A Small-Perturbation Automatic-Differentiation (SPAD) Method for Evaluating Uncertainty in Computational Electromagnetics

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By
Michael Gilbert, B.S., M.S.
Electrical and Computer Engineering Graduate Program

The Ohio State University
2012

Dissertation Committee
Professor Fernando Teixeira, Adviser
Professor Joel Johnson
Professor Robert Burkholder
Professor Fangyang Zheng
© Copyright by

Michael Gilbert

2012
ABSTRACT

In real-world problems, inputs to computational electromagnetics (CEM) simulations are often not known with full certainty (or precision): it is not possible to manufacture geometry exactly to the required specifications; results of material measurement systems have errors and uncertainty; and the frequency of operation and angles of observation will not be perfect in measurement systems. In spite of this, virtually all CEM algorithms presume a single deterministic solution (often from presumed mean values of the uncertain input parameters). The goal of this work is to devise a general approach for incorporating uncertainties directly into computational electromagnetics tools.

A new approach to solve these types of problems is proposed in this dissertation. This method involves a Small-Perturbation (SP) expansion augmented with an Automatic-Differentiation (AD) solver and is denoted as the Small-Perturbation Automatic-Differentiation or SPAD method.

It will be seen that the new method requires minimal change to the underlying code and may be more efficient (based on chosen limits) than existing techniques. The trade-off is that this new method is best applicable to “small” perturbations, whereas many other techniques are applicable to any sized perturbation. However, situations involving small deviations (such as tolerances in manufacturing and variability in measurement systems) comprise a vast majority of practical scenarios.
The new method also has specific advantages. The derivatives computed via automatic-differentiation can easily be used to solve the same problem with different input statistics (i.e., a different probability density function) or to compute other output quantities based on derivatives of the function (such as a frequency sweep with interpolation). The method is also applicable to a wide diversity of problems including measurement validation, systems involving rough surfaces, randomly varying media, and importantly mixed problems involving randomly varying aspects as well as deterministic components.
Dedicated to the memory of Alex.
ACKNOWLEDGMENTS

The completion of this dissertation is primarily due to the support and guidance from Professor Fernando Teixeira. I am incredibly grateful for his keen viewpoint, continual support, and openness to new ideas and new ways of looking at problems.

I am also greatly appreciative of Professor Joel Johnson’s expertise and guidance over almost ten years of my graduate education, and to Dr. Robert Burkholder for participating on my committee.

Thanks also to Tom Holcombe, Dr. Kueichien Hill, and the group at WPAFB for supporting my pursuit of education.

Thanks especially to my family for their continual love and support.

Finally, thanks to Megan, the love of my life, for being continually supportive as I spent countless hours away preparing this work.
VITA

April 10, 1979 ......................... Born - Canton, OH, USA

March 15, 2002 ....................... B.S.E.C.E,
                          The Ohio State University.

June 8, 2003 .......................... M.S.E.E,
                          The Ohio State University.

Present ............................... Ph.D. Candidate,
                          The Ohio State University.

FIELDS OF STUDY

Major Field: Electrical and Computer Engineering

Studies in:

Surface Scattering Theory ...................................... Prof. Joel Johnson
Uncertainty in Computational Electromagnetics .......... Prof. Fernando Teixeira
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Background and Fundamentals</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Basic Uncertainty Methodologies</td>
<td>3</td>
</tr>
<tr>
<td>1.3.1 Monte Carlo</td>
<td>3</td>
</tr>
<tr>
<td>1.3.2 Midpoint Integration</td>
<td>4</td>
</tr>
<tr>
<td>1.3.3 Adaptive Lagrange Interpolation</td>
<td>5</td>
</tr>
<tr>
<td>1.3.4 Sample Results via Basic Methods</td>
<td>9</td>
</tr>
<tr>
<td>1.4 Recent Developments in Uncertainty Analysis</td>
<td>22</td>
</tr>
<tr>
<td>1.5 Introducing the new SPAD Approach for Computing Uncertainties</td>
<td>30</td>
</tr>
<tr>
<td>1.6 Automatic-Differentiation</td>
<td>32</td>
</tr>
<tr>
<td>1.6.1 Forward-Mode Automatic-Differentiation</td>
<td>34</td>
</tr>
<tr>
<td>1.6.2 Reverse-Mode Automatic-Differentiation</td>
<td>39</td>
</tr>
<tr>
<td>2. Forward-mode SPAD</td>
<td>45</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>45</td>
</tr>
</tbody>
</table>
7. Conclusions ................................................. 125

Appendices:

A. Computational divided differencing ..................................... 129

B. Applying SPAD to Problems with “Fat Tail” Probability Distributions  132

C. Code Listing .................................................. 136

Bibliography ..................................................... 153
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The Gaussian distribution</td>
<td>10</td>
</tr>
<tr>
<td>1.2 Uncertainty of the RCS of a Dielectric (Approximately Teflon) Sphere via FEM ($ka = 1.0$, 5,000 Monte Carlo samples): 3 midpoint samples.</td>
<td>12</td>
</tr>
<tr>
<td>1.3 Uncertainty of the RCS of a Dielectric (Approximately Teflon) Sphere via FEM ($ka = 1.0$, 5,000 Monte Carlo samples): 7 midpoint samples.</td>
<td>13</td>
</tr>
<tr>
<td>1.4 Uncertainty of the RCS of a Dielectric (Approximately Teflon) Sphere via FEM ($ka = 1.0$, 5,000 Monte Carlo samples): 99 midpoint samples.</td>
<td>14</td>
</tr>
<tr>
<td>1.5 Uncertainty of the RCS of a Dielectric (Approximately Teflon) Sphere via FEM ($ka = 1.0$, 5,000 Monte Carlo samples): 1,001 midpoint samples.</td>
<td>15</td>
</tr>
<tr>
<td>1.6 Convergence of the RCS Uncertainty of a Dielectric (Approximately Teflon) Sphere via MIE Series ($ka = 1.0$, 100 million Monte Carlo samples).</td>
<td>17</td>
</tr>
<tr>
<td>1.7 Uncertainty of the RCS of a Dielectric Sphere via FEM ($ka = 1.5$, 5,000 Monte Carlo samples): 3 midpoint samples.</td>
<td>18</td>
</tr>
<tr>
<td>1.8 Uncertainty of the RCS of a Dielectric Sphere via FEM ($ka = 1.5$, 5,000 Monte Carlo samples): 7 midpoint samples.</td>
<td>19</td>
</tr>
<tr>
<td>1.9 Uncertainty of the RCS of a Dielectric Sphere via FEM ($ka = 1.5$, 5,000 Monte Carlo samples): 99 midpoint samples.</td>
<td>20</td>
</tr>
<tr>
<td>1.10 Uncertainty of the RCS of a Dielectric Sphere via FEM ($ka = 1.5$, 5,000 Monte Carlo samples): 1,001 midpoint samples.</td>
<td>21</td>
</tr>
</tbody>
</table>
1.11 Uncertainty of the RCS of a Dielectric Sphere via MIE Series at Resonance \((ka = 1.0, \langle Re(\epsilon_r) \rangle = 19.4, \Delta Re(\epsilon_r) = 2.0, 1,000,000 \text{ Monte Carlo samples})\): 3 midpoint samples ........................................ 23

1.12 Uncertainty of the RCS of a Dielectric Sphere via MIE Series at Resonance \((ka = 1.0, \langle Re(\epsilon_r) \rangle = 19.4, \Delta Re(\epsilon_r) = 2.0, 1,000,000 \text{ Monte Carlo samples})\): 7 midpoint samples ........................................ 24

1.13 Uncertainty of the RCS of a Dielectric Sphere via MIE Series at Resonance \((ka = 1.0, \langle Re(\epsilon_r) \rangle = 19.4, \Delta Re(\epsilon_r) = 2.0, 1,000,000 \text{ Monte Carlo samples})\): 99 midpoint samples ........................................ 25

1.14 Uncertainty of the RCS of a Dielectric Sphere via MIE Series at Resonance \((ka = 1.0, \langle Re(\epsilon_r) \rangle = 19.4, \Delta Re(\epsilon_r) = 2.0, 1,000,000 \text{ Monte Carlo samples})\): 1,001 midpoint samples ........................................ 26

1.15 Convergence of the RCS Uncertainty of a Dielectric Sphere Near Resonance via MIE Series (inputs: \(ka = 1.0, \langle Re(\epsilon_r) \rangle = 19.4, \Delta Re(\epsilon_r) = 2.0, 1,000,000 \text{ Monte Carlo samples})\). ........................................ 27

1.16 Convergence of the RCS Uncertainty of a Dielectric Sphere Near Resonance via MIE Series Using Adaptive Integration (inputs: \(ka = 1.0, \langle Re(\epsilon_r) \rangle = 2.1, \Delta Re(\epsilon_r) = 0.2\)). ........................................ 28

1.17 Convergence of the RCS Uncertainty of a Dielectric Sphere Near Resonance via MIE Series Using Adaptive Integration (inputs: \(ka = 1.0, \langle Re(\epsilon_r) \rangle = 19.4, \Delta Re(\epsilon_r) = 2.0\)). ........................................ 29

2.1 Electrostatic line charge configuration ........................................ 64

2.2 Induced line charge uncertainty: 5 segments. ........................................ 66

2.3 Induced line charge uncertainty: 16 segments. ........................................ 67

2.4 Infinite parallel wire configuration ........................................ 67

2.5 Infinite parallel wire scattering: 8 wires. ........................................ 70

2.6 Infinite parallel wire scattering: 12 wires. ........................................ 71

2.7 Infinite parallel wire scattering: 3 wires with correlation. ......................... 72
3.1 Induced line charge uncertainty: 5 segments . . . . . . . . . . . . . . 87
3.2 Induced line charge uncertainty: 8 segments . . . . . . . . . . . . . . 88
3.3 Scattering from 3 infinite wires with placement uncertainty . . . . . 90
4.1 Particle scattering DDA configuration . . . . . . . . . . . . . . . . . 93
4.2 DDA backscatter with uncertainty that violates SPAD conditions . . 96
4.3 DDA dipole moments with uncertainty that violates SPAD conditions 97
4.4 DDA backscatter reduced (valid SPAD) uncertainty . . . . . . . . . . 98
4.5 DDA dipole moments reduced (valid SPAD) uncertainty . . . . . . . 99
5.1 Example rough surface instance . . . . . . . . . . . . . . . . . . . . . 101
5.2 Coordinate system . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
5.3 Method-of-moments (infinite wire basis) discretized rough surface in-
stance (top: isometric view, bottom: profile view with line denoting or-
original surface and dots denoting discretized wires) . . . . . . . . . . 110
5.4 Gaussian spectrum rough surface scattering ($kh = 0.1\pi, kl = 0.4\pi,$
$\theta_i = 30^\circ$) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
5.5 Power-law spectrum rough surface scattering ($a_0 = 6.573e^{-4}, k_l = k/2,$
$k_u = 2k, \theta_i = 30^\circ$) . . . . . . . . . . . . . . . . . . . . . . 116
6.1 SPAD error for DDA problem: Gaussian distribution . . . . . . . . . 121
6.2 SPAD error for DDA problem: uniform distribution . . . . . . . . . 122
6.3 SPAD error for DDA problem: log-normal distribution . . . . . . . . 123
6.4 SPAD error for DDA problem: uniform distribution (1m expected value)124
B.1 SPAD difference between Gaussian and log-normal random variables . 134
CHAPTER 1

Introduction

1.1 Motivation

In real-world problems, inputs to computational electromagnetics (CEM) simulations are often not known with full certainty (or precision): it is not possible to manufacture geometry exactly to the required specifications; results of material measurement systems have errors and uncertainty; and the frequency of operation and angles of observation will not be perfect in measurement systems. In spite of this, virtually all CEM algorithms presume a single deterministic solution (often from presumed mean values of the uncertain input parameters). The goal of this work is to devise a general approach for incorporating uncertainties directly into computational electromagnetics tools.

A new approach to solve these types of problems is proposed in this dissertation. It will be seen that the method requires minimal change to the underlying code and may be more efficient (based on chosen limits) than existing techniques. The trade-off is that this new method is best applicable to "small" perturbations, whereas many other techniques are applicable to any sized perturbation. However, situations
involving small deviations (such as tolerances in manufacturing and variability in measurement systems) comprise a vast majority of practical scenarios.

1.2 Background and Fundamentals

A standardized set of quantities are often used to describe the nature of random (or uncertain) quantities. For any function $\phi(\theta)$, where $\theta$ is an input random variable with known probability distribution $\psi(\theta)$. These statistical measures are from lowest- to highest-order the expected value $\langle \phi(\theta) \rangle$, the standard deviation $\Delta(\phi(\theta))$, the normalized skew $\gamma_1(\phi(\theta))$, and the normalized kurtosis $\gamma_2(\phi(\theta))$. Mathematically, these quantities are defined as

$$\langle \phi(\theta) \rangle = \int_{-\infty}^{\infty} \psi^*(\theta) \phi(\theta) \psi(\theta) d\theta$$ (1.1)

$$\Delta(\phi(\theta))^2 = \left\langle (\phi(\theta) - \langle \phi(\theta) \rangle)^2 \right\rangle = \int_{-\infty}^{\infty} \psi^*(\theta)(\phi(\theta) - \langle \phi(\theta) \rangle)^2 \psi(\theta) d\theta$$ (1.2)

$$\gamma_1(\phi(\theta)) = \frac{\left\langle (\phi(\theta) - \langle \phi(\theta) \rangle)^3 \right\rangle}{\Delta(\phi(\theta))^3} = \frac{1}{\Delta(\phi(\theta))^3} \int_{-\infty}^{\infty} \psi^*(\theta)(\phi(\theta) - \langle \phi(\theta) \rangle)^3 \psi(\theta) d\theta$$ (1.3)

and

$$\gamma_2(\phi(\theta)) = \frac{\left\langle (\phi(\theta) - \langle \phi(\theta) \rangle)^4 \right\rangle}{\Delta(\phi(\theta))^4} - 3 = \frac{1}{\Delta(\phi(\theta))^4} \int_{-\infty}^{\infty} \psi^*(\theta)(\phi(\theta) - \langle \phi(\theta) \rangle)^4 \psi(\theta) d\theta - 3$$ (1.4)

This listing can be continued ad infinitum for the higher-order moments (order 5, 6, and so on), but will not be included in this analysis since these terms do not produce significantly more physical insight than the first four moments. Note that the $\theta$ and $\phi(\theta)$ notations are used for generality, where $\theta$ denotes an uncertain input quantity (for example, the relative dielectric constant of a material in a CEM problem) and $\phi(\theta)$ denotes the uncertain output quantity of interest (e.g. radar cross-section).
Oftentimes in electrical engineering literature, the probability distribution $\psi(\theta)$ is given by $p(\theta) = \langle \psi(\theta) | \psi(\theta) \rangle = |\psi(\theta)|^2$; however, in order to maximize generality, $\psi(\theta)$ will be used in order to support constructively and destructively interfering distributions. This property is a requirement for quantum mechanics and may be of interest for certain problems in electrical engineering.

### 1.3 Basic Uncertainty Methodologies

Before moving on to the discussion on the new work in this dissertation, some background discussion on some fundamental approaches for computing uncertainties will follow. All of the basic methods involve the use of integration techniques to compute uncertainty. This discussion will include the Monte Carlo, Midpoint Integration, and Lagrange Adaptive Integration methods.

#### 1.3.1 Monte Carlo

An obvious and straightforward approach that can be applied to statistical problems is the Monte Carlo technique. This method proceeds by randomly sampling a function numerous times with various deterministic input samples and averaging the result. The term Monte Carlo itself originates from a code word used to compartmentalize the then-secret process, which was invented at Los Alamos during the Manhattan Project [1]. The code word itself was derived from the name of a casino in Monaco, which relatedly hosted games involving probability and chance. Mathematically, the statistical integrations of interest (1.1) can be found via Monte Carlo as

$$\langle \phi(\theta) \rangle_{mc} = \langle \psi(\theta) | \phi(\theta) | \psi(\theta) \rangle = \frac{1}{N} \sum_{i=0}^{N} \phi(\theta_i) \quad (1.5)$$
\[ \Delta(\phi(\theta))_{mc}^2 = \frac{1}{N} \sum_{i=0}^{N} (\phi(\theta_i) - \langle \phi(\theta) \rangle_{mc})^2 \]  
\[ \gamma_1(\phi(\theta))_{mc} = \frac{1}{N \Delta(\phi(\theta))_{mc}^3} \sum_{i=0}^{N} (\phi(\theta_i) - \langle \phi(\theta) \rangle_{mc})^3 \]  

and
\[ \gamma_2(\phi(\theta))_{mc} = \frac{1}{N \Delta(\phi(\theta))_{mc}^4} \sum_{i=0}^{N} (\phi(\theta_i) - \langle \phi(\theta) \rangle_{mc})^4 - 3 \]

where \( \theta_i \) is one of \( N \) random samples of the uncertain input parameter \( \theta \) (described by probability distribution \( \psi(\theta) \)). The Monte Carlo approach is optimum in terms of intuitiveness and implementation simplicity (one only needs to generate an input probability distribution, calculate many deterministic solutions, and calculate the average of the results). However, its downfall is computational complexity (accuracy improves very slowly as \( O(1/\sqrt{N}) \), where \( N \) is the number of random samples).

### 1.3.2 Midpoint Integration

Another fairly straightforward approach is to evaluate the integrals of interest using common approximate integration techniques. For example, the integration limits in (1.1-1.4) can be truncated based on knowledge that the input probability distribution tails off at a far enough distance from the origin. Then, the midpoint-based integration technique can be used to obtain expressions for the statistics of the output parameter of interest, which are

\[ \langle \phi(\theta) \rangle_{mi} = \int_{\theta_-}^{\theta_+ + \Delta\theta_{max} + h/2} \psi^*(\theta) \phi(\theta) \psi(\theta) d\theta = h \sum_{i=0}^{P} \psi^*(\theta_i) \phi(\theta_i) \psi(\theta_i) \]  
\[ \Delta(\phi(\theta))_{mi}^2 = h \sum_{i=0}^{P} \psi^*(\theta_i) (\phi(\theta_i) - \langle \phi(\theta) \rangle_{mi})^2 \psi(\theta_i) \]  
\[ \gamma_1(\phi(\theta))_{mi} = \frac{h}{\Delta(\phi(\theta))_{mi}^3} \sum_{i=0}^{P} \psi^*(\theta_i) (\phi(\theta_i) - \langle \phi(\theta) \rangle_{mi})^3 \psi(\theta_i) \]
and
\[
\gamma_2(\phi(\theta))_{mi} = \frac{h}{\Delta(\phi(\theta))^4} \sum_{i=0}^{P} \psi^*(\theta_i)(\phi(\theta_i) - \langle \phi(\theta) \rangle_{mi})^4 \psi(\theta_i) - 3
\]  

(1.12)

Note that \( \Delta\theta_{max} \) is a tunable truncation limit. Appropriate choices for this parameter depend on the probability distribution, and will be described in the results section. Note also that the integrals were approximated using a set of \( P \) integration points, \( \theta_i = \langle \theta \rangle - \Delta\theta_{max} + ih \), with uniform spacing \( d\theta \approx h = 2\Delta\theta_{max}/P \). This method is anticipated to be useful and practical because the accuracy of Midpoint Integration improves rapidly as \( O(1/P^2) \) or equivalently \( O(h^2) \).

1.3.3 Adaptive Lagrange Interpolation

The midpoint method is not necessarily desirable from a code users’ point of view since it is not very automated. In order to simplify the code users’ experience, automated integration approaches are also a straightforward solution for determining computational uncertainties. One such method is Adaptive Lagrange Interpolation.

This method uses Lagrange polynomials as bases of the integration. The approach proceeds adaptively to find lowest-order moment (expected value) accurately (to within a requested tolerance) using a minimum number of samples with a goal of running the deterministic code a minimum number of times, and then use the same samples with new interpolations to compute the higher-order moments (standard deviation, skew, and kurtosis). The details of the algorithm follow.

