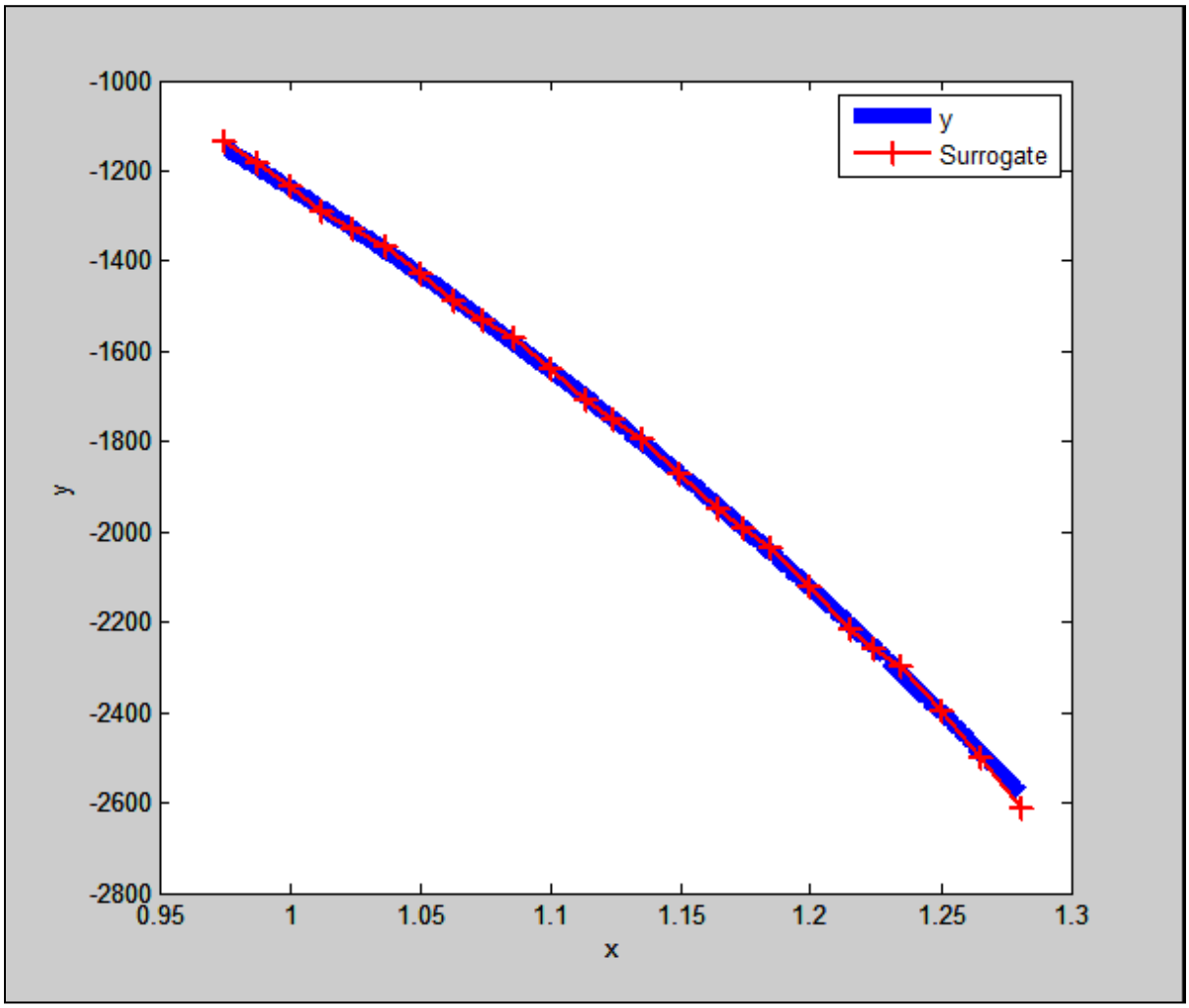

Figure 4.19: Second order surrogate model plotted with actual function

for inputvectors consisting of 100 values all equal to 0.9750 to 1.280. The surrogate model derivatives were evaluated at five points: $x = 1.000, 1.050, 1.100, 1.150, 1.200$, and 1.250. The surrogate model was then used to obtain estimated values of the model for $\pm 2.5\%$ and $\pm 1.25\%$ input perturbations around these points. In total, the surrogate model shown in the figure consists of 25 points.