The first step again is to truncate the integration limits. The adaptive aspect will only be applied to evaluation of the expected value, then the same output values will be used to determine the higher-order moments.
\[ \langle \phi(\theta) \rangle_{ai} = \int_{\theta_2}^{\theta_2} \psi^*(\theta)\phi(\theta)\psi(\theta)d\theta \] (1.13)

The adaptive algorithm takes three input samples \((\theta_{-2}, \theta_0, \text{ and } \theta_2)\) and three output samples \((\phi(\theta_{-2}), \phi(\theta_0), \text{ and } \phi(\theta_2))\) where the output samples are computed by executing the deterministic CEM code for the three input parameters. For the first step, the following points are chosen for the input samples: \(\theta_{-2} = \langle \theta \rangle - \Delta \theta_{max}, \theta_0 = \langle \theta \rangle, \text{ and } \theta_2 = \langle \theta \rangle + \Delta \theta_{max},\) where choosing \(\Delta \theta_{max}\) is discussed in the results section in this chapter.

The next step is to approximate the function between these three samples using Lagrange polynomials. This will be a second-order Lagrange polynomial denoted \(L^{(2)}(\theta)\) and is given by

\[ L^{(2)}(\theta) = \phi(\theta_{-2})l_{-2}(\theta) + \phi(\theta_0)l_0(\theta) + \phi(\theta_2)l_2(\theta) \] (1.14)

where

\[ l_j(\theta) = \prod_{i=-2,0,2,i\neq j} \frac{\theta - \theta_i}{\theta_j - \theta_i}, \quad j = -2, 0, 2 \] (1.15)

Next, the higher-order interpolating polynomial will be computed in order to determine whether the integration has converged. This is done by sampling at the midpoints between the three existing samples to obtain extra samples. These midpoints will be \(\theta_{-1} = (\theta_0 - \theta_{-2})/2\) and \(\theta_1 = (\theta_2 - \theta_0)/2.\) The deterministic code is run to get the two associated output samples \(\phi(\theta_{-1})\) and \(\phi(\theta_1)\) at those points. A fourth-order Lagrange polynomial is then be used to fit a curve through all five of these points and denoted by \(L^{(4)}\)

\[ L^{(4)}(\theta) = \phi(\theta_{-2})l_{-2}(\theta) + \phi(\theta_{-1})l_{-1}(\theta) + \phi(\theta_0)l_0(\theta) + \phi(\theta_1)l_1(\theta) + \phi(\theta_2)l_2(\theta) \] (1.16)
where
\[
l_j(\theta) = \prod_{i=-2,-1,0,1,2, i \neq j} \frac{\theta - \theta_i}{\theta_j - \theta_i}, \quad j = -2, -1, 0, 1, 2
\] (1.17)

The coefficients of these polynomials turn out to be the only quantities of interest, so once calculated, (1.14) and (1.16) can be rewritten as
\[
L^{(2)}(\theta) = a_2\theta^2 + a_1\theta + a_0
\] (1.18)
\[
L^{(4)}(\theta) = b_4\theta^4 + b_3\theta^3 + b_2\theta^2 + b_1\theta + b_0
\] (1.19)

where the \(a_i\)'s and \(b_i\)'s can be extracted. Next, approximate values of the integrals between the \(\theta_{-2}\) and \(\theta_2\) bounds are found using simple anti-derivatives as
\[
I^{(2)} = \int_{\theta_{-2}}^{\theta_2} L^{(2)}(\theta)d\theta = \frac{a_2}{3}(\theta_2^3 - \theta_{-2}^3) + \frac{a_1}{2}(\theta_2^2 - \theta_{-2}^2) + a_0(\theta_2 - \theta_{-2})
\] (1.20)
\[
I^{(4)} = \int_{\theta_{-2}}^{\theta_2} L^{(4)}(\theta)d\theta
\]
\[
= \frac{b_4}{5}(\theta_2^5 - \theta_{-2}^5) + \frac{b_3}{4}(\theta_2^4 - \theta_{-2}^4) + \frac{b_2}{3}(\theta_2^3 - \theta_{-2}^3) + \frac{b_1}{2}(\theta_2^2 - \theta_{-2}^2) + b_0(\theta_2 - \theta_{-2})
\] (1.21)

In order to determine whether these approximations have converged, \(I^{(2)}\) and \(I^{(4)}\) are compared to see if the difference is smaller than a tolerance as specified by the user. If the difference is less than the tolerance (\(|I^{(4)} - I^{(2)}| < \text{tolerance}\)), then the Lagrange polynomial is a reasonable approximation for the function and thus it is concluded that \(\langle \phi(\theta) \rangle = I^{(4)}\) is a converged approximation for the integral between \(\theta_{-2}\) and \(\theta_2\).

Otherwise, this is not a sufficient approximation, and more work needs to be done. In this case, the integral is split into two more parts and process is repeated.
\[
\langle \phi(\theta) \rangle_{ai} = I^{\text{low}} + I^{\text{high}}
\] (1.22)
where

\[ I_{\text{low}} = \int_{\theta_{-2}}^{\theta_0} \psi^*(\theta) \phi(\theta) \psi(\theta) d\theta \] (1.23)

\[ I_{\text{high}} = \int_{\theta_0}^{\theta_2} \psi^*(\theta) \phi(\theta) \psi(\theta) d\theta \] (1.24)

and for \( I_{\text{low}} \), the inputs are \( \theta_{-2} = \theta_2, \theta_0 = \theta_{-1}, \text{ and } \theta_2 = \theta_0 \). For \( I_{\text{high}} \), the inputs are \( \theta_{-2} = \theta_0, \theta_0 = \theta_1, \text{ and } \theta_2 = \theta_2 \). The same process is followed to determine two approximations for the integral for both \( I_{\text{low}} \) and \( I_{\text{high}} \) and compared to the tolerance. If the tolerance condition for \( I_{\text{low}} \) is met, then \( I_{\text{low}} = I_{\text{low}}^{(4)} \), and similarly for \( I_{\text{high}} \). If the tolerance condition is not met in either case, then the integral is further subdivided. This process is continued until the tolerance condition is met for all subdivisions.

The user-selectable tolerance parameter controls the absolute error (e.g. number of significant digits to expect in the output). For example, for a tolerance of \( 10^{-2} \), one would expect the adaptive solution to be accurate to two places after the decimal point. There is a chance that the error is worse than that requested if the difference between the approximations is close to the tolerance. For example if the difference is the tolerance in two subdivisions, then the total output error will be twice the tolerance. The code can present the addition of all differences as a pessimistic total error estimate, but the actual error will be less than this since, in the end, the better fourth-order polynomial is used.

Now that the expected value is obtained via the adaptive method, the higher-order moments can be computed using the same samples already found. Again, the goal was to minimize the number of times that the deterministic code needed to be run. From the above computations, samples in blocks of five (\( \theta_{-2}, \theta_{-1}, \theta_0, \theta_1, \text{ and } \theta_2 \) are used for each subdivision). Then, these samples are used to compute fourth-order
Lagrange polynomials for the higher-order moments are

\[ L_{\text{dev}}^{(4)}(\theta) = (\phi(\theta_2) - \langle \phi(\theta) \rangle_{\text{ai}})^2 l_{-2}(\theta) + (\phi(\theta_1) - \langle \phi(\theta) \rangle_{\text{ai}})^2 l_{-1}(\theta) + (\phi(\theta_0)) \]

\[ - (\phi(\theta))_{\text{ai}}^2 l_0(\theta) + (\phi(\theta_1) - \langle \phi(\theta) \rangle_{\text{ai}})^2 l_1(\theta) + (\phi(\theta_2) - \langle \phi(\theta) \rangle_{\text{ai}})^2 l_2(\theta) \]

(1.25)

\[ L_{\text{skw}}^{(4)}(\theta) = (\phi(\theta_2) - \langle \phi(\theta) \rangle_{\text{ai}})^3 l_{-2}(\theta) + (\phi(\theta_1) - \langle \phi(\theta) \rangle_{\text{ai}})^3 l_{-1}(\theta) + (\phi(\theta_0)) \]

\[ - (\phi(\theta))_{\text{ai}}^3 l_0(\theta) + (\phi(\theta_1) - \langle \phi(\theta) \rangle_{\text{ai}})^3 l_1(\theta) + (\phi(\theta_2) - \langle \phi(\theta) \rangle_{\text{ai}})^3 l_2(\theta) \]

(1.26)

\[ L_{\text{krt}}^{(4)}(\theta) = (\phi(\theta_2) - \langle \phi(\theta) \rangle_{\text{ai}})^4 l_{-2}(\theta) + (\phi(\theta_1) - \langle \phi(\theta) \rangle_{\text{ai}})^4 l_{-1}(\theta) + (\phi(\theta_0)) \]

\[ - (\phi(\theta))_{\text{ai}}^4 l_0(\theta) + (\phi(\theta_1) - \langle \phi(\theta) \rangle_{\text{ai}})^4 l_1(\theta) + (\phi(\theta_2) - \langle \phi(\theta) \rangle_{\text{ai}})^4 l_2(\theta) \]

(1.27)

The integrals \( I_{\text{dev}}^{(4)}, I_{\text{skw}}^{(4)}, \) and \( I_{\text{krt}}^{(4)} \) are computed as described above for the expected value using the Lagrange coefficients and anti-derivatives. Then the results of the integrals are summed for all subdivisions together to get the statistics of interest.

\[ \Delta(\phi(\theta))_{\text{ai}}^2 = \sum_{\text{all subdivisions}} I_{\text{dev}}^{(4)} \]

(1.28)

\[ \gamma_1(\phi(\theta))_{\text{ai}} = \frac{1}{\Delta(\phi(\theta))_{\text{ai}}^2} \sum_{\text{all subdivisions}} I_{\text{skw}}^{(4)} \]

(1.29)

\[ \gamma_2(\phi(\theta))_{\text{ai}} = \frac{1}{\Delta(\phi(\theta))_{\text{ai}}^4} \sum_{\text{all subdivisions}} I_{\text{krt}}^{(4)} \]

(1.30)

1.3.4 Sample Results via Basic Methods

There are many possible probability distributions that could be chosen to describe the probabilistic nature of the uncertain input quantity. As a starting point, the classic Gaussian probability distribution will be used (see Figure 1.1). This may or
Figure 1.1: The Gaussian distribution
may not describe real-world phenomena well, but a key advantage is that it is a simple
distribution and it is possible to analytically compute most of the mathematics, which
is not possible for most other distributions. The Gaussian probability density function
is given by

\[ \psi(\theta) = \frac{1}{(2\pi(\Delta\theta)^2)^{1/4}} \exp \left[ \frac{(\theta - \langle \theta \rangle)^2}{4(\Delta\theta)^2} \right] \]  

(1.31)

For the Monte Carlo approach, the distribution is used to choose the locations \( \theta_i \)'s)
of the input random samples.

For the Midpoint Integration approach, this distribution is simply plugged in for \( \psi(\theta_i) \) in the summation. One of the key concerns in this method is where to
truncate the integration. For the Gaussian distribution, choosing a truncation limit
of \( \Delta\theta_{\text{max}} = 5\Delta\theta \) is often reasonable since the probability density at that point is less
than 1/1,000. However, if high precision is a concern, a better choice is \( \theta_{\text{max}} = 10\Delta\theta \)
(a probability density of less than 1/1,000,000). Unless otherwise indicated, for the
results presented in this dissertation, a truncation of \( \Delta\theta_{\text{max}} = 5\Delta\theta \) was chosen.

Some sample results have been generated using the techniques described in the
previous sections. As a starting point, a dielectric sphere was chosen as a simple
problem where the real part of the permittivity is used as the uncertain input quantity
\( \theta = Re(\epsilon_r) \). Choosing \( ka = 1.0, \langle Re(\epsilon_r) \rangle = 2.1, \Delta Re(\epsilon_r) = 0.2 \), where \( \Delta \) indicates
standard deviation, and comparing the midpoint integration approach with 3, 7, 99,
and 1001 points to the Monte Carlo approach using \( N = 5,000 \) samples using a Finite
Element code for the deterministic solutions, there is good agreement between the
probability density function and statistics for this specific problem using 7 or more
integration points (see Figures 1.2-1.5).
Figure 1.2: Uncertainty of the RCS of a Dielectric (Approximately Teflon) Sphere via FEM \((ka = 1.0, 5,000 \text{ Monte Carlo samples})\): 3 midpoint samples.
Figure 1.3: Uncertainty of the RCS of a Dielectric (Approximately Teflon) Sphere via FEM ($ka = 1.0$, 5,000 Monte Carlo samples): 7 midpoint samples.
Figure 1.4: Uncertainty of the RCS of a Dielectric (Approximately Teflon) Sphere via FEM ($ka = 1.0$, 5,000 Monte Carlo samples): 99 midpoint samples.
Figure 1.5: Uncertainty of the RCS of a Dielectric (Approximately Teflon) Sphere via FEM ($ka = 1.0$, 5,000 Monte Carlo samples): 1,001 midpoint samples.
Looking at the convergence of the statistics for this case (Figure 1.6), it can be seen that the results do rapidly converge. However, for the larger integration truncation \((10\Delta \theta)\), it does take longer to converge; however, the accuracy is greatly improved, but this can only been seen in the input statistics since the accuracy of the output Monte Carlo prediction is only good to \(10^{-6}\) for the standard deviation and \(10^{-3}\) for the higher-order statistics.

This case was relatively straightforward since the output turned out to behave in the same manner as the input distribution; however, for general problems, this will often not be the case. The question then becomes whether this approach also works well for non-Gaussian outputs. In fact, it does work for determining the statistical quantities, but it cannot be used to reproduce the output probability density function if that function is not Gaussian. However, a weighting curve can be plotted that does give some insight into the nature of the output distribution. At \(ka = 1.5\), the RCS of the dielectric sphere has a null. Computing the uncertainty for this scenario via the midpoint integration approach with 3, 7, 99, and 1001 points to the Monte Carlo approach using \(N = 5,000\) samples, there is good agreement for the statistical quantities and a different, but illustrative curve for the probability density function (see Figure 1.7-1.10). The convergence for this case is almost the same as the \(ka = 1.0\) case, so it is not presented here. The conclusion to draw is that nulls do not particularly affect the quantification of uncertainties.

Another question is whether this approach will work for highly varying outputs such as resonant scenarios. It was found that the dielectric sphere has a resonance near \(Re(\epsilon_r) = 19.4\), so this case was chosen as a representative resonant scenario. The uncertainty for this case was computed for various numbers of midpoint integration
Figure 1.6: Convergence of the RCS Uncertainty of a Dielectric (Approximately Teflon) Sphere via MIE Series ($ka = 1.0$, 100 million Monte Carlo samples).
Figure 1.7: Uncertainty of the RCS of a Dielectric Sphere via FEM ($ka = 1.5$, 5,000 Monte Carlo samples): 3 midpoint samples.
Figure 1.8: Uncertainty of the RCS of a Dielectric Sphere via FEM ($ka = 1.5$, 5,000 Monte Carlo samples): 7 midpoint samples.
Figure 1.9: Uncertainty of the RCS of a Dielectric Sphere via FEM ($ka = 1.5$, 5,000 Monte Carlo samples): 99 midpoint samples.
Figure 1.10: Uncertainty of the RCS of a Dielectric Sphere via FEM ($ka = 1.5$, 5,000 Monte Carlo samples): 1,001 midpoint samples.
sampling points. The results can be seen in Figures 1.11-1.14. The convergence was also computed for an increasing number of midpoint integration points and compared the results to a Monte Carlo solution with \( N = 1,000,000 \) (see Figures 1.15). The MIE series was used as the deterministic code for this case. As can be seen, the statistical quantities do converge to the Monte Carlo solution; although, for this resonant case, many more samples (about 200) are required to obtain an accurate solution. It may be possible to vastly reduce the number of required samples with an adaptive integration method, and this type of approach will be worked on in the near future.

The final results are for the adaptive integration method. Two scenarios were evaluated – the straightforward case for \( \langle Re(\epsilon_r) \rangle = 2.1, \Delta(Re(\epsilon_r)) = 0.2 \) that has a Gaussian-like output distribution and the complicated resonant case. These results are presented in Figures 1.16 and 1.17. As can be seen in both cases, given enough samples, the adaptive approach converges to the Monte Carlo prediction. However, this requires many more samples for the resonant case as can be seen in the bottom-left plot.

### 1.4 Recent Developments in Uncertainty Analysis

There have been many recent advancements in the area of computational uncertainty. One approach involves integrating “stochastic methods” (another term used to illustrate the probabilistic nature of uncertainty) into the finite-element method ([2], [3], and [4]). These techniques do appear to work reasonably well, but require extensive modifications to the underlying algorithms and are computationally expensive. Another uncertainty approach involves use of “stochastic collocation”. These
Figure 1.11: Uncertainty of the RCS of a Dielectric Sphere via MIE Series at Resonance ($ka = 1.0, \langle Re(\epsilon_r) \rangle = 19.4, \Delta(Re(\epsilon_r)) = 2.0, Im(\epsilon_r) = 0.0, k_0 = 1.0$)

Monte Carlo $\sigma/\pi a^2 = 5.131, \Delta(\sigma/\pi a^2) = 6.188, \gamma_1 = 2.116$

Integration, 3 pts $\sigma/\pi a^2 = 60.337, \Delta(\sigma/\pi a^2) = 42.496, \gamma_1 = -0.708$
Figure 1.12: Uncertainty of the RCS of a Dielectric Sphere via MIE Series at Resonance ($ka = 1.0$, $\langle Re(\epsilon_r) \rangle = 19.4$, $\Delta Re(\epsilon_r) = 2.0$, $Im(\epsilon_r) = 0.0$, $ka = 1.0$)

Monte Carlo $< \sigma/\pi a^2 > = 5.131, \Delta (\sigma/\pi a^2) = 6.188, \gamma_1 = 2.116$
Integration, 7 pts $< \sigma/\pi a^2 > = 21.718, \Delta (\sigma/\pi a^2) = 12.317, \gamma_1 = -0.836$

Inputs: $< Re(\epsilon_r) > = 19.4$, $\Delta (Re(\epsilon_r)) = 2.0$, $Im(\epsilon_r) = 0.0$, $ka = 1.0$
Figure 1.13: Uncertainty of the RCS of a Dielectric Sphere via MIE Series at Resonance \((ka = 1.0, \langle Re(\epsilon_r) \rangle = 19.4, \Delta Re(\epsilon_r) = 2.0, \text{Im}(\epsilon_r) = 0.0, ka = 1.0)\): 99 midpoint samples.
Figure 1.14: Uncertainty of the RCS of a Dielectric Sphere via MIE Series at Resonance ($ka = 1.0, \langle Re(\epsilon_r) \rangle = 19.4, \Delta(Re(\epsilon_r)) = 2.0, Im(\epsilon_r) = 0.0, \gamma_1 = 2.116$) using Monte Carlo sampling ($\sigma/\pi a^2 = 5.131, \Delta(\sigma/\pi a^2) = 6.188, 1,000,000$ samples) and Integration ($\sigma/\pi a^2 = 5.159, \Delta(\sigma/\pi a^2) = 6.197, 1,000$ midpoint samples).
Figure 1.15: Convergence of the RCS Uncertainty of a Dielectric Sphere Near Resonance via MIE Series (inputs: $ka = 1.0$, $\langle Re(\epsilon_r) \rangle = 19.4$, $\Delta Re(\epsilon_r) = 2.0$, 1,000,000 Monte Carlo samples).
Figure 1.16: Convergence of the RCS Uncertainty of a Dielectric Sphere Near Resonance via MIE Series Using Adaptive Integration (inputs: $ka = 1.0$, $\langle Re(\epsilon_r) \rangle = 2.1$, $\Delta Re(\epsilon_r) = 0.2$).
Figure 1.17: Convergence of the RCS Uncertainty of a Dielectric Sphere Near Resonance via MIE Series Using Adaptive Integration (inputs: $ka = 1.0$, $\langle Re(\epsilon_r) \rangle = 19.4$, $\Delta Re(\epsilon_r) = 2.0$).
methods employ a specific set of (deterministically evaluated) output points from a computational method to compute an integration (often using Stroud cubature [5]) that provides an estimate of statistical properties of interest. Some references dealing with this approach include [6], which describes higher-order collocation methods for stochastic differential equations in general, and [7] and [8], which apply collocation methods in conjunction with CEM solvers. There is also an ISO guide for uncertainties [9]. Finally, reference [10] provides a general overview of fast methods in the broader field of uncertainty quantification.

The approach proposed in this dissertation attacks this problem from a different angle—using a Taylor series expansion augmented by an automatic-differentiation (AD) solver. Note that the Taylor expansion is a classic solution to many problems, including statistical ones. It has, in fact, already been utilized as a method for solving probabilistic finite-element problems [11],[12]. These methods utilize the Taylor series to directly expand the resulting mass, dampening, and stiffness matrices obtained from the finite-element discretization. The distinctive aspect of the approach described here is that (in essence) every operational step in the computational process is expanded rather than only certain higher-level quantities obtained via a particular solution technique; as such, the procedure can be more fully automated.

1.5 Introducing the new SPAD Approach for Computing Uncertainties

After spending some discussion on the fundamental aspects of uncertainty, this section introduces a new approach for computing uncertainties. This method will be denoted as the “small-perturbation automatic-differentiation” (SPAD) approach. The method begins by expressing a particular function \( \Phi(x) \) in terms of a Taylor series,
where $\Phi$ is an output of interest, which is obtained for purposes in this dissertation from a CEM algorithm, and $\mathbf{x} = \{x_0, ..., x_d\}$ is the set of uncertain inputs to that algorithm. Using this expansion, the expected value of $\Phi(\mathbf{x})$ (or any other statistical value of interest) can be found via knowledge of the statistical properties of $\mathbf{x}$ and the various higher-order derivatives of $\Phi(\mathbf{x})$ via the multivariate small-perturbation (SP) expansion given by

$$
\langle \Phi(\mathbf{x}) \rangle \approx \sum_{n_0=0}^{N_0} \cdots \sum_{n_d=0}^{N_d} \left[ \prod_{k=0}^{d} \left( x_k - \langle x_k \rangle \right)^{n_k} \frac{d^n \Phi}{d x_k^{n_k}} \right] \Phi(\mathbf{x}) \bigg|_{\mathbf{x} = \langle \mathbf{x} \rangle}
$$

$$
\approx \sum_{n_0=0}^{N_0} \cdots \sum_{n_d=0}^{N_d} A_n B_n \left[ \prod_{k=0}^{d} \frac{d^n \Phi}{d x_k^{n_k}} \right] \Phi(\mathbf{x}) \bigg|_{\mathbf{x} = \langle \mathbf{x} \rangle}
$$

(1.32)

where $N_0, ..., N_d$ are the number of terms used in the expansion for each input variable (usually chosen to be the same number for simplicity, $N = N_0 = ... = N_d$), $B_n$ is a simple combination of factorials and is given by

$$
B_n = \frac{1}{n_k!}
$$

(1.33)

and

$$
A_n = \left\langle \prod_{k=0}^{d} (x_k - \langle x_k \rangle)^{n_k} \right\rangle
$$

(1.34)

are the higher-order multivariate central moments of $\mathbf{x}$. The final piece of the puzzle is determining the derivatives of the output function of interest

$$
\left[ \prod_{k=0}^{d} \frac{d^n \Phi}{d x_k^{n_k}} \right] \Phi(\mathbf{x}) \bigg|_{\mathbf{x} = \langle \mathbf{x} \rangle}
$$

(1.35)

which can be found via various means (such as numerically), but for the approach discussed in this dissertation, it will be determined via an automatic-differentiation (AD) methodology; the background of which will be briefly introduced in the next section. All of the above combined comprise the SPAD methodology.
1.6 Automatic-Differentiation

As mentioned previously, the SPAD approach takes advantage of a field of study known as Automatic-Differentiation (AD); sometimes referred to as Algorithmic Differentiation. According to [13], this technique was originally envisioned as far back as the time of Newton and Leibniz. Their intended goal was to obtain numerical values for derivatives in differential equations. The significance of applying such a method to general computational problems wasn’t “rediscovered” until relatively recent times (during the 1950s) as a method for designing circuits [14]. Much later, with the advent of high-level object-oriented computer languages (such as PASCAL, C++, and FORTRAN), it finally became practical to apply the method to any/all computational problems. The first implementation was done in PASCAL [15],[16]; a procedural language that could be augmented with object-oriented features. Subsequently, various Automatic-Differentiation libraries were developed for C++ [17], FORTRAN [18], and other languages. Object-orientation is an important requirement in terms of AD simplicity since this feature enables basic operations (such as addition, multiplication, powers, etc.) to be overridden by alternative methods. For AD, whenever a method is called, the derivatives can thus be calculated and updated in the background without any forethought by the user. Eventually AD capabilities also became available for standard C [19]; however, this was in a cumbersome non-object-oriented form via a set of various basic arithmetic method calls that would need to be cobbled together.

For a solid foundation on the subject of Automatic-Differentiation, the papers by A. Griewank [20, 21] and the first chapter of [22] are an excellent source of information. An interesting quote in [23] is, “[i]f care is taken in handling quantities which are
common to the function and derivatives, the ratio is usually around 1.5, not \( n + 1 \).” Ratio in the previous quote refers to the computational complexity \( O(1.5) \) vs. \( O(n) \). This result sounds quite interesting since it indicates that the computational complexity of automatic-differentiation scales as a constant rather than as a function of the number of random variables. That order of complexity is not achieved in the methods discussed in this dissertation, and it would be an interesting area of research to attempt to reproduce those results.

Chapters 7, 8, and 9 of [13] have a significant discussion of sparsity as applied to the AD methodology. This reference also has various other useful information on error analysis (chapter 3) and complexity bounds (chapter 4). Chapter 13 contains a discussion of the AD method as applied in applications involving the Taylor series, which is specifically relevant to the SPAD topic under discussion here.

In terms of Automatic-Differentiation applications, they are diverse and widespread over the spectrum of humanity’s scientific endeavors. A survey of applications includes neural networks, solutions of ordinary differential equations, simulations of the Tevatron particle accelerator, circuit device modeling, race car optimization, structural mechanics, computational fluid dynamics, and many others as can be found in references [22, 24, 25]. Interestingly Chapter 4 of [22] involves an application of the AD approach for the purpose of propagating uncertainties through a calculation. This is ultimately very similar to the SPAD methodology, but written somewhat differently. The method states that the expected value of an output parameter of interest \( y \) given
\( \mathbf{x} = \{x_0, ..., x_N\} \) as a set of \( N \) random inputs is

\[
\langle y(\mathbf{x}) \rangle = y(\langle \mathbf{x} \rangle) + \sum_{j=0}^{N} \left[ \frac{d^2 y(\mathbf{z})}{d z_j^2} \bigg|_{z=0} + \langle z_j^3 \rangle \frac{d^3 y(\mathbf{z})}{d z_j^3} \bigg|_{z=0} + \langle z_j^4 \rangle \frac{d^4 y(\mathbf{z})}{d z_j^4} \bigg|_{z=0} \right] \\
+ \sum_{j=0}^{N} \sum_{k=j}^{N} \langle z_j^2 z_k^2 \rangle \frac{d^4 y(\mathbf{z})}{d z_j^2 d z_k^2} \bigg|_{z=0} + ... \quad (1.36)
\]

where \( \mathbf{z} = \{z_0, ..., z_N\} \) are the same random input variables but normalized (i.e. \( \langle z_j \rangle = 0 \) and \( \Delta z_j = 1 \) with \( x_j = \langle x_j \rangle + \Delta x_j z_j \)) and the derivatives of \( y \) are determined by applying AD throughout a computation of interest. Note that there can be correlation between these variables by specifying an appropriate value for \( \langle z_j z_k \rangle \). Interestingly, the authors of this reference appear to have chosen a fourth-order limit for the Taylor series expansion, which is the same order chosen for practical reasons in the SPAD methodology. As can be seen, the formulation of the problem is essentially the same as SPAD except that the input random variables in this approach are normalized first and also excludes various terms (such as \( \frac{d^2 y}{d z_j d z_k} \), \( \frac{d^3 y}{d z_j^2 d z_k} \), and \( \frac{d^4 y}{d z_j^3 d z_k} \)) are excluded. These terms are included in the in the fourth-order SPAD approach in order to fully support correlation. This reference is also missing a factorial in the denominator of the Taylor series terms, but that may just be a typographical error.

The AD focus area is relatively mature at this point. Recent advances primarily center around improvements to computational complexity via application of hierarchical and multilevel methods; as described for example in [26, 27], but the foundations of the technology have been firmly established for almost 30 years. There is also a related methodology called computational divided differencing, which was not studied in this effort, but a brief introduction is included in Appendix A.
1.6.1 Forward-Mode Automatic-Differentiation

The best way to explain Automatic-Differentiation is via example (as is done in all of the foundational references on this topic). This isn’t to say that the method can only be applied to analytical problems as it is quite general and applicable to any computation involving elementary operations within its calculation steps, which in actuality is every computation. The analytical problem however is very useful for educational purposes as every step can be worked through and understood. A simple illustrative example is given in [13]. Suppose the goal of a computation is to calculate the following function

\[
\phi(x_1, x_2) = \left[ \sin \left( \frac{x_1}{x_2} \right) + \frac{x_1}{x_2} - e^{x_2} \right] \left[ \frac{x_1}{x_2} - e^{x_2} \right]
\]

(1.37)

where the goal is to determine the derivatives of \( \phi(x_1, x_2) \) with respect to those input parameters \( x_1 \) and \( x_2 \). Each step of the computation can be stored in a vector (denoted as \( \mathbf{v} \)). For the example function (1.37), the \( \mathbf{v} \) vector (evaluated at inputs \( x_1 = 3.0 \) and \( x_2 = 0.5 \) for example) is

\[
\begin{bmatrix}
  v_0 \\
v_1 \\
v_2 \\
v_3 \\
v_4 \\
v_5 \\
v_6 \\
v_7
\end{bmatrix} = \begin{bmatrix}
x_1 \\
x_2 \\
v_0/v_1 \\
\sin(v_2) \\
\exp(v_1) \\
v_2 - v_4 \\
v_3 + v_5 \\
v_0v_5
\end{bmatrix} = \begin{bmatrix}
  1.5 \\
  0.5 \\
  3.0 \\
  0.1411 \\
  1.6487 \\
  1.3513 \\
  1.4924 \\
  2.0167
\end{bmatrix}
\]

(1.38)

producing a value of \( \phi(x_1, x_2) = v_7 = 2.0167 \) of that function evaluated at the inputs of interest.

That is kind of a roundabout way of producing a calculation, but suppose now that the derivative of the output function (e.g. \( \frac{d\phi(x_1, x_2)}{dx_1} \)) is instead the quantity of interest. In order to obtain this parameter, a similar approach can be applied, but
involving a vector of derivatives instead. For this example, this derivative vector is given by

\[ \frac{d\mathbf{v}}{dx_1} = \begin{bmatrix}
  \frac{dv_0}{dx_1} | x_1=1.5, x_2=0.5 \\
  \frac{dv_1}{dx_1} | x_1=1.5, x_2=0.5 \\
  \frac{dv_2}{dx_1} | x_1=1.5, x_2=0.5 \\
  \frac{dv_3}{dx_1} | x_1=1.5, x_2=0.5 \\
  \frac{dv_4}{dx_1} | x_1=1.5, x_2=0.5 \\
  \frac{dv_5}{dx_1} | x_1=1.5, x_2=0.5 \\
  \frac{dv_6}{dx_1} | x_1=1.5, x_2=0.5 \\
  \frac{dv_7}{dx_1} | x_1=1.5, x_2=0.5 \\
\end{bmatrix} \]

\[ = \begin{bmatrix}
  \frac{dx_1}{dx_1} | x_1=1.5, x_2=0.5 \\
  \frac{dx_2}{dx_1} | x_1=1.5, x_2=0.5 \\
  (\frac{dv_1}{dx_1} | x_1=1.5, x_2=0.5 - v_2 \frac{dv_1}{dx_1} | x_1=1.5, x_2=0.5) / v_1 \\
  \frac{dv_2}{dx_1} | x_1=1.5, x_2=0.5 \cos(v_2) \\
  \frac{dv_3}{dx_1} | x_1=1.5, x_2=0.5 \exp(v_1) \\
  \frac{dv_4}{dx_1} | x_1=1.5, x_2=0.5 - \frac{dv_4}{dx_1} | x_1=1.5, x_2=0.5 \\
  \frac{dv_5}{dx_1} | x_1=1.5, x_2=0.5 + \frac{dv_5}{dx_1} | x_1=1.5, x_2=0.5 \\
  \frac{dv_6}{dx_1} | x_1=1.5, x_2=0.5 v_5 + v_6 \frac{dv_6}{dx_1} | x_1=1.5, x_2=0.5 \\
\end{bmatrix} = \begin{bmatrix}
  1.5 \\
  0.0 \\
  2.0 \\
  -1.98 \\
  0.0 \\
  2.0 \\
  0.02 \\
  3.0118 \\
\end{bmatrix} \]

and thus the desired quantity of interest is \( \frac{d\phi(x_1,x_2)}{dx_1} = \frac{dv_7}{dx_1} = 3.0118 \). Note that a similar methodology can be applied to obtain various higher-order and multivariate derivatives such as \( \frac{d^2\phi(x_1,x_2)}{dx_1^2} \). The fundamental feature being utilized is that new derivative values at each step in the computation can be obtained directly from the derivatives obtained from previous steps (all using the same AD methodology). As mentioned previously, the elementary operations (addition, multiplication, subtraction, etc.) involved above can be implemented in object-oriented computer languages via operator overloading. Those overloaded operators are a kind of backbone that calculate values and derivatives appropriately.
In terms of an implementation, the above example would involve the following steps and associated object-oriented language operator override calls.

<table>
<thead>
<tr>
<th>operation</th>
<th>override function call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0 = x_1$</td>
<td><strong>init</strong>(x_1)</td>
</tr>
<tr>
<td>$v_1 = x_2$</td>
<td><strong>init</strong>(x_2)</td>
</tr>
<tr>
<td>$v_2 = v_0/v_1$</td>
<td><strong>mul</strong>(v_0, <strong>pow</strong>(v_1, -1))</td>
</tr>
<tr>
<td>$v_3 = \sin(v_2)$</td>
<td><strong>sin</strong>(v_2)</td>
</tr>
<tr>
<td>$v_4 = \exp(v_1)$</td>
<td><strong>exp</strong>(v_1)</td>
</tr>
<tr>
<td>$v_5 = v_2 - v_4$</td>
<td><strong>add</strong>(v_2, -<strong>neg</strong>(v_4))</td>
</tr>
<tr>
<td>$v_6 = v_3 + v_5$</td>
<td><strong>add</strong>(v_3, v_5)</td>
</tr>
<tr>
<td>$v_7 = v_5 \cdot v_6$</td>
<td><strong>mul</strong>(v_5, v_6)</td>
</tr>
</tbody>
</table>

The following python code is an implementation of the underlying class (called forward), which demonstrates some of the overrides mentioned above (negation, absolute values, addition, multiplication). Full details of the logic behind the following implementation is discussed in Chapter 2.

```python
class forward:
    def __init__(self, value, ndimensions, order=4):
        self.order = order
        self.ndimensions = ndimensions
        self.zeroth = value
        self.first = numpy.zeros(ndimensions, numpy.object)
        for c in range(0, ndimensions):
            self.first[c] = numpy.zeros(self.order, numpy.object)
            for n in range(0, self.order):
                self.first[c][n] = decimal_float('0.0')

    def __neg__(self):
        new = forward(self.ndimensions, self.order)
        new.zeroth = -self.zeroth
        for c in range(0, self.ndimensions):
            for n in range(0, self.order):
                new.first[c][n] = -self.first[c][n]
        return new

    def __abs__(self):
        new = forward(self.ndimensions, self.order)
        new.zeroth = abs(self.zeroth)
        for c in range(0, self.ndimensions):
            for n in range(0, self.order):
                new.first[c][n] = abs(self.first[c][n])
```
return new

def __add__( self , other ):
    if isinstance( other , forward ) :
        new = forward( self.ndimensions , self.order )
        new.zeroth = self.zeroth + other.zeroth
        for c in range( 0 , self.ndimensions ) :
            for n in range( 0 , self.order ) :
                new.first[c][n] = self.first[c][n] + other.first[c][n]
    elif isinstance( other , decimal_float ) \
        or isinstance( other , decimal_complex ) \
        or isinstance( other , int ) :
        new = self.copy()
        new.zeroth += other
    else:
        raise AttributeError( '%s + %s not supported' \
            % ( self.__class__ , other.__class__ ) )
    return new

def __mul__( self , other ) :
    new = forward( self.ndimensions , self.order )
    if isinstance( other , forward ) :
        factorial = numpy.array( [ decimal_float( '1.0' ) , \
            decimal_float( '1.0' ) , decimal_float( '2.0' ) , \
            decimal_float( '6.0' ) , decimal_float( '24.0' ) ] )
        new.zeroth = self.zeroth*other.zeroth
        for c in range( 0 , self.ndimensions ) :
            for n in range( 0 , self.order ) :
                new.first[c][n] += self.zeroth*other.first[c][n]
                new.first[c][n] += self.first[c][n]*other.zeroth
                for m in range( 0 , n ) :
                    binom = factorial[n+1]/factorial[m+1]/factorial[n-m]
                    new.first[c][n] += binom \*
                    *self.first[c][m]*other.first[c][n-m+1]
    elif isinstance( other , int ) or isinstance( other , decimal_float ) \
        or isinstance( other , decimal_complex ) :
        new.zeroth = other*self.zeroth
        for c in range( 0 , self.ndimensions ) :
            for n in range( 0 , self.order ) :
                new.first[c][n] = other*self.first[c][n]
    else:
        raise AttributeError( '%s * %s not implemented' \
            % ( self.__class__ , other.__class__ ) )
    return new

This is an implementation of a fourth-order approach as discussed in Chapter 2.

In order to use this class, the uncertain variables \( x \) in a code can be initialized
with
which invokes the \_init\_ routine and \( d \) is the number of input random variables. At any subsequent point in the code, the higher-order derivatives associated with any particular variable \( y \) derived from operations involving \( x \) at some point prior (e.g. \( y = x^2 \)) can be inspected with the first array within the forward class. For example, the output of this code

\[
x = \text{forward}(3.0, 1) \\
y = x^2 \\
print y\text{.zeroth}, y\text{.first}
\]

is

\[
9.0 [6.0,1.0,0.0,0.0,0.0]
\]

where 9.0 is the value of \( y = x^2 \) evaluated at \( x = 3.0 \), 6.0 is the first derivative of \( y \) with respect to \( x \) (again evaluated at \( x = 3.0 \)), 1.0 is its second derivative, and so on.