4.7 MATWS Test Case

In the MATWS code package, single channel calculations were performed using the following inputs: axial expansion coefficient, Doppler coefficient, moderator temperature coefficient, control rod driveline, and core radial expansion coefficient. The outputs of interest are various temperatures within the channel: coolant temperature, structure temperature, cladding temperature, and fuel temperature. This gives 4×5 output for the first order derivatives and 15 and 35 directions for second and third order, respectively. The following figures (Figure 4.20 through Figure 4.24) give plots of the fuel temperature (TFUEL) for coarse variations ($\pm 50\%, \pm 100\%$) of each input variable separately in order to see the amount of non-linearity. First and second order polynomial fits are included with each figure along with the corresponding R^2 values. The degree to which each order polynomial fits can be used to estimate what order of dependence TFUEL has on each variable.

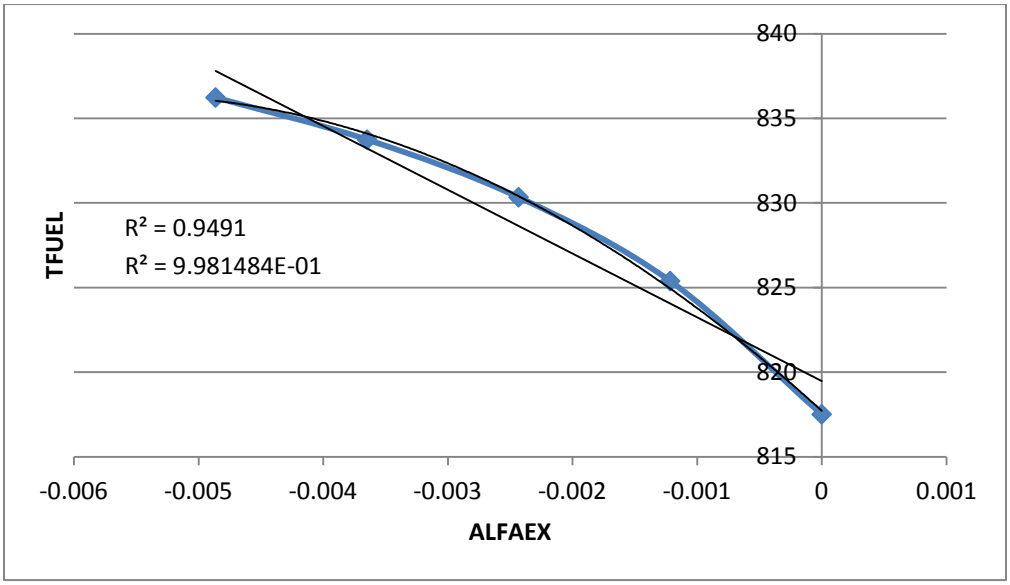

Figure 4.20: Fuel temperature for variations of the axial expansion reactivity coefficient