### 1.6.2 Reverse-Mode Automatic-Differentiation

The above description is for the “forward” AD method. There is also a “reverse” approach; also best explained via example. In this approach, the Jacobian of the
function is utilized, which is given by

\[
J = \begin{bmatrix}
\frac{dv_0}{dv_0} & \frac{dv_1}{dv_0} & \frac{dv_2}{dv_0} & \frac{dv_3}{dv_0} & \frac{dv_4}{dv_0} & \frac{dv_5}{dv_0} & \frac{dv_6}{dv_0} & \frac{dv_7}{dv_0} \\
\frac{dv_0}{dv_0} & \frac{dv_1}{dv_1} & \frac{dv_2}{dv_1} & \frac{dv_3}{dv_1} & \frac{dv_4}{dv_1} & \frac{dv_5}{dv_1} & \frac{dv_6}{dv_1} & \frac{dv_7}{dv_1} \\
\frac{dv_0}{dv_1} & \frac{dv_1}{dv_1} & \frac{dv_2}{dv_1} & \frac{dv_3}{dv_1} & \frac{dv_4}{dv_1} & \frac{dv_5}{dv_1} & \frac{dv_6}{dv_1} & \frac{dv_7}{dv_1} \\
\frac{dv_0}{dv_2} & \frac{dv_2}{dv_2} & \frac{dv_3}{dv_2} & \frac{dv_4}{dv_2} & \frac{dv_5}{dv_2} & \frac{dv_6}{dv_2} & \frac{dv_7}{dv_2} & \frac{dv_7}{dv_2} \\
\frac{dv_0}{dv_3} & \frac{dv_3}{dv_3} & \frac{dv_3}{dv_3} & \frac{dv_4}{dv_3} & \frac{dv_5}{dv_3} & \frac{dv_6}{dv_3} & \frac{dv_7}{dv_3} & \frac{dv_7}{dv_3} \\
\frac{dv_0}{dv_4} & \frac{dv_4}{dv_4} & \frac{dv_4}{dv_4} & \frac{dv_4}{dv_4} & \frac{dv_4}{dv_4} & \frac{dv_4}{dv_4} & \frac{dv_4}{dv_4} & \frac{dv_7}{dv_4} \\
\frac{dv_0}{dv_5} & \frac{dv_5}{dv_5} & \frac{dv_5}{dv_5} & \frac{dv_5}{dv_5} & \frac{dv_5}{dv_5} & \frac{dv_5}{dv_5} & \frac{dv_5}{dv_5} & \frac{dv_7}{dv_5} \\
\frac{dv_0}{dv_6} & \frac{dv_6}{dv_6} & \frac{dv_6}{dv_6} & \frac{dv_6}{dv_6} & \frac{dv_6}{dv_6} & \frac{dv_6}{dv_6} & \frac{dv_6}{dv_6} & \frac{dv_7}{dv_6} \\
\frac{dv_0}{dv_7} & \frac{dv_7}{dv_7} & \frac{dv_7}{dv_7} & \frac{dv_7}{dv_7} & \frac{dv_7}{dv_7} & \frac{dv_7}{dv_7} & \frac{dv_7}{dv_7} & \frac{dv_7}{dv_7}
\end{bmatrix}
\]

Again, in order to explain the methodology, the best approach is via example. Choosing again the same analytically solvable educational example as described previously

\[
\phi(x_1, x_2) = \left[ \sin \left( \frac{x_1}{x_2} \right) + \frac{x_1}{x_2} - e^{x_2} \right] \left[ \frac{x_1}{x_2} - e^{x_2} \right]
\]

the system is again written in its vector format

\[
v = \begin{bmatrix}
    v_0 \\
v_1 \\
v_2 \\
v_3 \\
v_4 \\
v_5 \\
v_6 \\
v_7
\end{bmatrix}
= \begin{bmatrix}
x_1 \\
x_2 \\
v_0/v_1 \\
\sin(v_2) \\
\exp(v_1) \\
v_2 - v_4 \\
v_3 + v_5 \\
v_6v_5
\end{bmatrix}
\]

(1.42)
and using (1.40), its Jacobian matrix can be found

\[ J = \begin{bmatrix}
1 & 0 & 1/v_1 & 0 & 0 & 0 & 0 \\
0 & 1 & -v_0/v_1^2 & 0 & \exp(v_1) & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & v_6 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix} \] (1.43)

Note that \( J \) has some very useful properties. One is that it is a very sparse matrix and thus can be stored using sparse techniques; saving memory and computational time. Sparseness is obtained since at most each step involves only two variables, thus each line of the Jacobian matrix has at most three non-zero elements, which is relatively quite sparse when the dimensions of \( J \) are large. The other interesting property is that it can be written in terms of the identity matrix \( I \) and an upper-triangular matrix \( K \); \( J = I + K \). Finally, the derivative of interest can be obtained from \( \frac{d\phi(x_1,x_2)}{dx_1} = f_7 \) where \( f_n \) is given by the following recursive formula

\[ f_n = \begin{cases} 
\sum_{m=0}^{n-1} J_{nm} f_{n-1} & n > 0 \\
1 & n = 0
\end{cases} \] (1.44)

or equivalently by solving the following matrix equation

\[
\begin{bmatrix}
-1 & J_{n(n-1)1} & \ldots & \ldots & J_{nm1} \\
\vdots & \ddots & \ddots & \vdots & \vdots \\
-1 & J_{(m+2)(m+1)1} & \ldots & \ldots & J_{(m+2)m1} \\
-1 & & & \ldots & J_{(m+1)m1} \\
-1 & & & & \ldots
\end{bmatrix}
\begin{bmatrix}
dv_n/dv_m \\
\vdots \\
dv_{m+2}/dv_m \\
dv_{m+1}/dv_m \\
dv_m/dv_m
\end{bmatrix}
= \begin{bmatrix}
0 \\
\vdots \\
0 \\
0 \\
-1
\end{bmatrix} \] (1.45)

which can be computed efficiently using sparse matrix numerical solution methods since \( J \) is a very sparse matrix as is demonstrated by the Jacobian for the above example (1.43).

The reverse-mode implementation overrides the exact same calls that are overridden in the forward method. The following python code is an implementation of the
underlying reverse-mode class (called reversevalue). The full logic behind this implementation and further mathematical derivations and details are discussed in Chapter 3.

class reversevalue:

    def __init__( self , trace , value , deviation=None ):
        self.maxorder = 2
        self.value = value
        self.trace = trace
        self.index = trace.index
        trace.index += 1

        if deviation is not None:
            nvariables = trace.covariance.n
            newcovariance = triangular_matrix( nvariables + 1 )
            for n in range( 0 , nvariables ):
                for m in range( n , nvariables ):
                    newcovariance[n,m] = trace.covariance[n,m]
                    newcovariance[nvariables,nvariables] = deviation**2
            self.trace.covariance = newcovariance
            self.trace.indices.append( self.index )

    def __neg__( self ):
        svalue = self.value
        new = reversevalue( self.trace , -svalue )
        self.trace.appendjacobian( new.index , self.index , 1 , -1.0 )
        return new

    def __add__( self , other ):
        svalue = self.value
        if isinstance( other , reversevalue ):
            self._checktrace( other )
            ovalue = other.value
            new = reversevalue( self.trace , svalue + ovalue )
            self.trace.appendjacobian( new.index , self.index , 1 , 1.0 )
            self.trace.appendjacobian( new.index , other.index , 1 , 1.0 )
        else:
            new = reversevalue( self.trace , svalue + other )
            self.trace.appendjacobian( new.index , self.index , 1 , 1.0 )
        return new

    def __mul__( self , other ):
        svalue = self.value
        if isinstance( other , reversevalue ):
            if self.index == other.index: # handle squaring of existing value
                return self**2
            self._checktrace( other )
            ovalue = other.value
            new = reversevalue( self.trace , svalue*ovalue )
self.trace.appendjacobian( new.index, self.index, 1, ovalue )
self.trace.appendjacobian( new.index, other.index, 1, svalue )
self.trace.appendhessian( new.index, self.index, other.index, 
                           1, 1, 1.0 )

else:
    new = reversevalue( self.trace, svalue*other )
    self.trace.appendjacobian( new.index, self.index, 1, other )
return new

Note that the index variable is used to keep track of the indices of calculation steps in order to store the Jacobian and Hessian values in correct locations. The above implementation also makes use of a program trace class, which serves to store the Jacobian and Hessian matrices, and to provide functions to calculate the derivatives of interest via those stored matrices. A python implementation looks like

class reversetrace:
    def __init__( self ):
        self.index = 0
        self.indices = []
        self.hessian = hessianarray()
        self.jacobian = jacobianarray()
        self.covariance = triangular_matrix( 0 )

    def appendjacobian( self, n, m, orderm, value ):
        self.jacobian[n,m,orderm] = value

    def appendhessian( self, n, m, l, orderm, orderl, value ):
        self.hessian[n,m,l,orderm,orderl] = value

    def derivative1( self, k, n=None, d1={} ):
        if n is None:
            n = self.index - 1
        if n == k:
            return 1.0
        total = 0.0
        for m in range( k, n ):
            multiplier = self.jacobian[n,m,1]
            if multiplier != 0.0:
                if (k,n,m) not in d1.keys():
                    d1[(k,n,m)] = self.derivative1( k, m, d1 )
                total += multiplier*d1[(k,n,m)]
        return total

43
Note that the `derivative1` method above only calculates first order derivatives. Appropriate methods for higher-order derivatives have been implemented, but are not included here for brevity. One can use the methods described in Chapter 3 in order to derive and implement higher-order derivatives if they are of interest.

An example usage of this class follows. Note that a program trace needs to be initialized before initializing other variables unlike in the forward-mode. The following example can be used to produce derivatives for \( y = x^2 \)

```python
trace = reversetrace()
x = reversevalue( trace , 3.0 )
y = x*x
print x.value , trace.derivative1( 0 )
```

which produces the same values that the forward-mode implementation does

```
9.0 6.0
```

This has been a brief overview of automatic-differentiation, and the fine details will be fully discussed in the following chapters in conjunction with the SPAD methodology.
2.1 Introduction

The forward-mode SPAD methodology is one of two (equivalent) approaches that can be used to compute uncertainties in computations. The other is the reverse-mode SPAD, which will be deferred until the next Chapter. The term forward-mode refers to the order of the automatic-differentiation approach. In this case, the derivatives are computed in a logical forward manner (using previous values to determine subsequent ones). The following Chapter includes a discussion on the background, derivation, details, and application of this methodology.

2.2 SPAD Forward-Mode Single-Variate Methodology

The Taylor series approximation for \( \Phi(x) \) expanded about the expected value of an input variable \( x = \langle x \rangle \), can be written as

\[
\Phi(x) \approx \sum_{n=0}^{N} \frac{(x - \langle x \rangle)^n}{n!} \frac{d^n \Phi(x)}{dx^n} \bigg|_{x = \langle x \rangle}
\]

where \( N \) is the number of terms in the expansion, \( \Phi(x) \) is a chosen output produced by a CEM calculation, and \( x \) is an input with uncertainty. Typically, \( \Phi(x) \) will correspond to quantities such as radar cross-section, antenna gain, or input impedance;
while \( x \) may correspond to some geometrical parameter, frequency, or incidence angle. Because all of the derivatives of \( \Phi(x) \) are evaluated at deterministic values, \( x = \langle x \rangle \), it is possible to write a “separable” expression for the desired objective

\[
\langle \Phi(x) \rangle \approx \left\langle \sum_{n=0}^{N} \frac{(x - \langle x \rangle)^n}{n!} \frac{d^n \Phi(x)}{dx^n} \bigg|_{x=\langle x \rangle} \right\rangle
\]

\[
\approx \sum_{n=0}^{N} \frac{A_n d^n \Phi(x)}{n! dx^n} \bigg|_{x=\langle x \rangle}
\]

(2.2)

where \( A_n = \langle (x - \langle x \rangle)^n \rangle \) are the \( n \)-th order statistical central moments of the input random variable \( x \). Note that Equation (2.2) requires the corresponding moments of \( A_n \) to converge. The latter is true for many commonly used probability density functions of interest, such as Gaussian and uniform distributions. For example, the central moments for a Gaussian random variable are given by

\[
A_n = \langle (x - \langle x \rangle)^n \rangle = \begin{cases} 
0 & n \text{ odd} \\
(n - 1)!! (\Delta x)^n & n \text{ even}
\end{cases}
\]

(2.3)

where \( \Delta x \) is the standard deviation of \( x \). Combining (2.2) and (2.3), the expression for the expected value of \( \Phi(x) \) for a Gaussian variable \( x \) can be written as

\[
\langle \Phi(x) \rangle \approx \frac{N}{2} \sum_{n=0}^{N/2} \frac{(\Delta x)^{2n}}{(2n)!!} \frac{d^{2n} \Phi(x)}{dx^{2n}} \bigg|_{x=\langle x \rangle}
\]

For a uniform random variable \( x \), the central moments are given by

\[
A_n = \langle (x - \langle x \rangle)^n \rangle = \begin{cases} 
0 & n \text{ odd} \\
\frac{1}{n+1} (\Delta x \sqrt{3})^n & n \text{ even}
\end{cases}
\]

(2.4)

Combining (2.2) and (2.4), the expected value of a function of uniform random variable \( x \) can be found via

\[
\langle \Phi(x) \rangle \approx \frac{N}{2} \sum_{n=0}^{N/2} \frac{3^n (\Delta x)^{2n} d^{2n} \Phi(x)}{(2n+1)!! dx^{2n}} \bigg|_{x=\langle x \rangle}
\]

(2.5)
where the statistical parameters $\langle x \rangle$ and $\Delta x$ for the uniform distribution are related to the upper and lower limits ($b$ and $a$ respectively) over which that particular distribution is defined, i.e.

$$\langle x \rangle = \frac{b + a}{2}, \quad \Delta x = \frac{b - a}{2\sqrt{3}}$$ \hspace{1cm} (2.6)

From the above, it can be observed that if the higher-order derivatives of $\Phi(x)$ are known (and the series converges), then the expected value of $\Phi(x)$ can be found. Of course, practically speaking, the series needs to be truncated, so at best, this method can only find a good approximation for the expected value. Since the convergence is faster for small $\Delta x$, the term “small-perturbation” is apropos. This approach is “separable” in the sense that, for a given practical implementation, (2.2) needs to be computed only once. Once the derivatives are calculated for a specific expected value, it is rather fast to recompute it for any change in the probability distribution of the input random variable since those two parts of the resulting equation are separate. In other words, $A_n$ is replaceable by (2.3), (2.4), or any set of known central moments for the input variables.

2.2.1 Single-Variate Derivative Computation

It is clear that, at this point, the problem has been reduced to determining all of the derivatives up to a certain order of numerically computed function $\Phi(x)$, i.e. determining

$$\frac{d^n \Phi(x)}{dx^n} \bigg|_{x=\langle x \rangle}$$ \hspace{1cm} (2.7)

One way to find these derivatives is via an “automatic-differentiation” procedure, which will involve minimal modifications to the same deterministic CEM code that produced $\Phi(x)$. Indeed, for an object-oriented CEM code, the only modification will
be on the variable declarations because operators such as addition and multiplication can be overridden in such codes. Automatic-differentiation proceeds by decomposing the numerical operations used to compute $\Phi(x)$ into elementary operations that can be differentiated individually and later combined using the chain rule. References [28] and [29] expound on automatic-differentiation from a general viewpoint.

Automatic-differentiation is an alternative approach to either numerical differentiation (i.e., finite-differencing of multiple solutions at nearby input data points) or symbolic differentiation (using symbolic mathematical packages). Numerical differentiation is problematic in this context because $\Phi(x)$ is subject to “noise” due to numerical errors. It is well-known that numerical differentiation of such data is ill-posed: small errors in the data can produce large errors in the numerical approximation of the derivative [30]. This problem is compounded for the higher-order derivatives that are required here. With that said, finite-differencing might prove useful in certain situations and can be a potential avenue for future research. Symbolic differentiation, on the other hand, is problem-dependent and cannot be fully done in a practical fashion for a function $\Phi(x)$ that is generated by codes of very high complexity, only to relatively simple functions. Still, symbolic differentiation can be hybridized with the automatic-differentiation proposed here, i.e. with the application of the former to specific, limited parts of the computation chain. This could speed up the computation but at the expense of automation and generality and may be another avenue for future research.
2.2.2 Single-Variate Elementary Operations

A CEM code is based on the execution of a sequence of arithmetic operations such as additions and multiplications, and evaluation of functions such as trigonometric functions, exponentials, special functions, etc. The fundamental idea of automatic-differentiation is that, through application of the chain rule to each arithmetic operations on a CEM code, it is possible to compute derivatives of arbitrary order exactly (to numerical accuracy).

It is convenient to first explain two approaches for obtaining the initial derivative values to be used as inputs for the chain rule computation. One approach would be to use a symbolic math package to obtain in closed-form the (analytical) derivatives of certain expressions along the numerical chain. The main advantage to this approach is that some chain rule computations will be avoided because the derivatives are simplified upfront; however, this has the drawback that the implementation will be tied to the particular problem at hand. Another approach is to utilize the fact that the derivatives of the input random variables are known and given by

\[
\frac{d^n}{dx^n} x \bigg|_{x=\langle x \rangle} = \begin{cases} 
\langle x \rangle & n = 0 \\
1 & n = 1 \\
0 & n \geq 2
\end{cases}
\]  

(2.8)

This approach has the advantage in that it is generally applicable to any problem; however, since there are no simplifications upfront, the number of numerical calculations will be higher than for the symbolic approach.

Starting from the above known derivatives (2.8), the derivatives of any function compositions of the above can also be obtained via automatic-differentiation. For example, for an addition operation \( f(x) = g(x) + h(x) \) (with higher-order derivatives
of \( g(x) \) and \( h(x) \) already computed), the new derivatives are

\[
\frac{d^n f(x)}{dx^n} \bigg|_{x=\langle x \rangle} = \frac{d^n}{dx^n} \left[ g(x) + h(x) \right] \bigg|_{x=\langle x \rangle} = \frac{d^n}{dx^n} g(x) \bigg|_{x=\langle x \rangle} + \frac{d^n}{dx^n} h(x) \bigg|_{x=\langle x \rangle}
\]

(2.9)

where \( n = \{0, \ldots, N\} \) and again \( N \) is the number of terms used in the Taylor series expansion. For multiplication, the generalized Leibniz rule (a widely known higher-order generalization of the product rule) can be applied. Given known derivatives for \( g(x) \) and \( h(x) \), the derivatives of \( f(x) = g(x)h(x) \) are

\[
\frac{d^n f(x)}{dx^n} \bigg|_{x=\langle x \rangle} = \frac{d^n}{dx^n} \left[ g(x)h(x) \right] \bigg|_{x=\langle x \rangle} = \sum_{m=0}^{\infty} \binom{n}{m} \frac{d^m}{dx^m} g(x) \frac{d^{n-m}}{dx^{n-m}} h(x) \bigg|_{x=\langle x \rangle}
\]

(2.10)

where

\[
\binom{n}{m} = \frac{n!}{m!(n-m)!}
\]

(2.11)

is the binomial coefficient, which gives the number of ways to choose \( m \) elements from a set of \( n \). Equation (2.10) is broadly known, but [31] puts it in a context useful for the approach pursued in this effort.

To obtain derivatives of all other operations, the Faà di Bruno formula (a higher-order generalization of the chain rule) can be applied. This rule is also defined in a useful context in [31]. However, in terms of an efficient numerical implementation it is far more ideal to make use of the Bell polynomial form as described in [32]. In this expression, lower-order polynomials are used recursively to determine higher-order
ones. Using this form, the derivatives for \( f(x) = h(g(x)) \) can be found via

\[
\frac{d^n}{dx^n} f(x) \bigg|_{x=\langle x \rangle} = \frac{d^n}{dx^n} h(g(x)) \bigg|_{x=\langle x \rangle} = \sum_{m=0}^{n+1} B_{mn} \frac{d^{m+1} h(u)}{du^{m+1}} \bigg|_{u=g(\langle x \rangle)}
\]

(2.12)

for \( n > 0 \) and

\[
\frac{d^0}{dx^0} f(x) \bigg|_{x=\langle x \rangle} = h(g(\langle x \rangle))
\]

(2.13)

where the partial Bell polynomials \( B_{mn} \) are given by

\[
B_{mn} = \frac{T_{m,(n+1)}}{(m + 1)!}
\]

(2.14)

where

\[
T_{mn} = \begin{cases} 
\frac{d^n g(x)}{dx^n} \bigg|_{x=\langle x \rangle} & m = 0 \\
\sum_{k=1}^{n} \binom{n}{k} T_{0k} T_{(m-1),(n-k)} & m > 0, n > m \\
0 & n \leq m
\end{cases}
\]

(2.15)

For generality, the function \( h(u) \) is explicitly left undefined above since it can be validly replaced by any operation with known derivatives. For example, if the operation of interest is \( f(x) = e^{g(x)} \), then \( h(u) = e^u \), and its straightforward set of higher-order derivatives are

\[
\frac{d^n}{du^n} h(u) \bigg|_{u=g(\langle x \rangle)} = e^{g(\langle x \rangle)}
\]

(2.16)

which can be inserted into the chain rule formula (2.12). Note that \( g(\langle x \rangle) \) is available as one of the known derivatives (the zeroth one) of the input function \( g(x) \).

Another interesting set of operations that require utilization of the chain rule are powers. The function of interest in this case is \( f(x) = g(x)^m \) or \( h(u) = u^m \), which has higher-order derivatives given by

\[
\frac{d^n}{du^n} h(u) \bigg|_{u=g(\langle x \rangle)} = \frac{m!}{(m-n)!} g(\langle x \rangle)^{m-n}
\]

(2.17)
or in terms of an efficient implementation

\[ \left. \frac{d^0}{du^0} h(u) \right|_{u=g(\langle x \rangle)} = g(\langle x \rangle)^m \]  

(2.18)

with all other terms given recursively by

\[ \left. \frac{d^n}{du^n} h(u) \right|_{u=g(\langle x \rangle)} = \frac{1 - n + m}{g(\langle x \rangle)} \left( \frac{d^{n-1}}{du^{n-1}} h(u) \right) \]  

(2.19)

Note that various other mathematical operations can be obtained by reusing the above equations. For example, the higher-order derivatives of the square root operation can be determined with \( m = 0.5 \), and reciprocals (and consequently the division operation) can be obtained with \( m = -1 \).

There may also be instances of special functions within the calculation of interest. For those scenarios, the same process can be followed to obtain the derivatives of those special functions. For the results presented in this dissertation, there is a brief diversion to discuss the specific special functions when it is encountered in the calculation.

All of these operations (addition, multiplication, and the chain rule) produce exact derivatives (to within machine precision) at every step of the computation. However, in terms of a real numerical implementation, machine precision can be an issue because rounding errors can accumulate along the numerical chain. A straightforward solution to this problem is to use decimal floating point values, rather than the default binary floating point used in most numerical computations today. This does cause a performance hit on hardware that does not support the IEEE 754-2008 standard [33], which actually includes most hardware available as of 2012.
2.3 SPAD Forward-Mode Multivariate Methodology

Single-variate uncertainty, while interesting, is not of primary interest itself since such computations can easily be achieved via Monte Carlo methods and simple integration techniques as described in Chapter 1. It is however useful as a starting point for the more general problem; multivariate uncertainty, which will be discussed in detail in this section.

For this problem, the output function of interest \( \Phi(\mathbf{x}) \) is a function of multiple random input variables \( \mathbf{x} \), where \( \mathbf{x} = \{x_0, ..., x_d\} \) and \( d \) is the number of random variables. Note that these variables are all potentially correlated. This relationship is described via covariance coefficients \( C(x_l, x_m) \) between pairs of variables, the set of which form a symmetric \( d \times d \) matrix. Note that covariance is related to correlation via

\[
C(x_l, x_m) = \Delta x_l \Delta x_m \rho(x_l, x_m)
\]

(2.20)

where \( \rho(x_l, x_m) \) is the correlation coefficient between variables \( x_l \) and \( x_m \) and \( -1 \leq \rho(x_l, x_m) \leq 1 \).

Again, the goal of this effort is to obtain an approximation for the expected value of any function interest. In this case, the same Taylor series expansion can be applied, although now in multiple dimensions

\[
\langle \Phi(\mathbf{x}) \rangle \approx \sum_{n_0=0}^{N_0} \cdots \sum_{n_d=0}^{N_d} \left[ \prod_{k=0}^{d} \left( x_k - \langle x_k \rangle \right)^{n_k} \frac{n_k!}{d^{n_k}} \frac{d^{n_k}}{dx_k} \right] \Phi(\mathbf{x}) \bigg|_{\mathbf{x} = \langle \mathbf{x} \rangle} \approx \sum_{n_0=0}^{N_0} \cdots \sum_{n_d=0}^{N_d} A_n B_n \left[ \prod_{k=0}^{d} \frac{d^{n_k}}{dx_k} \right] \Phi(\mathbf{x}) \bigg|_{\mathbf{x} = \langle \mathbf{x} \rangle}
\]

(2.21)
where \( N_0, ..., N_d \) are the number of terms used in the expansion for each input variable (usually chosen to be the same number, \( N = N_0 = ... = N_d \)), \( B_n \) is a simple combination of factorials and is given by

\[
B_n = \prod_{k=0}^{d} \frac{1}{n_k!}
\]  

(2.22)

and

\[
A_n = \left\langle \prod_{k=0}^{d} (x_k - \langle x_k \rangle)^{n_k} \right\rangle
\]  

(2.23)

are the higher-order multivariate central moments of \( x \). Similar to the single-variate case, these are all solely dependent on the covariances between the elements of \( x \). As an example, if all of the elements of \( x \) are jointly Gaussian, then the central moments are

\[
A_n = \begin{cases} 
\sum \prod_{l,m} C(x_l, x_m) & P_d \text{ even} \\
0 & P_d \text{ odd}
\end{cases}
\]  

(2.24)

where \( \sum \prod_{l,m} \) symbolizes a sum over all \( P_d/2 \) products of the covariances between elements of \( x \) [34] and

\[
P_d = \sum_{k=0}^{d} n_k
\]  

(2.25)

For example, if \( P_d = 4 \) then \( A_n \) is all of the combinations of pairs of covariances, and if \( P_d = 6 \) then \( A_n \) is all of the combinations of triplets of covariances. There are \( (P_d - 1)! \) of these terms/combinations in \( A_n \) (for example there are 3 terms for \( P_d = 4 \) and 15 terms for \( P_d = 6 \)). For other multivariate distributions (i.e. not jointly Gaussian), the form of \( A_n \) is not quite so straightforward, but it can certainly be derived.
2.3.1 Multivariate Derivative Computation

Finally with respect to (2.21),
\[
\left[ \prod_{k=0}^{d} \frac{d^{m_k}}{dx_k^{n_k}} \right] \Phi(x) \bigg|_{x=\langle x \rangle}
\]
are the higher-order multidimensional derivatives of \( \Phi(x) \). Similar to the single-variate case, these are the values that need to be determined throughout the computation. Conveniently, the foundations established in the previous single-variate section can be reused. However, the computational cost will be significantly greater since there are many more derivative values involved.

Note that it is also possible to determine all of the higher-order moments of that function as will be described later in this Chapter.

2.3.2 Multivariate Elementary Operations

The scheme for initializing derivative values in this case is given by
\[
\frac{d^k}{dx_1^k} x_m \bigg|_{x=\langle x \rangle} = \begin{cases} 
  \langle x_m \rangle & m = l, k = 0 \\
  1 & m = l, k = 1 \\
  0 & \text{otherwise}
\end{cases}
\]
(2.26)

Via induction, the outcome of multivariate addition is
\[
\left[ \prod_{k=0}^{d} \frac{d^{m_k}}{dx_k^{n_k}} \right] \Phi(x) \bigg|_{x=\langle x \rangle} \\
= \left[ \prod_{k=0}^{d} \frac{d^{m_k}}{dx_k^{n_k}} \right] [g(x) + h(x)] \bigg|_{x=\langle x \rangle} \\
= \left[ \prod_{k=0}^{d} \frac{d^{m_k}}{dx_k^{n_k}} \right] g(x) \bigg|_{x=\langle x \rangle} + \left[ \prod_{k=0}^{d} \frac{d^{m_k}}{dx_k^{n_k}} \right] h(x) \bigg|_{x=\langle x \rangle}
\]
(2.27)
Also, the multivariate multiplication rule \([31]\) is given by

\[
\prod_{k=0}^{d} \frac{d^{n_k}}{dx_k^{n_k}} \Phi(x) \bigg|_{x=\langle x \rangle} = \prod_{k=0}^{d} \frac{d^{n_k}}{dx_k^{n_k}} \left[ g(x) h(x) \right] \bigg|_{x=\langle x \rangle} = \left[ \prod_{k=0}^{d} \frac{d^{m_k}}{dx_k^{m_k}} \left[ g(x) \right] \bigg|_{x=\langle x \rangle} \right] \left[ \prod_{k=0}^{d} \frac{d^{m_k}}{dx_k^{m_k}} \left[ h(x) \right] \bigg|_{x=\langle x \rangle} \right] \cdot \left[ \prod_{k=0}^{d} \frac{d^{m_k-m_k}}{dx_k^{m_k-m_k}} \right] \left[ \frac{d}{du} h(u) \bigg|_{u=g(\langle x \rangle)} \right] (2.28)
\]

which can be derived by applying the single-variate multiplication rule (Faà di Bruno Formula) to progressively higher orders and extrapolating to express the general multivariate rule.

It is rather difficult to explicitly state the multivariate chain rule, and it does not appear to be stated explicitly in \([31]\). An approach for determining the indices is as follows

\[
\prod_{k=0}^{d} \frac{d^{n_k}}{dx_k^{n_k}} \Phi(x) \bigg|_{x=\langle x \rangle} = \prod_{k=0}^{d} \frac{d^{m_k}}{dx_k^{m_k}} h(g(x)) \bigg|_{x=\langle x \rangle} = \sum_{k=1}^{P_d} D(k, n) \frac{d^k}{du^k} h(u) \bigg|_{u=g(\langle x \rangle)} (2.29)
\]

for \(P_d > 0\) where \(P_d\) is defined in Equation (2.25) and

\[
\prod_{k=0}^{d} \frac{d^0}{dx_k^0} \Phi(x) \bigg|_{u=g(\langle x \rangle)} = h(g(\langle x \rangle)) (2.30)
\]

for \(P_d = 0\). Note that

\[
\frac{d^k}{du^k} h(u) \bigg|_{u=g(\langle x \rangle)} (2.31)
\]

are the same single dimensional higher-order derivatives of the outside function \(h(u)\) as described in detail in the previous section. \(D(k, n)\) are various combinations of...
the derivatives of the function \( g(x) \). These terms do not follow an easily definable pattern, so they must be generated by applying the chain rule and product rule until the term of interest is obtained.

For example, starting from \( f(x) = h(g(x)) \), apply the chain rule to get

\[
\frac{d}{dx_0} \Phi(x) \bigg|_{x=(x)} = \frac{dg(x)}{dx_0} \bigg|_{x=(x)} \frac{dh(u)}{du} \bigg|_{u=g(x)}
\]

which provides an element of \( D \)

\[
D(1, \{1, 0, ..., 0\}) = \frac{dg(x)}{dx_1} \bigg|_{x=(x)}
\]

Next apply the chain rule and product rule to (2.32) to obtain for example the next derivative

\[
\frac{d^2}{dx_0^2} \Phi(x) = \frac{d^2g(x)}{dx_0^2} \bigg|_{x=(x)} \frac{dh(u)}{du} \bigg|_{u=g(x)}
\]

\[
+ \left( \frac{dg(x)}{dx_0} \right)^2 \bigg|_{x=(x)} \frac{d^2h(u)}{du^2} \bigg|_{u=g(x)}
\]

which provides two more elements of \( D \)

\[
D(1, \{2, 0, ..., 0\}) = \frac{d^2g(x)}{dx_2} \bigg|_{x=(x)}
\]

\[
D(2, \{2, 0, ..., 0\}) = \left( \frac{dg(x)}{dx_1} \right)^2 \bigg|_{x=(x)}
\]

The same approach can be applied to (2.32) to obtain various other derivatives

\[
\frac{d}{dx_0} \frac{d}{dx_1} \Phi(x) = \frac{d}{dx_0} \frac{d}{dx_1} g(x) \bigg|_{x=(x)} \frac{dh(u)}{du} \bigg|_{u=g(x)}
\]

\[
= \frac{dg(x)}{dx_0} \bigg|_{x=(x)} \frac{dg(x)}{dx_1} \bigg|_{x=(x)} \frac{d^2h(u)}{du^2} \bigg|_{u=g(x)}
\]

which gives another two elements of \( D \)

\[
D(1, \{1, 1, 0, ..., 0\}) = \frac{d}{dx_0} \frac{d}{dx_1} g(x) \bigg|_{x=(x)}
\]

\[
D(2, \{1, 1, 0, ..., 0\}) = \frac{dg(x)}{dx_0} \bigg|_{x=(x)} \frac{dg(x)}{dx_1} \bigg|_{x=(x)}
\]
This approach can be applied further to obtain the required $D(k, n)$ terms.

### 2.4 Computational Complexity Estimates

Since the SPAD methodology does add additional computational complexity as compared to the same problem done deterministically, it is prudent to understand the resulting costs. The additional complexity is best described by the number of operations required for the computation with uncertainty versus the number of operations for the same computation done deterministically. With respect to the single-variate case, addition with uncertainty (2.9) requires $N$ operations for every deterministic addition, multiplication (2.10) requires $3N^2/2$ operations for every deterministic multiplication, and all other operations (2.12) require $N2^{N-1}$ operations for every deterministic operation. In terms of memory usage, the algorithm with uncertainty requires $N$ bytes for every deterministic byte used.

For the multivariate case, addition (2.27) requires $N^d$ operations for every deterministic addition, multiplication (2.28) requires $3N^{2d}/2$ operations for every deterministic multiplication, and all other operations (2.29) require $N^d2^{N^d-1}$ for every deterministic operation. In terms of storage, the algorithm with multivariate uncertainty requires $N^d$ bytes for every deterministic byte. Again, $N$ is the number of terms used in the expansions, and $d$ is the number of random variables. For these conclusions, it has been assumed that the number of terms in each dimension are the same. At first glance, these costs appear to be severe. However, in practice it is possible to obtain good results with a small number of terms. For the results presented in this dissertation, an optimized fourth-order approach was used, which is discussed next.
2.5 An Optimized Fourth-Order Multivariate SPAD Implementation

In order to develop an accurate implementation of this method while minimizing the computational overhead, a fourth-order SPAD approach was applied to generate all the results in this Chapter.

Starting from (2.21) and keeping only derivatives up to the fourth-order, the expected value can be written as

\[
\langle \Phi(x) \rangle \approx \Phi(\langle x \rangle) + \sum_{n=0}^{d} \sum_{l=1}^{4} \frac{A_{10}(x_n, 0)}{l!} \frac{d^l}{dx_n^l} \Phi(x) \bigg|_{x=\langle x \rangle} + \sum_{n=0}^{d} \sum_{m=n+1}^{d} \sum_{l=1}^{3} \sum_{k=1}^{l+k \leq 4} \frac{A_{lk}(x_n, x_m)}{l!k!} \frac{d^l}{dx_n^l} \frac{d^k}{dx_m^k} \Phi(x) \bigg|_{x=\langle x \rangle}
\]

(2.40)

where \(d\) is the number of input random variables as described before, and \(A_{lk}\) (the central moments) are defined as

\[
A_{lk}(x_n, x_m) = \left\langle (x_n - \langle x_n \rangle)^l (x_m - \langle x_m \rangle)^k \right\rangle
\]

(2.41)
As an example, the non-zero $A_{lk}$ for a set of jointly Gaussian input random variables are

\[
A_{20}(x_n, 0) = C(x_n, x_n) \tag{2.42}
\]
\[
A_{40}(x_n, 0) = 3C^2(x_n, x_n) \tag{2.43}
\]
\[
A_{11}(x_n, x_m) = C(x_n, x_m) \tag{2.44}
\]
\[
A_{13}(x_n, x_m) = 3C(x_n, x_n)C(x_n, x_m) \tag{2.45}
\]
\[
A_{31}(x_n, x_m) = 3C(x_m, x_m)C(x_m, x_n) \tag{2.46}
\]
\[
A_{22}(x_n, x_m) = 2C^2(x_n, x_m) + C(x_n, x_n)C(x_m, x_m) \tag{2.47}
\]

Note that (2.40) is not completely fourth-order since terms involving more than two variables are intentionally excluded, e.g.

\[
\left. \frac{d^2}{dx_n^2} \frac{d^1}{dx_m^1} \frac{d^1}{dx_l^1} \Phi(x) \right|_{x=(\bar{x})} \quad (n \neq m \neq l) \tag{2.48}
\]

and

\[
\left. \frac{d^1}{dx_n^1} \frac{d^1}{dx_m^1} \frac{d^1}{dx_l^1} \frac{d^1}{dx_k^1} \Phi(x) \right|_{x=(\bar{x})} \quad (n \neq m \neq l \neq k) \tag{2.49}
\]

This is done to keep the scaling of the computational complexity manageable at $O(d^2)$. The memory required for the (pseudo) fourth-order approximation above is $18d^2$ versus $N^d$ for the general multivariate case.
The complexity of the automatic-differentiation procedure is also much reduced.

An example multiplication update equation is

$$\begin{align*}
\frac{d^1}{dx^1_n} \frac{d^3}{dx^3_n} \Phi(x) \bigg|_{x=\langle x \rangle} = & \frac{d^0}{dx^0_n} \frac{d^0}{dx^0_n} g(x) \bigg|_{x=\langle x \rangle} \frac{d^1}{dx^1_n} \frac{d^3}{dx^3_n} h(x) \bigg|_{x=\langle x \rangle} \\
+ & 3 \frac{d^0}{dx^0_n} \frac{d^1}{dx^1_n} g(x) \bigg|_{x=\langle x \rangle} \frac{d^1}{dx^1_n} \frac{d^2}{dx^2_n} h(x) \bigg|_{x=\langle x \rangle} \\
+ & 3 \frac{d^0}{dx^0_n} \frac{d^2}{dx^2_n} g(x) \bigg|_{x=\langle x \rangle} \frac{d^1}{dx^1_n} \frac{d^1}{dx^1_n} h(x) \bigg|_{x=\langle x \rangle} \\
+ & \frac{d^0}{dx^0_n} \frac{d^0}{dx^0_n} g(x) \bigg|_{x=\langle x \rangle} \frac{d^1}{dx^1_n} \frac{d^3}{dx^3_n} h(x) \bigg|_{x=\langle x \rangle} \\
+ & 3 \frac{d^1}{dx^1_n} \frac{d^1}{dx^1_n} g(x) \bigg|_{x=\langle x \rangle} \frac{d^0}{dx^0_n} \frac{d^2}{dx^2_n} h(x) \bigg|_{x=\langle x \rangle} \\
+ & 3 \frac{d^1}{dx^1_n} \frac{d^2}{dx^2_n} g(x) \bigg|_{x=\langle x \rangle} \frac{d^0}{dx^0_n} \frac{d^1}{dx^1_n} h(x) \bigg|_{x=\langle x \rangle} \\
+ & \frac{d^1}{dx^1_n} \frac{d^3}{dx^3_n} g(x) \bigg|_{x=\langle x \rangle} \frac{d^0}{dx^0_n} \frac{d^0}{dx^0_n} h(x) \bigg|_{x=\langle x \rangle} \\
+ & 3 \frac{d^1}{dx^1_n} \frac{d^1}{dx^1_n} h(x) \bigg|_{x=\langle x \rangle} \frac{d^0}{dx^0_n} \frac{d^3}{dx^3_n} h(x) \bigg|_{x=\langle x \rangle} \\
+ & 3 \frac{d^1}{dx^1_n} \frac{d^2}{dx^2_n} h(x) \bigg|_{x=\langle x \rangle} \frac{d^0}{dx^0_n} \frac{d^1}{dx^1_n} h(x) \bigg|_{x=\langle x \rangle} \\
+ & \frac{d^1}{dx^1_n} \frac{d^3}{dx^3_n} h(x) \bigg|_{x=\langle x \rangle} \frac{d^0}{dx^0_n} \frac{d^0}{dx^0_n} h(x) \bigg|_{x=\langle x \rangle}
\end{align*}$$

(2.50)
Other multiplication formulas can be derived in a rather straightforward manner using (2.28). An example of one of the chain rule update equations is

\[
\begin{align*}
\frac{d^1}{dx^1} \frac{d^3}{dx^3} \Phi(x) & \bigg|_{x=(x)} \\
= & \frac{d^1}{dx^1} \frac{d^3}{dx^3} g(x) \bigg|_{x=(x)} \frac{d^1 h(u)}{du^1} \bigg|_{u=g(x)} \\
+ & 3 \frac{d^1}{dx^1} \frac{d^2}{dx^2} g(x) \bigg|_{x=(x)} \frac{d^0}{dx^0} \frac{d^1}{dx^1} g(x) \bigg|_{x=(x)} \frac{d^2 h(u)}{du^2} \bigg|_{u=g(x)} \\
+ & 3 \frac{d^1}{dx^1} \frac{d^1}{dx^1} g(x) \bigg|_{x=(x)} \frac{d^0}{dx^0} \frac{d^2}{dx^2} g(x) \bigg|_{x=(x)} \frac{d^2 h(u)}{du^2} \bigg|_{u=g(x)} \\
+ & \frac{d^1}{dx^1} \frac{d^0}{dx^0} g(x) \bigg|_{x=(x)} \frac{d^0}{dx^0} \frac{d^3}{dx^3} g(x) \bigg|_{x=(x)} \frac{d^2 h(u)}{du^2} \bigg|_{u=g(x)} \\
+ & 3 \frac{d^1}{dx^1} \frac{d^1}{dx^1} g(x) \bigg|_{x=(x)} \frac{d^0}{dx^0} \frac{d^1}{dx^1} g(x) \bigg|_{x=(x)} \frac{d^3 h(u)}{du^3} \bigg|_{u=g(x)} \\
+ & \frac{d^0}{dx^0} \frac{d^2}{dx^2} g(x) \bigg|_{x=(x)} \frac{d^3 h(u)}{du^3} \bigg|_{u=g(x)} \\
+ & \frac{d^1}{dx^1} \frac{d^0}{dx^0} g(x) \bigg|_{x=(x)} \frac{d^0}{dx^0} \frac{d^1}{dx^1} g(x) \bigg|_{x=(x)} \frac{d^4 h(u)}{du^4} \bigg|_{u=g(x)} \\
\end{align*}
\]

(2.51)

The other chain rule update equations can be derived using (2.29), but it was found to be a bit easier to manually start from the lowest order equations and derive the subsequent higher-order formulas by applying the chain and product rules as needed.