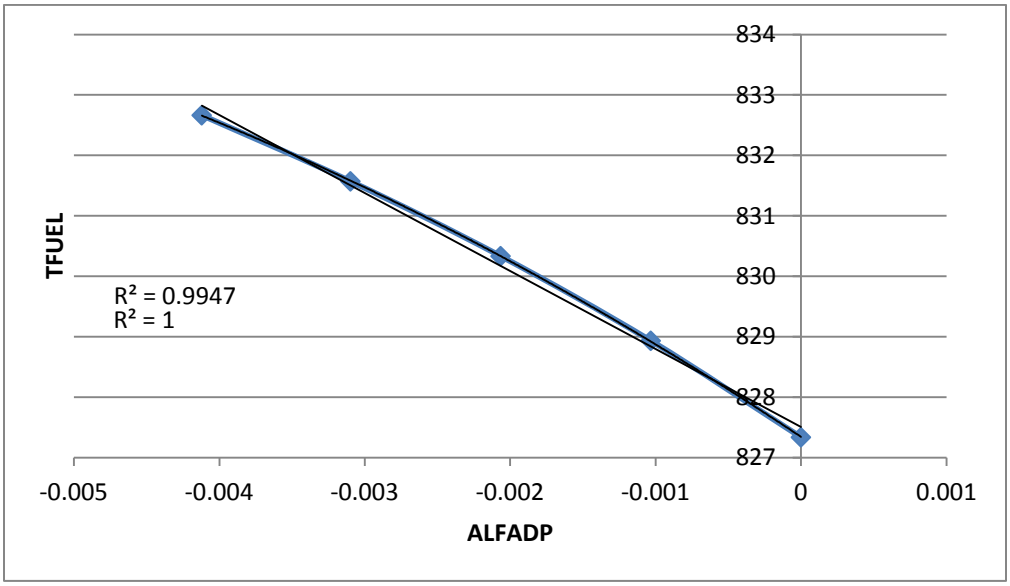

Figure 4.21: Fuel temperature for variations of the Doppler reactivity coefficient

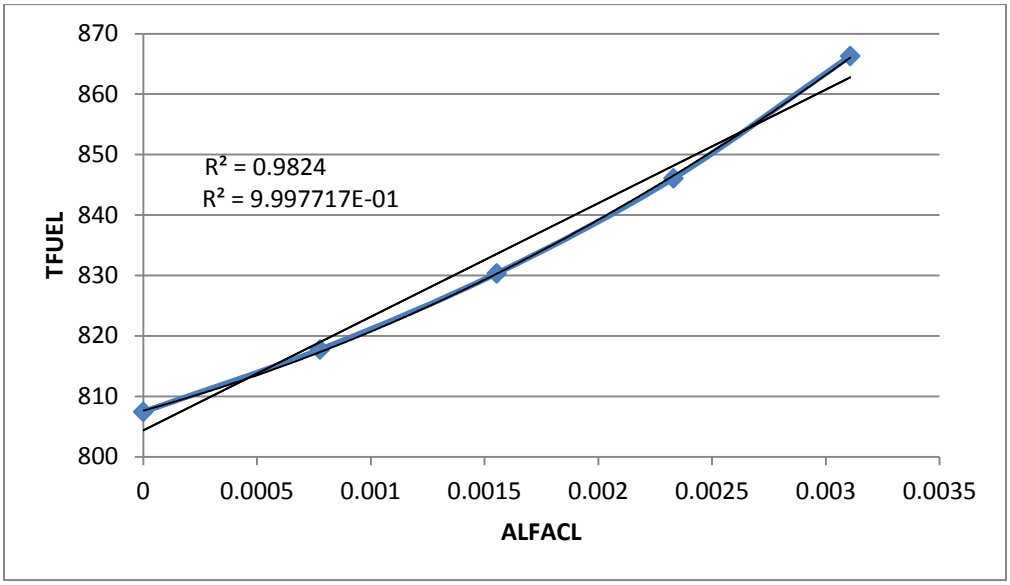

Figure 4.22: Fuel temperature for variations of the coolant reactivity coefficient