One final consideration is that the numerical complexity of (2.40) can be reduced to \(O(d)\) by eliminating the third term in that equation. This approximation is only valid if correlations between input variables can be ignored (for example, if they are known to be uncorrelated).
2.6 Higher-order Statistical Moments

Given any one of these methods for obtaining the derivatives (and consequently the expected value), it is relatively straightforward to determine higher-order statistical moments as well. For example, the standard deviation can be computed by defining \( \Lambda(x) = [\Phi(x)]^2 \), calculating the higher-order derivatives of \( \Lambda(x) \) via the routines as described in Appendix A, and using those values to determine the average \( \langle \Lambda(x) \rangle \) (2.2). Finally the standard deviation for function \( \Phi(x) \) can be calculated from

\[
\Delta \Phi(x) = \sqrt{\langle \Lambda(x) \rangle - \langle \Phi(x) \rangle^2}
\] (2.52)

The same general idea can be applied to obtain any higher-order statistical moment of interest.

2.7 Induced Line Charge Example

According to [35] the deterministic electrostatic charge induced on a cylindrical wire (Figure 2.7) can be obtained by solving the matrix equation

\[
p = Z^{-1}v
\] (2.53)

where \( p_n \) are the induced line charges to be calculated, \( v_m = 4\pi\epsilon_0 V_m \), and the elements of \( Z \) are given by

\[
Z_{mn} = \begin{cases} 
2 \ln \left[ \frac{s/2 + \sqrt{a_m^2 + (s/2)^2}}{a_m} \right] & m - n = 0 \\
\ln \left[ \frac{c_{mn} + \sqrt{c_{mn}^2 + d_{mn}^2}}{d_{mn} + \sqrt{d_{mn}^2 + a_m^2}} \right] & 0 < |m - n| \leq 2 \\
\ln \left[ \frac{c_{mn}}{d_{mn}} \right] & |m - n| > 2 
\end{cases}
\] (2.54)

where the wire is divided into \( N \) segments over length \( l \), \( V_m \) is the applied voltage, and \( a_m \) is the wire radius at segment \( m \). Note that \( c_{mn} = l_{mn} + s/2, d_{mn} = l_{mn} - s/2, l_{mn} = |y_m - y_n|, y_n = ns, s = l/N, n = \{0, ..., N\}, \) and \( m = \{0, ..., N\} \).
Using the fourth-order multivariate approach described in Appendix D, the radius $a_m$ of each segment was chosen to be the uncertain parameter. For this example, the expected value of the radii were chosen to be the same for each segment $\langle a_m \rangle = 1$ millimeter and the standard deviations were chosen to be $\Delta a_m = \langle a_m \rangle / (10 + m)$. It is assumed that the radii are uncorrelated

$$C(a_m, a_n) = \begin{cases} 0 & m \neq n \\ \Delta a_m^2 & m = n \end{cases}$$

(2.55)

in this case.

Using a deterministic line charge code based off of the above equations from [35], the multivariate forward-mode SPAD approach described previously was utilized. The core of the approach was implemented using object oriented methods to override the mathematical operations (i.e. addition, multiplication, exponentiation, etc.) already
in use by the deterministic implementation. Hence with this approach, an uncertainty-
augmented code was easily written by recycling an existing deterministic code.

Note that the higher-order derivatives of the natural logarithm were needed to
calculate the elements of the $Z$ matrix. These are given by

$$
\frac{d^n}{du^n} \ln[u] \bigg|_{u=g(\langle x \rangle)} = \begin{cases} 
\ln[g(\langle x \rangle)] & n = 0 \\
\frac{d^{n-1}}{du^{n-1}} u^{-1} \bigg|_{u=g(\langle x \rangle)} & n > 0
\end{cases} \tag{2.56}
$$

where the derivatives of $u^{-1}$ are given by the power derivatives described in Appendix
A (2.19).

Also note that the matrix equation (2.53) was solved using Gaussian elimination,
but any matrix solver, either direct or iterative, could have been used since the
underlying mathematical operators are simply overridden, and the solution technique
itself does not matter..

Results for two scenarios for this example were calculated. In Figure 2.2, the line
is divided into 5 segments and in Figure 2.3, the line is divided into 16 segments. As
can be seen, the new method agrees very well with a Monte Carlo reference using $10^6$
sample points.

2.8 Infinite Parallel Wire Example

While the previous electrostatics problem is revealing, the following example will
demonstrate a more interesting electrodynamics scenario. For this case, the scattering
from a set of infinite wires will be evaluated where the positions/placement of the
wires are uncertain by a small amount (Figure 2.8).

Just like the line charge example, the goal is to solve the matrix equation

$$
p = Z^{-1}v \tag{2.57}
$$
Figure 2.2: Induced line charge uncertainty: 5 segments.
\[ \Delta a = \frac{\langle a \rangle}{10 + n} \text{ on } n\text{'th segment}, \quad \langle a \rangle = 0.001 \text{m}, \quad V = 1 \text{V} \]

Figure 2.3: Induced line charge uncertainty: 16 segments.

Figure 2.4: Infinite parallel wire configuration
where for infinite parallel wire scattering [36], the known elements are given by

\[ v_m = e^{-i(k_{xi}x_m + k_{yi}y_m)} \]

\[ Z_{mn} = H_0^{(2)}(k_{\rho_{mn}}) \quad (2.58) \]

and again the goal is to solve for the induced currents, i.e. the \( p_n \) values. Note that \( x_m \) and \( y_m \) are the coordinates of wire \( m \), \( a_m \) is the radius of wire \( m \), \( m = \{0, ..., N\} \), \( n = \{0, ..., N\} \), \( N \) is the number of wires, \( k = 2\pi/\lambda \) is the wavenumber, \( \lambda \) is the wavelength, \( k_{\rho} = \sqrt{k^2 - k_{zi}^2} \), \( k_{xi} = k \sin \theta_i \cos \phi_i \), \( k_{yi} = k \sin \theta_i \sin \phi_i \), \( k_{zi} = k \cos \theta_i \) and \( \phi_i \) are the angles of the incident wave, and \( \rho_{mn} \) is given by

\[ \rho_{mn} = \begin{cases} a_m & m = n \\ \sqrt{(x_m - x_n)^2 + (y_m - y_n)^2} & m \neq n \end{cases} \quad (2.59) \]

Once the induced currents \( p_n \) are found, the scattered field can be obtained by summing the radiated field of each individual wire according to [36] as

\[ E(\phi_s) = \sum_{n=0}^{N-1} p_n e^{ik_{\rho}(x_n \cos \phi_s + y_n \sin \phi_s)} \quad (2.60) \]

where \( \phi_s \) is the observation angle.

Note that in order to apply the small-perturbation approach, the higher-order derivatives of the Hankel function are needed. These can be determined by applying the well-known recursion relation [37]

\[ \frac{d}{du} H_0^{(2)}(u) = \frac{1}{2} \left[ H_0^{(2)}(u) - H_1^{(2)}(u) \right] \quad (2.61) \]

many times, thus producing the needed higher-order form, which is

\[ \frac{d^n}{du^n} H_m^{(2)}(u) \bigg|_{u = g(\langle x \rangle)} = \frac{1}{2^n} \sum_{k=0}^{n} (-1)^k \binom{n}{k} H_{m+2k-n}^{(2)}(g(\langle x \rangle)) \quad (2.62) \]
A few scenarios were also calculated for this example. Figure 2.5 demonstrates the expected value and standard deviation of the scattering from 8 aligned uncorrelated parallel wires and Figure 2.6 presents the same for 12 aligned wires. As in the line charge example, it can be seen again that the new approach agrees very well with the Monte Carlo reference (again using \(10^6\) samples). Common deterministic inputs for these cases are: wavelength \(\lambda = \pi\), incident angles \(\theta_i = 90^\circ\) and \(\phi_i = 0^\circ\). In this case the uncertain parameter was chosen to be the \(y\)-placement of each of the wires with \(x_n = 0\) and \(\langle y_n \rangle = 0\). In Figures 2.5 and 2.6, these are assumed independent and given by

\[
C(y_n, y_m) = \begin{cases} 
0 & m \neq n \\
\Delta y_n^2 & m = n
\end{cases}
\]  

(2.63)

where \(\Delta y_n = s/(n + 10)\). Figure 2.7 demonstrates another case where the wire locations are in fact correlated. For this result, all parameters are chosen the same as for the previous cases except for the covariance matrix, which was chosen as

\[
C = \begin{bmatrix}
\Delta y_0^2 & 0.85\Delta y_0\Delta y_1 & 0.86\Delta y_0\Delta y_2 \\
0.85\Delta y_1\Delta y_0 & \Delta y_1^2 & 0.87\Delta y_1\Delta y_2 \\
0.86\Delta y_2\Delta y_0 & 0.87\Delta y_2\Delta y_1 & \Delta y_2^2
\end{bmatrix}
\]  

(2.64)

These correlation values were chosen arbitrarily for example purposes, and the number of wires was limited to three for readability/simplicity.

2.9 Summary

It has been demonstrated that a forward-mode small-perturbation automatic-differentiation (SPAD) method can be applied to determine the consequences of uncertainty in CEM calculations. The present approach offers a few advantages over other “stochastic methods” found in the existing literature. First, it can be applied to any existing object-oriented deterministic CEM code with minimal changes; only
Figure 2.5: Infinite parallel wire scattering: 8 wires.
Figure 2.6: Infinite parallel wire scattering: 12 wires.
3 wires: $<x_n> = 0$, $\Delta x_n = 0$, $<y_n> = ns$, $\Delta y_n = s/(n+10)$, $s = 1/k$, $n = $ wire #

Figure 2.7: Infinite parallel wire scattering: 3 wires with correlation.
modification of variable declarations are needed. This is because operators such as addition and multiplication can be overridden in object-oriented languages. Second, the increased computational complexity is small when the number of terms chosen in the expansion is small, which is feasible when uncertainties of the input parameters are small. Finally, the result of the expansion applied is “separable” in the sense that once the derivatives are calculated for a specific expected value, it is rather fast to recompute it for any change in the probability distribution of the input variable(s) since those two parts of the resulting equation are separate. Stated another way, the probabilistic aspects of the problem have been separated from the core of the approach.

Since Stroud-based collocation techniques are considered to be perhaps the most computationally cost-effective at present, it is worth spending some time to discuss the relative merits of the SPAD technique with respect to this method. Interestingly, Stroud cubature only requires $O(d)$ operations to compute statistical properties of interest for both degree 2 and 3 versions of that theory, where $d$ is the number of random variables. As described in Appendix D, the SPAD approach can also produce results with the same numerical complexity; however, correlations between random variables would not be supported in such a configuration or via the Stroud methodology. At $O(d^2)$ cost (for which all results in this dissertation have been generated), the SPAD methodology in contrast does support correlation between input variables. Another difference is that (as discussed in [6]) collocation methods cannot produce error estimates on the result obtained; whereas such estimation is possible with the SPAD methodology.
CHAPTER 3

Reverse-mode SPAD

3.1 Introduction

The discussion in the previous Chapter was based on a forward-mode automatic-differentiation methodology. As discussed there, this produces a computational complexity in the range \( O(d) \) to \( O(d^2) \) (depending on order and complexity chosen), where \( d \) is the number of input random variables used in the calculation. Depending on the problem, scaling at this complexity may be unacceptable for large \( d \). Interestingly, automatic-differentiation also supports an alternative, reverse-mode approach, which has a computational complexity independent of the number of input random variables [29, 28, 15, 20, 21, 22]; instead, it scales with the number of computational steps in the calculation. Consequently, this feature potentially enables a different regime of problems to be tackled more efficiently, and importantly independent of the number of input random variables.

3.2 SPAD Reverse Mode Methodology

The derivation of this approach proceeds in the same manner as the forward mode SPAD, which has also been published in [38]. It is assumed that a given
output function of interest can be expanded via a multivariate Taylor series about the expected values of the uncertain input variables \( x = \{ x_0, ..., x_d \} \) where \( d \) is the number of random variables. These variables can be correlated, and this relationship is described via covariance coefficients \( C(x_l, x_m) \) between random variables \( l \) and \( m \), for example. This is a \( d \times d \) symmetric matrix. Recall that covariance is related to correlation via

\[
C(x_l, x_m) = \Delta x_l \Delta x_m \rho(x_l, x_m),
\]

(3.1)

where \( \rho(x_l, x_m) \) is the correlation coefficient between \( x_l \) and \( x_m \) and \(-1 \leq \rho(x_l, x_m) \leq 1\). The SP expansion again is

\[
\langle \Phi(x) \rangle \approx \sum_{n_0}^{N_0} \cdots \sum_{n_d}^{N_d} \left[ \prod_{k=0}^{d} \frac{(x_k - \langle x_k \rangle)^{n_k}}{n_k!} \frac{d^{n_k}}{dx_k^{n_k}} \right] \Phi(x) \bigg|_{x = \langle x \rangle}
\]

(3.2)

where \( x = \{ x_0, ..., x_d \} \) are the uncertain input variables, \( A_n = \langle \prod_{k=0}^{d} (x_k - \langle x_k \rangle)^{n_k} \rangle \) are the \( n \)-th order statistical central moments of those random variables and \( B_n = \prod_{k=0}^{d} 1/n_k! \).

For brevity, the discussion that follows will focus mostly on reverse mode aspects that differ from the forward mode approach discussed previously and as in [38], which is a very useful starting point for the foundation of the approach and details on the common aspects.

Just like in the forward mode SPAD, the goal of the reverse more SPAD is to calculate the higher-order derivatives as needed in (3.2)

\[
\left[ \prod_{k=0}^{d} \frac{d^{n_k}}{dx_k^{n_k}} \right] \Phi(x) \bigg|_{x = \langle x \rangle}
\]

(3.3)

The reverse mode operation to find these values involves the use of a derivative “trace.” The first step is to initialize the aspects of the derivative trace, which can
be implemented as a class in object-oriented languages. The *trace* class facilitates the higher-order Jacobian $J$ and Hessian $H$ matrices, which are sparse and store the derivative and cross-derivatives obtained during code evaluation. The Jacobian matrix is populated for all elementary operations, and the Hessian matrix is used only for operations that involve two inputs. An element of the higher-order Jacobian matrix will be denoted by

$$J_{nmo} = \frac{d^{p_m}}{dv_m^{o_m}} v_n(v_m), \quad (3.4)$$

where $n$ is the index of the current (or output) operation, $v_n$ is the function at operation $n$, $m$ is the index of the input operation, $v_m$ is the input variable at index $m$, and $o_m$ is the order of the derivative with respect to operation $m$. An element of the Hessian matrix will be denoted by

$$H_{nml} = \frac{d^2}{dv_m dv_l} v_n(v_m, v_l), \quad (3.5)$$

where $n$ is the index of the output operation, $m$ is the index of one of the input operations, and $l$ is the index of the other input operation. Higher-order Hessian values are unnecessary since no elementary computational operations produce higher-order Hessian values. It happens to be convenient to store the index of the current operation as well as the indices of the code step for each initialized random variable within the *trace* class.

A secondary “value” class is then used for each and every value calculated at each and every step of the code. This class also keeps track of the index of the operation that generated each value (for use in determining indices for the $J$ and $H$ matrices). It also makes use of the program *trace* to store derivatives as necessary (the $J$ and $H$ matrices); although at initialization, no values need to be stored in those matrices;
whereas in the forward-mode, a certain set of derivative values were needed at initialization time. For initializing variables with uncertainty, the covariances between other input random variables can also passed through to be stored for convenience in the program trace class.

The reverse mode AD method is quite well studied at this point. Some of the relevant references on this topic include [22] and [13]. These references also include a discussion on sparse matrix techniques, which are key to getting reasonable performance out of a reverse mode AD implementation.

3.3 Elementary Operations

The primary purpose of the value class is to handle operator overloading in calculations of all of the needed derivative values during each step of the program. The following discussion considers the calculation of these values for most of the elementary operations.

Given inputs with indices $m$ and $l$, in order to perform an addition operation with index $n$, the following values are calculated

\[ v_n(v_m, v_l) = v_m + v_l \quad (3.6) \]

\[ J_{nmo} = \begin{cases} 1 & o_m = 1 \\ 0 & \text{otherwise} \end{cases} \quad (3.7) \]

\[ J_{nlo} = \begin{cases} 1 & o_l = 1 \\ 0 & \text{otherwise} \end{cases} \quad (3.8) \]

Values themselves (e.g. $v_n$) are stored within the value class, while the derivatives $J_{nmo}$ and $J_{nlo}$ are stored within the trace class. This is because these values are transient (i.e., used only at the instant they are needed), whereas the Jacobian values need to be stored until the end and then used to solve for the derivatives of interest.
For subtraction, again given input indices $m$ and $l$, the following values are calculated

$$v_n(v_m, v_l) = v_m - v_l \quad (3.9)$$

$$J_{nmom} = \begin{cases} 1 & o_m = 1 \\ 0 & \text{otherwise} \end{cases} \quad (3.10)$$

$$J_{nmo_l} = \begin{cases} -1 & o_l = 1 \\ 0 & \text{otherwise} \end{cases} \quad (3.11)$$

For multiplication with input indices $m$ and $l$, the following values are calculated

$$v_n(v_m, v_l) = v_m v_l \quad (3.12)$$

$$J_{nmom} = \begin{cases} v_l & o_m = 1 \\ 0 & \text{otherwise} \end{cases} \quad (3.13)$$

$$J_{nmo_l} = \begin{cases} v_m & o_l = 1 \\ 0 & \text{otherwise} \end{cases} \quad (3.14)$$

$$H_{nml} = 1 \quad (3.15)$$

Multiplication is the only elementary operation that has the ability to populate the Hessian matrix, which is due to the product rule for derivatives. For negation, given input index $m$, the following values are calculated

$$v_n(v_m) = -v_m \quad (3.16)$$

$$J_{nmom} = \begin{cases} -1 & o_m = 1 \\ 0 & \text{otherwise} \end{cases} \quad (3.17)$$

For powers, given input index $m$ and power $p$, the following values are calculated

$$v_n(v_m) = v_m^p \quad (3.18)$$

$$J_{nmom} = \begin{cases} \frac{v_n}{(p - n)J_{nm(o_m-1)}/v_m} & o_m = 0 \\ (p - n)J_{nm(o_m-1)}/v_m & \text{otherwise} \end{cases} \quad (3.19)$$

For the sine trigonometric function, given input index $m$, the following values are calculated

$$v_n(v_m) = \sin(v_m) \quad (3.20)$$

$$J_{nmom} = \begin{cases} s \cos(v_m) & o_m \text{ odd} \\ s \sin(v_m) & o_m \text{ even} \end{cases} \quad (3.21)$$
where $s$ is a sign that alternates every second power of $o_m$ starting at $o_m = 2$. For cosine, similarly

\begin{align*}
    v_n(v_m) &= \cos(v_m) \\
    J_{nmo_m} &= \begin{cases} 
        s\sin(v_m) & o_m \text{ odd} \\
        s\cos(v_m) & o_m \text{ even}
    \end{cases}
\end{align*}

with $s$ is a sign that alternates every second power of $o_m$ starting at $o_m = 1$. For exponentiation,

\begin{align*}
    v_n(v_m) &= e^{v_m} \\
    J_{nmo_m} &= v_n \quad \text{for all } o_m
\end{align*}

It should be clear that various other basic operations can be obtained by combining the above set of elementary operations. For example, division can be implemented via the power operation with $p = -1$ for the divisor. In other words, to determine $v_3 = v_1/v_0$, the calculation would be broken down into the following steps

\begin{align*}
    v_2 &= v_0^{-1} \\
    v_3 &= v_1v_2
\end{align*}

and associated Jacobian values filled in via the power and multiplication processes as described above. Division could of course be implemented as an elementary operation, but since that produces higher-order Hessian values, but that would require a more involved approach than the one prescribed below. So, it is better to handle it in this way. Another typical operation of interest is the square root. That function can again be implemented via the power operation with $p = 0.5$. In other words, to calculate $v_1 = \sqrt{v_0}$ the following would be used

\begin{align*}
    v_1 &= v_0^{0.5}
\end{align*}
with Jacobian values populated via the power operation as described previously.