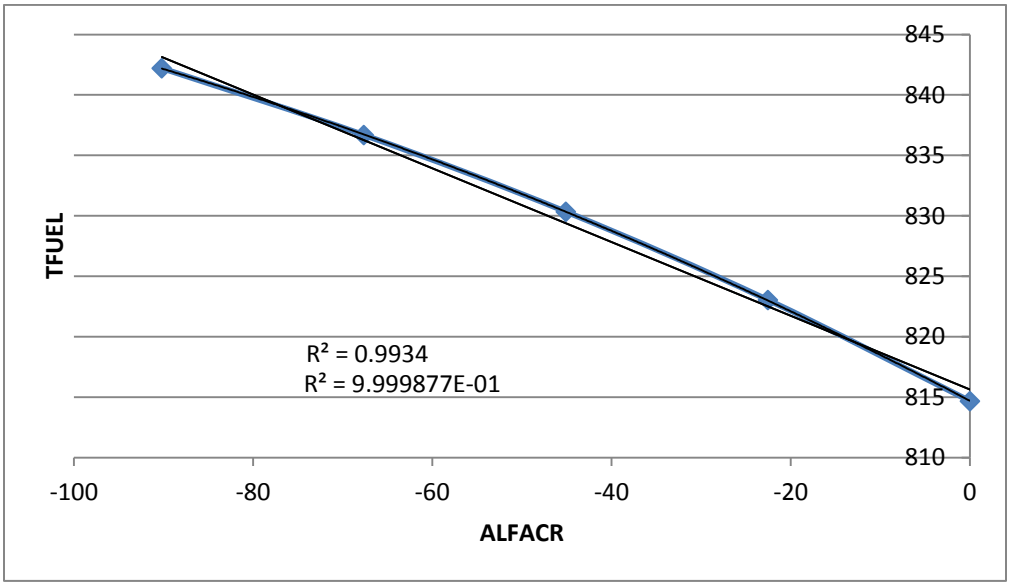

Figure 4.23: Fuel temperature for variations of the control rod expansion reactivity coefficient

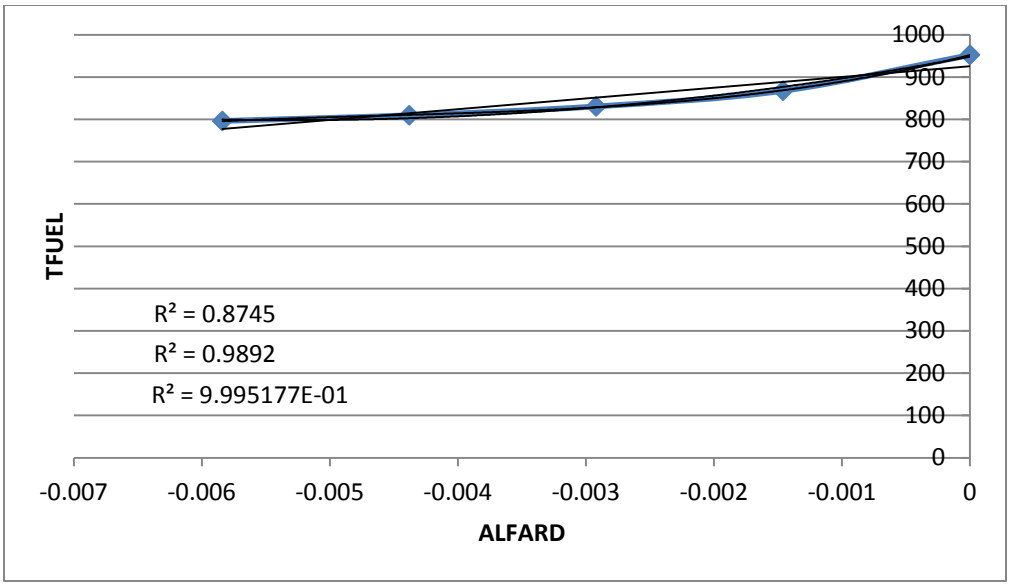

Figure 4.24: Fuel temperature for variations of the radial core expansion coefficient

After applying the subspace identification algorithm (Python script in Appendix B) and using OpenAD [11], it was found that the effective rank was $r = 3$, giving 6 and 10 directions for second and third order derivative calculations. The accuracy of the reduced derivative calculations will be evaluated by constructing a surrogate model for the output temperature vector T using the reference temperature vector T_0 and input perturbation vector $\Delta\alpha$:

$$T = T_0 + \mathbf{J}\Delta\alpha + \begin{bmatrix} \Delta\alpha^T \mathbf{H}_1 \Delta\alpha \\ \Delta\alpha^T \mathbf{H}_2 \Delta\alpha \\ \Delta\alpha^T \mathbf{H}_3 \Delta\alpha \\ \Delta\alpha^T \mathbf{H}_4 \Delta\alpha \end{bmatrix} \tag{34}$$

where $\mathbf{J}$ is the 4×5 Jacobian matrix of first order derivatives and $\mathbf{H}_i$ are the 5×5 Hessian matrices of second order derivatives that correspond to each output. Using an input perturbation of 0.01% the maximum relative errors were found using the unreduced and reduced derivatives. The 'Relative Error (AD)' column of Table 4-6 gives the maximum relative error between the temperatures found using the unreduced and reduced surrogate models. The 'Relative Error (Real)' column of Table 4-6 gives the maximum relative error between the temperatures found using the reduced surrogate model and the actual values that MATWS gives.

The output temperatures that MATWS calculates are $\sim 800^o F$, making these errors on the order of single degrees.

Table 4-6: MATWS test case results

Derivative Order	Unreduced Directions	Reduced Directions	Relative Error (AD)	Relative Error (Real)
1	5	3	9.216×10^{-5}	6.695×10^{-3}
2	15	6	1.252×10^{-4}	3.182×10^{-3}

CHAPTER 5 Conclusions and Future Work Recommendations

5.1 Conclusions

This thesis has presented a new way to more efficiently use automatic differentiation in order to calculate the derivatives of a computer code consisting of a computational model with many inputs and outputs. This method can theoretically be applied to any program that computes outputs based on multiple inputs, but the most effective use is when the computational model in question is rank deficient. This rank deficiency can be exploited by using the principles of ESM. A reduced set of pseudo variables can then be introduced into a version of the source code that will be transformed using the AD software. An approximate set of reduced derivatives can then be assembled using simple matrix-vector operations. These derivatives are then suitable for usage in a variety of different applications including sensitivity analysis, uncertainty quantification, and surrogate modeling.

5.2 Future Work Recommendations

Future work with this method will include the application to codes that are differentiable with OpenAD and Rapsodia and are able to utilize larger input and output streams. Potential codes include the components of the SCALE package from Oak Ridge National Lab, namely CENTRM and BONAMI. These codes process thousands of cross section values and would benefit from a more efficient differentiation method. Within

SCALE, the passage of derivative information from one differentiated code module to another would be of interest as well.

REFERENCES

[1] A. Griewank, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, SIAM, 1987.

[2] H. Bücker, B. Lang, A. Rasch and C. Bischof, "Computation of sensitivity information for aircraft design by automatic differentiation," in *International Conference on Computational Science*, Amsterdam, The Netherlands, 2002.

[3] M. Losch and P. Heimbach, "Adjoint Sensitivity of an Ocean General Circulation Model to Bottom Topography," *Journal Of Physical Oceanography,* vol. 37, pp. 377-393, 2006.

[4] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill and C. Wunsch, "OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes," *ACM Transactions on Mathematical Software,* 2008.

[5] I. Charpentier and J. Utke, *Rapsodia User Manual,* Argonne National Laboratory, 2011.

[6] S. J. Wilderman and G. S. Was, "Application of Adjoint Sensitivity Analysis to Nuclear Reactor Fuel Rod Performance," *Nuclear Engineering and Design,* vol. 80, pp. 27-38, 1984.

[7] H. S. Abdel-Khalik, P. D. Hovland, A. Lyons, T. E. Stover and J. Utke, "A Low Rank Approach to Automatic Differentiation," *Advances in Automatic Differentiation,* pp. 55-65, 2008.

[8] H. Abdel-Khalik, P. Turinsky and M. Jessee, "Efficient subspace methods-based algorithms for performing sensitivity, uncertainty, and adaptive simulation of large-scale computational models," 2007.

[9] E. Moris, "Uncertainty in unprotected loss-of-heat-sink, loss-of-flow, and transient-

overpower accidents," Argonne National Laboratory, 2007.

[10] J. Utke, U. Naumann and A. Lyons, "OpenAD/F: User Manual," Argonne National Laboratory, 2012.