There may also be instances of special functions within the calculation of interest. For those scenarios, the same process can be followed to obtain the derivatives of those special functions. For the results presented in this dissertation, there is a brief diversion to discuss the specific special functions when it is encountered in the calculation.

The above operations store a set of derivatives, but they are not actually the output of interest that is needed in order to perform the proper SPAD statistical analysis. In order to determine those values, the Jacobian and Hessian matrices obtained via the above process need to be solved. This process will be shown first via a derivation followed by a discussion of sparse matrix techniques to solve the resulting equation.

First for the derivative, given input of interest \( v_m \) at step \( m \) in the calculation, the progression of variable dependencies at subsequent steps are

\[
\begin{align*}
v_m & \\
v_{m+1}(v_m) & \\
v_{m+2}(v_m, v_{m+1}(v_m)) & \\
v_{m+3}(v_m, v_{m+1}(v_m), v_{m+2}(v_m, v_{m+1}(v_m)))
\end{align*}
\]
and so on. Applying standard calculus rules, the derivatives with respect to \(v_m\) at each subsequent step are

\[
\frac{dv_m}{dv_m} = 1 \quad (3.33)
\]

\[
\frac{v_{m+1}(v_m)}{dv_m} = \text{an already computed value} \quad (3.34)
\]

\[
\frac{v_{m+2}(v_m, v_{m+1}(v_m))}{dv_m} = \frac{dv_{m+2}(v_m)}{dv_m} + \frac{dv_{m+2}(v_{m+1})}{dv_{m+1}} \frac{v_{m+1}(v_m)}{dv_m}
\]

\[
(3.35)
\]

\[
\frac{v_{m+3}(v_m, v_{m+1}(v_m), v_{m+2}(v_m, v_{m+1}(v_m)))}{dv_m} = \frac{dv_{m+3}(v_m)}{dv_m} + \frac{dv_{m+3}(v_{m+1})}{dv_{m+1}} \frac{v_{m+1}(v_m)}{dv_m}
\]

\[+
\frac{dv_{m+3}(v_{m+2})}{dv_{m+2}} \frac{v_{m+2}(v_m, v_{m+1}(v_m))}{dv_m}
\]

\[
(3.36)
\]

As can be seen, this is a process that builds upon the previously determined derivative values. Via inspection, the derivative with respect to \(v_m\) for any subsequent step (denoted as \(n\)), can be found via the following recursive function

\[
\frac{dv_n(v_m, \ldots, v_{n-1})}{dv_m} = \sum_{k=m}^{n-1} \frac{dv_n(v_k)}{dv_k} \frac{dv_k(v_m, \ldots, v_{k-1})}{dv_m}
\]

\[
(3.37)
\]

or in terms of the Jacobian notation

\[
\frac{dv_n(v_m, \ldots, v_{n-1})}{dv_m} = \sum_{k=m}^{n-1} J_{nk1} \frac{dv_k(v_m, \ldots, v_{k-1})}{dv_m}
\]

\[
(3.38)
\]

This turns out to be the solution to the following upper-triangular matrix equation

\[
\begin{bmatrix}
-1 & J_{n(n-1)1} & \cdots & \cdots & J_{nn1} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
-1 & J_{(n+2)(n+1)1} & J_{(n+2)1m1} & \cdots & \cdots \\
\end{bmatrix}
\begin{bmatrix}
\frac{dv_n}{dv_m} \\
\vdots \\
\frac{dv_{m+2}}{dv_m} \\
\frac{dv_{m+1}}{dv_m} \\
\frac{dv_m}{dv_m}
\end{bmatrix}
= \begin{bmatrix}
0 \\
\vdots \\
0 \\
0 \\
-1
\end{bmatrix}
\]

\[
(3.39)
\]

Thus, given the Jacobian values for the sequence of operations throughout the calculation chain, the above matrix can be populated and solved to determine the output.
derivatives of interest: the $dv/dv_m$ vector. Since the Jacobian matrix is a sparse matrix with only $O(n - m)$ nonzero elements, fast sparse techniques can be used to solve this system.

Further higher-order derivatives can also be derived by applying elementary calculus rules to the above system. For example, in order to obtain a second-order derivative with respect to variable $v_l$ obtained at calculation step $l$, the system becomes

$$
\begin{bmatrix}
0 & J_{n(n-1)}^{(2)} & \cdots & \cdots & J_{nm}^{(2)} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & J_{(m+2)(m+1)}^{(2)} & \cdots & J_{(m+2)m}^{(2)} & 0 \\
\end{bmatrix}
\begin{bmatrix}
d^2v_n/dv_mdv_l \\
\vdots \\
d^2v_{m+2}/dv_mdv_l \\
d^2v_{m+1}/dv_mdv_l \\
d^2v_m/dv_mdv_l \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
\vdots \\
0 \\
0 \\
0 \\
\end{bmatrix}
$$

(3.40)

where the $J_{nm}^{(2)}$ are obtained by solving their own systems, which are given by

$$
\begin{bmatrix}
-1 & J_{n(n-1)} & \cdots & \cdots & J_{nl2} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
-1 & J_{(l+2)(l+1)} & J_{(l+2)l} & J_{(l+1)2} & -1 \\
\end{bmatrix}
\begin{bmatrix}
J_{nl}^{(2)} \\
\vdots \\
J_{(l+2)l}^{(2)} \\
J_{(l+1)l}^{(2)} \\
J_{ll}^{(2)} \\
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
\vdots \\
0 \\
0 \\
-1 \\
\end{bmatrix}
$$

(3.41)

Note also that when the row index above is equal to $m$, the Hessian value is used instead of the higher-order Jacobian. This process can be done ad infinitum up to any derivative order of interest. There is however an ever increasing computational cost to do so.

3.4 Walk-through Example

Perhaps the best way to understand any AD process is via example, so a simple analytic problem will be considered next to demonstrate the process. Although, as stated previously, AD is generally applicable, and not just to analytical examples.
Take the following simple function

\[ \phi(x) = (x + x^3)(x + \sin(x)) \]  \hspace{1cm} (3.42)

which is intended to be evaluated at \( x = 5.0 \). The operations are chosen (with any arbitrary order) and evaluated

\[
\begin{align*}
v_0 & \equiv x \quad = 5.0 \quad \hspace{1cm} (3.43) \\
v_1 & = v_0^3 \quad = 125.0 \quad \hspace{1cm} (3.44) \\
v_2 & = \sin(v_0) \quad = -0.9589 \quad \hspace{1cm} (3.45) \\
v_3 & = v_0 + v_1 \quad = 130.0 \quad \hspace{1cm} (3.46) \\
v_4 & = v_0 + v_2 \quad = 4.0411 \quad \hspace{1cm} (3.47) \\
v_5 & = v_3v_4 \quad = 525.3398 \quad \hspace{1cm} (3.48)
\end{align*}
\]

where \( \phi(x) \equiv v_5 \). Concurrently, the Jacobian and Hessian values are populated (this is automatically done in the background in an object-oriented code via operator overloading). Choosing a maximum order of two for this example, the non-zero values
are

\begin{align*}
J_{101} &= 3v_0^2 = 75.0 \\
J_{102} &= 6v_0 = 30.0 \\
J_{201} &= \cos(v_0) = 0.2836 \\
J_{202} &= -\sin(v_0) = 0.9589 \\
J_{301} &= 1 = 1.0 \\
J_{311} &= 1 = 1.0 \\
J_{401} &= 1 = 1.0 \\
J_{421} &= 1 = 1.0 \\
J_{531} &= v_4 = 4.0411 \\
J_{541} &= v_3 = 130.0 \quad (3.57)
\end{align*}

\begin{align*}
H_{543} &= 1 = 1.0 \quad (3.59)
\end{align*}

with all other \( J \) and \( H \) values being zero. There is only one non-zero Hessian value because there is only one multiplication in this example (since only multiplications produce Hessian values).

Finally, the system is solved; for five operations, this is rather straightforward and can be written analytically in a somewhat compact form. Choosing the derivative of interest to be the one with respect to \( v_0 \equiv x \), the following are the derivatives at each
computational step

\[
\frac{dv_1}{dv_0} = J_{101} \quad (3.60)
\]

\[
\frac{dv_2}{dv_0} = J_{201} + J_{211} \frac{dv_1}{dv_0} \quad (3.61)
\]

\[
\frac{dv_3}{dv_0} = J_{301} + J_{311} \frac{dv_1}{dv_0} + J_{321} \frac{dv_2}{dv_0} \quad (3.62)
\]

\[
\frac{dv_4}{dv_0} = J_{401} + J_{411} \frac{dv_1}{dv_0} + J_{421} \frac{dv_2}{dv_0} + J_{431} \frac{dv_3}{dv_0} \quad (3.63)
\]

\[
\frac{dv_5}{dv_0} = J_{501} + J_{511} \frac{dv_1}{dv_0} + J_{521} \frac{dv_2}{dv_0} + J_{531} \frac{dv_3}{dv_0} + J_{541} \frac{dv_4}{dv_0} \quad (3.64)
\]

Thus to find \( \frac{dφ(x)}{dx} \equiv \frac{dv_5}{dv_0} \), all of the above equations are evaluated. In this case, this produces \( \frac{dφ(x)}{dx} = 473.9978 \). The higher-order process can also be followed to obtain, for example, \( \frac{d^2φ(x)}{dx^2} = \frac{dv_5}{dv_0} = 441.0091 \) for this case.

### 3.5 Computing Uncertainties

Given the above AD process, the derivatives of the output of interest can now be obtained. The expected value of this quantity can be determined simply by substituting the AD derivative values into (3.2).

Higher-order moments are also obtained by the same equation, but with a bit more processing. As an example, the standard deviation can be computed by defining \( Λ(x) = [Φ(x)]^2 \), calculating the higher-order derivatives of \( Λ(x) \) via the routines as described previously, and using those values to determine the average \( \langle Λ(x) \rangle \) (3.2). Finally the standard deviation for function \( Φ(x) \) can be calculated from

\[
ΔΦ(x) = \sqrt{\langle Λ(x) \rangle - \langle Φ(x) \rangle^2} \quad (3.65)
\]

The same general idea can be applied to obtain any higher-order statistical moment of interest.
3.6 Induced Line Charge Example

A canonical example involving uncertainties induced in an electrostatic line charge will now be discussed. This basic scenario was also discussed in Chapter 2 and the foundations will not be repeated here.

The uncertain parameters are the radii \( a_m \) of the \( N \) segments comprising the line (Figure 2.7). The expected value of the radii are all chosen as \( \langle a_m \rangle = 1 \text{ mm} \), and the standard deviations are chosen as \( \Delta a_m = \langle a_m \rangle / (10 + m) \). The radii are uncorrelated with

\[
C(a_m, a_n) = \left\{ \begin{array}{ll}
0 & m \neq n \\
\Delta a_m^2 & m = n
\end{array} \right. \quad (3.66)
\]

The reverse mode implementation was utilized within an existing deterministic code, with the minor initialization modifications as described above. Note that this particular example utilizes the natural logarithm in some of its operations. To compute the reverse mode values for this operation, the following are the values and Jacobian elements needed or the reverse mode

\[
v_n = \ln(v_m) \quad (3.67)
\]

\[
J_{nm0m} = \left\{ \begin{array}{ll}
1/v_m & o_m = 1 \\
-(1 + n)J_{nm(o_m-1)}/v_m & \text{otherwise}
\end{array} \right. \quad (3.68)
\]

Results for two scenarios for this example were calculated. In Figure 3.1, the line is divided into 5 segments and in Figure 3.2, the line is divided into 8 segments. As can be seen, the new method agrees very well with a Monte Carlo reference result using \( 10^6 \) sample points.
Figure 3.1: Induced line charge uncertainty: 5 segments
Figure 3.2: Induced line charge uncertainty: 8 segments
3.7 Infinite Parallel Wire Example

Next, the scattering from a set of infinite wires is computed where the positions of each wire are uncertain by a small amount (Figure 2.8). In this case the $y$-coordinate of the wire placements are the uncertain inputs. This particular problem requires Hankel function derivatives to populate some of the Jacobian values. Those are given by the following parameters

\[
v_n = H_k^{(2)}(v_m)
\]

\[
J_{nmo,m} = \frac{1}{2^n} \sum_{l=0}^{n} (-1)^k \binom{n}{l} H_{k+2l-n}(v_m)
\]

The reverse-mode AD method was utilized by augmenting an existing deterministic infinite wire scattering code. More details of that code are discussed in the previous Chapter. A particular problem with three wires with uncertainty in $y$ placement were chosen as an example. The parameters for this problem are $\langle x_n \rangle = 0$, $\Delta x_n = 0$, $\langle y_n \rangle = ns$, $\Delta y_n = s/(n + 10)$, $s = 1/k$, and $\phi_i = 0^\circ$ where $n$ is the wire number. Results are shown in Figure 3.3. It can be seen that the SPAD method again agrees well with a Monte Carlo reference using $10^6$ sample points. These results were produced using a second-order expansion, and some differences can be seen in the output uncertainty at angles closer to grazing incidence. It is expected that such differences would be reduced if a higher-order expansion were used.

3.8 Summary

It has been demonstrated that a reverse-mode automatic-differentiation method is useful for calculating uncertainties via the SPAD methodology. There are a number of trade-offs between the reverse-mode SPAD implemented here, the forward-mode
3 wires: $<x_n> = 0, \Delta x_n = 0$, $<y_n> = ns, \Delta y_n = s/(n+10)$, $s = 1/k$, $n =$ wire #

Figure 3.3: Scattering from 3 infinite wires with placement uncertainty
SPAD, and the various other techniques for evaluating uncertainties. In terms of reverse versus forward-mode SPAD implementations, the computational complexities behave quite different. The base complexity of the forward mode scales as $O(d^N)$ where $d$ is the number of input random variables, and $N$ is the order of cross-derivatives kept for the approximation. On the other hand, the reverse-mode scales with respect to the number of steps $c$ in a given calculation chain, or as $O(c^M)$ with $M$ being the number of terms in the small-perturbation (Taylor series) expansion. This scaling is achieved via sparse matrix techniques. Practically speaking, calculations of interest can have very large number of steps, so $c$ is potentially very large and needs to be stored; thus memory and hard disk space are important considerations. However, in cases where the values can be kept in memory, the $O(c^M)$ scaling obtained via sparse matrix techniques becomes attractive.
CHAPTER 4

Applying SPAD to Particle Scattering Problems

The Discrete Dipole Approximation (or DDA) is a computational approach that can be used to numerically compute scattering from a set of point-like (dipole) particles. In this approach, the geometrical aspect of a problem is represented as a set of unknown point dipole moments, which can be calculated by solving a matrix equation. This method has some limitations, and the generalized method-of-moments is considered more robust for general problems since surfaces and/or volumes can be discretized into fundamental element shapes, which can be used to more accurately represent the geometry of interest. It is however of use in certain classes of problems such as particle scattering, and importantly is another use-case for the SPAD methodology.

4.1 Methodology

The DDA methodology is discussed in detail in [39]. The development of the method is somewhat recent; being first published in 1993. The method-of-moments preceded the DDA by many decades, and its origination culminated from a rethinking of the bases for method-of-moments problems.
The particular problem of interest in this discussion is that of two nearby spheres (Figure 4.1) where coupling may or may not be a significant concern \[40\],[41]. The DDA approach is a good fit for this particular problem. In this particular configuration, one of the spheres is located at \( \mathbf{r}_1 = 0 \) and the other at \( \mathbf{r}_2 = \hat{x}d \) where \( d \) is an uncertain separation distance and given by a uniform probability distribution with limits \( 0 < d < d_0 \). This configuration has an expected value of \( \langle d \rangle = d_0/2 \) and standard deviation \( \Delta d = \langle d \rangle / \sqrt{3} \). The deterministic characteristics of this problem include a \( \hat{z} \)-directed incident plane wave propagating in the \( \hat{x} \) direction (\( \mathbf{E}^i = \hat{x}e^{ikz} \)).

The spheres have a permittivity of \( \epsilon = 3 + i0.1 \) and Rayleigh scattering parameter of \( \alpha = \frac{3\epsilon_0(\epsilon - 1)}{\epsilon + 2} \frac{4\pi a^4}{3} \) where \( a = 0.001m \) is the radius of the spheres and the frequency is \( f = 1 \) GHz. The quantities of interest are the induced dipole moments on each of the spheres and the backscattered far-field radar cross-section (RCS).

The mathematical approach for DDA begins by expressing a total field as the incident field \( \mathbf{E}^i \) plus the contributions from the dipoles \( \mathbf{E}^d_k \)

\[
\mathbf{E} = \mathbf{E}^i + \sum_k \mathbf{E}^d_k \tag{4.1}
\]

where \( k \) is the dipole number. The dipole fields are radiated by current \( \mathbf{J}_k \), so rewriting the above equation

\[
\mathbf{E} = \mathbf{E}^i - \sum_k A_k \mathbf{J}_k \tag{4.2}
\]
where $A_k$ is matrix for the above stated problem is given by

$$A_k = \frac{e^{ikd}}{4\pi \epsilon d} \left[ \begin{array}{cccc} 2\frac{ikd-1}{d^2} & 0 & 0 & \frac{2ikd-1}{d^2} \\ 0 & -k^2 - \frac{ikd-1}{d^2} & 0 & 0 \\ 0 & 0 & -k^2 - \frac{ikd-1}{d^2} & 0 \\ \frac{2ikd-1}{d^2} & 0 & 0 & -k^2 - \frac{ikd-1}{d^2} \end{array} \right]$$

(4.3)

Rewriting the polarizability of the electric fields $P_k = \alpha_k E_k$, the equation of interest becomes

$$\frac{P_j}{\alpha_j} + \sum_{k \neq j} A_k P_k = E^i_j$$

(4.4)

Combining the two terms above, the equation becomes

$$ZP = E^i$$

(4.5)

where the $Z$ matrix from the two-particle problem is

$$Z = \frac{e^{ikd}}{4\pi \epsilon d} \left[ \begin{array}{cccc} 1/\alpha_1 & 0 & 0 & \frac{2ikd-1}{d^2} \\ 0 & 1/\alpha_1 & 0 & 0 \\ 0 & 0 & 1/\alpha_1 & 0 \\ \frac{2ikd-1}{d^2} & 0 & 0 & 1/\alpha_1 \\ 0 & -k^2 - \frac{ikd-1}{d^2} & 0 & 0 \\ 0 & 0 & -k^2 - \frac{ikd-1}{d^2} & 0 \\ 0 & 0 & 0 & 1/\alpha_2 \\ 0 & 0 & 0 & 1/\alpha_2 \end{array} \right]$$

(4.6)

The above set of equations can be analytically solved for the two-particle problem and the solutions are

$$P_1z = \alpha_1 \frac{1 + \alpha_2 \beta e^{ikd}}{1 - \alpha_1 \alpha_2 \beta}$$

(4.7)

$$P_2z = \alpha_2 \frac{e^{ikd} + \alpha_1 \beta}{1 - \alpha_1 \alpha_2 \beta}$$

(4.8)

where

$$\beta = \frac{e^{ikd}}{4\pi \epsilon d} \left( k^2 + \frac{ikd-1}{d^2} \right)$$

(4.10)

Those results are calculated via Monte Carlo and used as references, but the goal is to solve the general problem, which supports larger numbers of particles. That involves
the application of matrix solution techniques such as Gaussian elimination to obtain

\[ P = Z^{-1}E^i \]  
(4.11)

4.2 Results

Two approaches were applied to this particular problem, and results are presented in Figures 4.2 and 4.3. One approach involves a Monte Carlo method applied to (4.7) using 10,000 random samples from the uniformly-distributed input variable \( d \) (labeled as “monte carlo” in the figures). The other approach is using the SPAD methodology (labeled as “taylor approach” in the figures) and solving the matrix equation (4.11). As a starting point, the uncertainty was chosen as a uniform distribution from 0 to \( d_0 \). As can be seen, the SPAD method is not particularly suited to that particular configuration (Figures 4.2 and 4.3) due to the fact that this regime covers cases where the distances between the spheres are very small, so \( 1/d \) is quite large; thus violating the limitations of the SPAD methodology.