[11] M. Alexe, O. Roderick, M. Anitescu, J. Utke, T. Fanning and P. Hovland, "Automatic differentiation of codes in nuclear engineering applications," ANL-MCS-310, 2010.

[12] H. Abdel-Khalik, "Adaptive Core Simulation," North Carolina State University, 2004.

[13] N. Halko, P. Martinsson and J. Tropp, "Finding Structure with Randomness Probabilistic Algorithms for Constructing Approximate Matrix Decompositions," in *SIAM*, 2011.

APPENDICES

Appendix A

OpenAD code for the Scalar-valued Model Example

```
subroutine head(x,ypseudo,n)
  integer :: i,n
  double precision :: x(n)
  double precision :: ypseudo,y1,y2,y3,y4
  double precision :: a(n),b(n),c(n),d(n),e(n)

  open(unit=20, file='Rinput.in.txt', status='old', action='read')

  do i=1,n
    read(20,*) a(i),b(i),c(i),d(i),e(i),x(i)
  end do

  close(20)

  y=0.0d0
  y1=0.0D0
  y2=0.0D0
  y3=0.0D0
  y4=0.0D0

!$openad INDEPENDENT(x)

  do i=1,n
       y1=y1+a(i)*x(i)
       y2=y2+b(i)*x(i)
       y3=y3+c(i)*x(i)
       y4=y4*d(i)*x(i)
  end do

  ypseudo=y1+y2**2+sin(y3)+1/(1+exp(y4))

!$openad DEPENDENT(ypseudo)

end subroutine
```

```
program driver
  use OAD_active
  use OAD_rev
  implicit none
  external head
  type(active),allocatable :: x(:)
  type(active) :: ypseudo
  integer :: i,n,j
  character*3 :: ypos
  character(len=5) :: BUFFER

  call getarg(1,BUFFER)
  read(BUFFER,'(i10)') n

  allocate(x(n))

  open(unit=11, file='der_out.txt', action='write')

 our_rev_mode%tape=.TRUE.

   ypseudo%d=1.0D0
   x%d=0.0D0
   call head(x,ypseudo,n)
   do j=1,n
      write(11,*) x(j)%d
   end do

 close(11)

end program driver
```

Rapsodia code for the Scalar-valued Model Example

```
PROGRAM DRIVER
  INCLUDE 'RAinclude.i90'
  USE higherOrderTensorUtil
  IMPLICIT NONE
  INTEGER :: O, DIRS, max_order
  TYPE(higherOrderTensor) :: T
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: SeedMatrix
  REAL(KIND=RAdKind), DIMENSION(:,:), &
                               ALLOCATABLE :: TaylorCoefficients
  REAL(KIND=RAdKind), DIMENSION(:),   &
                               ALLOCATABLE :: CompressedTensor

  TYPE(RARealD), allocatable :: x(:), pseudo(:)
  TYPE(RARealD) :: y
  integer :: i,n,j,k,p

  max_order=1

  n=100
  k=3

  allocate(x(n),pseudo(k))

  CALL setNumberOfIndependents(T, k)
  CALL setHighestDerivativeDegree(T, max_order)
  DIRS = getDirectionCount(T)
  ALLOCATE(SeedMatrix(k, DIRS))
  CALL getSeedMatrix(T, SeedMatrix)
  ALLOCATE(TaylorCoefficients(max_order, DIRS))
  ALLOCATE(CompressedTensor(DIRS))

  !output file
  open(unit=10, file='deriv_out.txt', action='write')


    WRITE (*,'(A,I3.1)') 'Number of directions = ', DIRS

    do j=1,k
    do i=1,DIRS
      CALL RAset(pseudo(j), i, 1, REAL(SeedMatrix(j, i), KIND=RAdKind))
    end do
    end do

    call head(x,y,n,k,pseudo)

    do i=1,max_order
      do j=1,DIRS
       CALL RAget(y, j, i, TaylorCoefficients(i, j))
      end do
    end do
```

(continued)

```
   CALL setTaylorCoefficients(T, TaylorCoefficients)
   CALL getCompressedTensor(T, max_order, CompressedTensor)

   DO i = DIRS, 1,-1
     DO j = 1, k
       WRITE (10,'(A,I1,A)',ADVANCE='NO') '[', SeedMatrix(j, i), ']'
     END DO
     WRITE (10,*) CompressedTensor(i)
   END DO


 close(10)
 DEALLOCATE(CompressedTensor)
 DEALLOCATE(TaylorCoefficients)
 DEALLOCATE(SeedMatrix)

END PROGRAM DRIVER

subroutine head(x,y,n,k,pseudo)
  INCLUDE 'RAinclude.i90'
  integer :: n,i,k
  TYPE(RARealD) :: x(n), pseudo(k),y1,y2,y3,y4,y
  double precision :: a(n),b(n),c(n),d(n),e(n),q(n,k)

  !read input
  open(unit=42, file='Rinput.in.txt', status='old', action='read')

  do i=1,n
    read(42,*) a(i),b(i),c(i),d(i),e(i),x(i)%v
  end do

  close(42)

  !read Q
  open(unit=55, file='Q.in.txt', status='old', action='read')

  do i=1,n
      read(55,*) (q(i,j),j=1,k)
  end do

  close(55)

  !pseudo variables

      do j=1,k
       do i=1,n
             pseudo(j)%v=pseudo(j)%v+q(i,j)*x(i)%v
       end do
       end do

       x=0.0d0
       do i=1,n
         do j=1,k
              x(i)=x(i)+q(i,j)*pseudo(j)
        end do
       end do
```

(continued)

```
  !model calculation

  do i=1,n
       y1=y1+a(i)*x(i)
       y2=y2+b(i)*x(i)
       y3=y3+c(i)*x(i)
       y4=y4*d(i)*x(i)
  end do

  y=y1+y2**2+sin(y3)+1/(1+exp(y4))

  !end model calculation

end subroutine


       x=0.0d0
       do i=1,n
         do j=1,k
              x(i)=x(i)+q(i,j)*pseudo(j)
        end do
       end do

  !model calculation

  do i=1,n
       y1=y1+a(i)*x(i)
       y2=y2+b(i)*x(i)
       y3=y3+c(i)*x(i)
       y4=y4*d(i)*x(i)
  end do

  y=y1+y2**2+sin(y3)+1/(1+exp(y4))

  !end model calculation

end subroutine
```

Appendix B

Python script used for the subspace identification algorithm for MATWS

```
#!/usr/local/apps/python-2.6.5/bin/python

from scipy import linalg
import numpy, getopt, sys
from numpy import dot, transpose, zeros, random
import os, commands

e=0.000001

#get value of n
n=int(sys.argv[1])
#get value of k
k=int(sys.argv[2])

In=numpy.genfromtxt('input.txt')
Y=numpy.zeros((n,k))

i=0
while (i<k):
   #run the forward driver
   status,output=commands.getstatusoutput('./reactor_OAD_MP <
Reference_ULOF.input > output')
   print status
   #input Y from output of forward driver
   Y[:,i]=numpy.genfromtxt('der_out.txt')
   j=0
   while(j<n):
      In[j]=In[j]+In[j]*0.2
      j=j+1

   numpy.savetxt('input.txt',In)
   i=i+1

numpy.savetxt('Y',Y)

#calculate SVD of Y
U,S,V=linalg.svd(Y)

#determine reduced Y
r=S.shape[0]
S[r-1]=0
Yk=dot(dot(U,linalg.diagsvd(S,len(Y),len(V))),V)
Yk0=Yk
i=2
while (linalg.norm(Y-Yk0)/linalg.norm(Y) < e):
   Yk=Yk0
   S[r-i]=0
   Yk0=dot(dot(U,linalg.diagsvd(S,len(Y),len(V))),V)
   i=i+1

#calculate QR of Yk
Q,R=linalg.qr(Yk)
r=numpy.sum(S > 1e-10)+1
Q1=Q[:,0:r]

#output Q to file for reverse input
numpy.savetxt('Q.in.txt',Q1)
```