The above case demonstrates a break-down of the methodology, but for other more valid parameter choices, it is entirely possible to get good results. As an alternative (valid SPAD) example, the same problem and parameters were chosen, but with a smaller uncertainty in \( d \) (so that values very close to \( d = 0 \) are avoided). A particular example for demonstration purposes is a case with \( \Delta d = \langle d \rangle /3 \) instead of \( \Delta d = \langle d \rangle /\sqrt{3} \). The results for this example are presented in Figures 4.4 and 4.5. As can be seen, when care is taken to avoid the known fundamental limitations of the SPAD methodology, results generated using this technique can be very good.
Backscattered RCS: $<d> = d_0/2$, $\Delta d = <d>/\sqrt{3}$, uniform PDF

Figure 4.2: DDA backscatter with uncertainty that violates SPAD conditions
Figure 4.3: DDA dipole moments with uncertainty that violates SPAD conditions
Backscattered RCS: $<d> = d_0/2$, $\Delta d = <d>/3$, uniform PDF

![Graph showing DDA backscatter reduced (valid SPAD) uncertainty]

**Figure 4.4:** DDA backscatter reduced (valid SPAD) uncertainty

Discussion of other potential issues with respect to this particular problem (such as error quantification and support for “fat tail” distributions) are included in Chapter 6 and Appendix B.
Dipole moments: $<d> = \frac{d_0}{2}$, $\Delta d = \frac{<d>}{3}$, uniform PDF

Figure 4.5: DDA dipole moments reduced (valid SPAD) uncertainty
CHAPTER 5

Applying SPAD to Rough Surface Scattering Problems

5.1 Introduction

A problem very much related to uncertainty is that of scattering by random media such as rough surfaces, where the height of the surface at all of its many points are randomly varying. Since this involves random (or equivalently uncertain) processes, the SPAD methodology can also be applied to these types of problems. Having said that, the intent of this Chapter is intended primarily for validation purposes. The problems considered here are simple configurations (such as the zero-mean rough surface) already have efficient analytical solutions and it would be far more practical to take advantage of those models rather than SPAD itself. However, the SPAD method becomes more appealing for arbitrary surfaces with (arbitrary) roughness or (doubly) curved surfaces where analytical solutions can become very involved.

SPAD also becomes attractive for very high-order solutions, which require more and more laborious derivation for each further order. Plus the higher-order terms usually require a numerical implementation anyway to solve the resulting integrations. SPAD simplifies this process since a single code can be written to support any order of interest. Even though these higher-order SPAD calculations will be computationally
expensive, the man-power is much reduced as there is only one SPAD code that is capable of supporting any expansion order. Also as computers become more powerful, the brute force nature of the SPAD methodology becomes more attractive as it can take advantage of future computing power.

The canonical rough surface scattering problem is a half-space separated by a boundary with a randomly varying surface height. Figure 5.1 demonstrates an example instance of such a random surface separating two half-spaces. There are two classical analytical methods often used to solve such problems: the Small Perturbation Method (SPM) and Physical Optics (PO or KA for Kirchoff Approach). There are also newer analytical methods such as the Small-Slope Approximation [42],[43],[44] and the Elfouhaily model [45]. The following discussion includes solutions for PEC surfaces only. There are solutions for dielectric surfaces, but those will not be discussed here for simplicity.

5.2 Small Perturbation Method for Rough Surfaces

The Small Perturbation Method, or SPM, is one of the classical solutions to rough surface scattering problems. This method was originally derived and proposed in the 1950s and 1960s for remote sensing problems [46],[47], such as ocean surface monitoring. Importantly, the SPM method uses a Taylor series expansion as a part of its solution just like SPAD; although the process itself is quite different. A discussion of
the approach to determine a first-order SPM solution follows. Higher-order solutions are discussed in various references such as [46], [48], and others. This section is a brief overview and derivation of the SPM for informative purposes; there are no new developments in this section.

To start, the electric field for a plane wave impinging on the random surface from the top half-space is written as

\[ E_{yi}(x,y,z) = e^{ik_{xi}x}e^{-ik_{zi}z} \] (5.1)

which is normalized so that \(|E_{yi}|^2 = 1\) for convenience and the incident wave vector components are given by

\[ k_{xi} = k \sin(\theta_i) \] (5.2)
\[ k_{zi} = k \cos(\theta_i) \] (5.3)

where \(\theta_i\) is the incident elevation direction. The surface and its roughness are assumed rotationally invariant, so the incident azimuthal direction \(\phi_i\) does not come into play. \(z = z(x,y)\) are the heights of the rough surface itself, which will vary based on the roughness profile at various \(x\) and \(y\) locations. An image of the coordinate system is shown in Figure 5.2.

There will be a specular field reflected by the surface \(E_{yi}^r\), which is known

\[ E_{yi}^r(x,y,z) = e^{ik_{xi}x}e^{ik_{zi}z} \] (5.4)

There will also be a diffuse scattered field, which will be found via some analytical manipulation. To start off, a Fourier transform will be used to expand all components
Figure 5.2: Coordinate system

of the electric field

\[ E_x^s = \frac{1}{2\pi} \int dk_x \int dk_y E_x^s(k_x, k_y)e^{ik_x x}e^{ik_y y}e^{ik_z z} \]  
\[ (5.5) \]

\[ E_y^s = \frac{1}{2\pi} \int dk_x \int dk_y E_y^s(k_x, k_y)e^{ik_x x}e^{ik_y y}e^{ik_z z} \]  
\[ (5.6) \]

\[ E_z^s = \frac{1}{2\pi} \int dk_x \int dk_y E_z^s(k_x, k_y)e^{ik_x x}e^{ik_y y}e^{ik_z z} \]  
\[ (5.7) \]

Note that \( k_z \) above needs to satisfy

\[ k_z = \sqrt{k^2 - k_x^2 - k_y^2} \]  
\[ (5.8) \]

Applying the Taylor series (i.e. small-perturbation expansion) to \( e^{ik_z z} \),

\[ e^{ik_z z} = 1 + ik_z z - k_z^2 z^2 + ... \]  
\[ (5.9) \]

and looking only at the first non-zero terms (taking into account the cancellation of leading 1 in the resulting total field), the first-order SPM incident field (5.1) and
The reflected field (5.4) are

\[ E_i^y \approx e^{ikzi}(1 - ikz) \]  
\[ E_r^y \approx e^{ikzi}(-1 - ikz) \]  
\[ E_i^r + E_r^r \approx -2ikz \]  

Inserting the first term of (5.9) (i.e.  \( e^{ikz} \approx 1 \)), the first-order scattered fields can be written as

\[ E_s^x \approx \frac{1}{2\pi} \int dk_x \int dk_y E_x^{(1)}(k_x, k_y) e^{ik_x x} e^{ik_y y} \]  
\[ E_s^y \approx \frac{1}{2\pi} \int dk_x \int dk_y E_y^{(1)}(k_x, k_y) e^{ik_x x} e^{ik_y y} \]  
\[ E_s^z \approx \frac{1}{2\pi} \int dk_x \int dk_y E_z^{(1)}(k_x, k_y) e^{ik_x x} e^{ik_y y} \]  

where  \( E^{(1)} \) are the first-order field corrections to be determined. Next, via the divergence equation \( \nabla \cdot \mathbf{E} = 0 \), an initial restriction on the so-far unknown coefficients is

\[ k_x E_x^{(1)}(k_x, k_y) + k_y E_y^{(1)}(k_x, k_y) + k_z E_z^{(1)}(k_x, k_y) = 0 \]  

The surface normal is given by

\[ \hat{n} = \frac{\hat{x} n_x + \hat{y} n_y + \hat{z} n_z}{\sqrt{n_x^2 + n_y^2 + n_z^2}} \]  

where

\[ n_x = \frac{-dz(x, y)}{dx} \]  
\[ n_y = \frac{-dz(x, y)}{dy} \]  
\[ n_z = \frac{dz(x, y)}{dz} = 1 \]  

Next, (5.13) can be inserted into one of the tangential boundary conditions

\[ E_i^x(x, y, 0) + E_r^x(x, y, 0) + E_s^x(x, y, 0) = 0 \]
which gives \( E_x^{(1)} = 0 \) since both \( E_x^i \) and \( E_x^r \) are zero. The second coefficient can be found via the other tangential boundary condition

\[
E_y^i(x, y, 0) + E_y^r(x, y, 0) + E_y^s(x, y, 0) = 0 \tag{5.22}
\]
or

\[
E_y^s(x, y, 0) = -2ikzi(x, y) \tag{5.23}
\]

Expanding \( E_y^s \) via (5.13) and the surface profile \( z(x, y) \) via

\[
z(x, y) = \frac{1}{2\pi} \int dk_x \int dk_y P(k_x, k_y) e^{ik_x x} e^{ik_y y} \tag{5.24}
\]

where \( P(k_x, k_y) \) is the correlation function of the random surface, which relates correlation strength between various points on the surface, it can be seen that

\[
\frac{1}{2\pi} \int dk_x \int dk_y E_y^{(1)}(k_x, k_y) e^{ik_x x} e^{ik_y y} = -2ikzi \frac{1}{2\pi} \int dk_x \int dk_y z(k_x, k_y) e^{ik_x x} e^{ik_y y} \tag{5.25}
\]

producing the next coefficient as

\[
E_y^{(1)}(k_x, k_y) = -2ikzi P(k_x, k_y) e^{ik_x x} \tag{5.26}
\]

finally, the third unknown term can be determined from (5.16) as

\[
E_z^{(1)} = \frac{1}{k_z} \left[ k_x E_x^{(1)} + k_y E_y^{(1)} \right] \tag{5.27}
\]

\[
E_z^{(1)} = \frac{2ikzi k_y}{k_z} P(k_x, k_y) \tag{5.28}
\]

Equations for the scattered fields can now be written. For example

\[
E^s_{\phi} = \int dx' \int dy' \left[ E_x^s(x', y') \sin(\phi_s) \cos(\theta_s) + E_y^s(x', y') \cos(\phi_s) \cos(\theta_s) \right] e^{ik_{x'} x'} e^{ik_{y'} y'} \tag{5.29}
\]
where the scattered wavevector numbers are

\[ k_{xs} = k \cos(\phi_s) \sin(\theta_s) \]  
(5.30)

\[ k_{ys} = k \sin(\phi_s) \sin(\theta_s) \]  
(5.31)

Since \( E_x^s = 0 \), (5.29) simplifies to

\[ E_{\phi}^{(1)} = \cos(\phi_s) \cos(\theta_s) \int dx' \int dy' E_y^s(x', y') e^{ik_{xs}x'} e^{ik_{ys}y'} \]  
(5.32)

or

\[ E_{\phi}^{(1)} = \frac{i \cos(\phi_s) \cos(\theta_s) k_{zi}}{\pi} \int dx' \int dy' \int dk_x' \int dk_y' \int dx'' \int dy'' \int dk_x'' \int dk_y'' P(k_x', k_y') e^{i(k_x'-k_x) x} e^{i(k_y'-k_y) y} e^{ik_{xs}(x' - x'')} e^{ik_{ys}(y' - y'')} \]  
(5.33)

The goal finally is to determine the expected value of the radar-cross section, which can be expanded as

\[ \langle \sigma_{\phi\phi}^{(1)} \rangle = 4\pi \langle E_{\phi}^{(1)} E_{\phi}^{(1)\ast} \rangle \]

\[ = \frac{4}{\pi} \cos^2(\phi_s) \cos^2(\theta_s) k_{zi}^2 \int dx' \int dy' \int dk_x' \int dk_y' \int dx'' \int dy'' \int dk_x'' \int dk_y'' \]

\[ \langle P(k_x', k_y') P(k_x'', k_y'') \rangle e^{i(k_x'-k_x') x} e^{i(k_y'-k_y') y} e^{ik_{xs}(x' - x'')} e^{ik_{ys}(y' - y'')} \]  
(5.34)

where the expected value of the correlations is

\[ \langle P(k_x', k_y') P(k_x'', k_y'') \rangle = \frac{\pi^2}{L^2} W(k_{xs} - k_{x}, k_{ys} - k_{y}) \]  
(5.35)

with \( W(k_x, k_y) \) being the power spectrum of the rough surface, which is a fundamental aspect relating points on the surface. Following some final integration rules after inserting the power spectrum above, the expected SPM result can be found as

\[ \langle \sigma_{\phi\phi}^{(1)} \rangle = 4\pi \cos^2(\phi_s) \cos^2(\theta_s) k_{zi}^2 W(k_{xs} - k_{x}, k_{ys} - k_{y}) \]  
(5.36)
The above procedure can also be followed for other polarizations of interest, but will not be reproduced here. The book by Ishimaru [46] as well as [49] are good references on the derivation of solutions for other polarizations. The procedure can also be followed similarly for higher-orders as well, but it turns out that the integrals tend not to simplify so easily and a lot more work needs to be done. This has been studied and published in various articles such as [50].

Note that $\sigma^{(1)}_{\phi\phi}$ is a correction term. The expected value of the specular return is given by

$$\langle \sigma^{(0)}_{\phi\phi} \rangle = 4\pi \langle E^{(0)}_\phi E^{(0)*}_\phi \rangle$$

(5.37)

where

$$E^{(0)}_\phi = E_y r \cos(\phi_s) \cos(\theta_s)$$

(5.38)

giving the expected value of the specular return as

$$\langle \sigma^{(0)}_{\phi\phi} \rangle = 4\pi \cos^2(\phi_s) \cos^2(\theta_s) \langle e^{-ikz\Delta z} \rangle$$

(5.39)

$$\langle \sigma^{(0)}_{\phi\phi} \rangle = 4\pi \cos^2(\phi_s) \cos^2(\theta_s) e^{-k_s^2\Delta z^2/2}$$

(5.40)

### 5.3 Physical Optics Method for Rough Surfaces

The Physical Optics method is the other classical method used for computing scattering from planar interfaces. The full details will not be described here since this approach is not valid in the same regime as the SPAD and SPM methods.

Physical Optics is appropriate for large slowly varying roughnesses. The fundamental approximation is the use of

$$\mathbf{J} = 2\hat{n} \times \mathbf{H}^i$$

(5.41)

to determine the currents $\mathbf{J}$ induced from incident magnetic field $\mathbf{H}^i$. 107
Starting with this approximation and applying the appropriate surface roughness, some mathematical manipulation can be done to obtain the Physical Optics solution, which is given by

\[ \sigma_{\phi\phi} = \frac{k^2}{\pi \cos^2(\theta_i)} D_I \]  

where \( D_I \) depends on the surface spectrum. For a Gaussian spectrum it is

\[ D_I = \pi l^2 e^{k_d^2 h^2} \sum_{m=1}^{\infty} \frac{k_d^{2m} h^{2m}}{m!m} \exp \left[ -\frac{(k_d^2 + k^2)^2}{4m^2} \right] \]

One interesting aspect of \( D_I \) is that there was a small-perturbation expansion involved in its derivation as well. An exponential involving the surface correlation function was expanded via a Taylor series as

\[ e^{k_d^2 h^2 C(\rho)} \approx \sum_{m=0}^{\infty} \frac{k_d^{2m} h^{2m} C(\rho)^m}{m!m} \]

with \( C(\rho) \) being the correlation function of the rough surface. This is an indication that SPAD could also potentially be applied to obtain PO results, but this was not done in this effort.

The above equations are as an additional reference included in comparisons with SPAD results that follow. Complete details of the derivation can be found in references [49] and [46].

5.4 SPAD Method for Rough Surfaces

The previous sections included a discussion of the classical approaches for scattering from randomly rough surfaces. The following section discusses a procedure for applying the new SPAD methodology to the same types of problems.

The procedure for solving such problems is very similar to the infinite wire scattering problem as discussed in Chapters 2 and 3, and was originally described in [49].
The approach described here only supports parallel polarization, but as an extension, a two-dimensional canonical grid method is described in [51], which can be used to compute both parallel and perpendicular polarizations.

Thinking of this solution in terms of a modification to a deterministic code (one of the advantages of the SPAD methodology), the non-statistical code itself would solve the scattering from a flat surface. Once uncertainty is applied (with appropriate correlation relating the surface points) to the surface inputs of that flat surface scattering code, it becomes a rough surface scattering code.

In terms of a derivation, the first step is to represent the rough surface (as well as the equivalent deterministic flat surface) via a collection of infinite wires (Figure 5.4). The goal will be to solve for currents induced along those wires via a method-of-moments approach, and radiate those to determine the surface scattering. Note that the spacing between wires representing the surface must be sufficiently small with respect to the wavelength to sufficiently discretize the surface.

There are some key differences compared to the classical methods. One is that the rough surface should be infinite in length, but the wire representation is limited in extent. This problem is addressed via a tapered incident field that illuminates only a subset of the surface. This tapering is later scaled out of the final solution to obtain a normalized (or per unit area) result. The other difference is that the uncertainties applied to the wires will be very specific and described by the properties of the rough surface itself.

The derivation of this approach proceeds as follows. First of all, the incident field with tapering applied is given by [52]

\[ E_y^i(x) = e^{i(k_x \alpha_x - k_z \alpha_z)(1+w(x))} - f(x) \]  

(5.45)
where \( k_{xi} \) and \( k_{zi} \) were defined in (5.2) and again \( E_i^y \) is normalized for convenience (\(|E_i^y|^2 = 1\)). The tapering functions themselves (\( w \) and \( f \)) are given by

\[
\begin{align*}
  f(x) &= \frac{(x + z(x)\tan(\theta))^2}{g^2} \quad (5.46) \\
  w(x) &= \frac{2f(x) - 1}{g^2k_{zi}^2} \quad (5.47)
\end{align*}
\]

where \( g \) is a selectable parameter that controls the scale of the tapering. Many references suggest \( g = L/4 \) as a good choice where \( L \) is the length of the illuminated part of the surface.

Next, the surface heights are the parameters of interest, which need to be prepared appropriately. For a Gaussian process, the following can be used to initialize the surface parameters. Starting with \( N \) independent Gaussian uncertain parameters \( y_n \) where \( N \) is the number of wires chosen to represent the discretized surface, the
spectral domain surface profile is generated via a transformation of those parameters [52]

\[ F(k_m) = \sqrt{2\pi L W(k_m)} \begin{cases} y_n & m = 0, N/2 \\ y_n + i y_{n+N/2} & \text{otherwise} \end{cases} \]  

(5.48)

where \( W(k_m) \) is the power spectrum of the surface roughness and is the same parameter as encountered in the SPM and PO approaches. Note that in a SPAD implementation the \( y_n \) are the input uncertain parameters and would be initialized using the appropriate class. Given the spectral domain values (5.48), the surface profile itself can be generated via the following series expansion (which can be calculated efficiently via the Fast Fourier Transform at \( O(n \log(n)) \) complexity)

\[ z(x_n) = \frac{1}{L} \sum_{m=-N/2}^{N/2} F(k_m) e^{ik_m x_n} \]  

(5.49)

Again, \( L \) is the length of the illuminated portion of the surface.

The derivatives of the surface will also be needed later in the derivation. Many approaches are available to determine these values, but for simplicity, a finite central-differencing approach is chosen

\[ \frac{dz}{dx} = \frac{z(x_{n+1}) - z(x_{n-1})}{2\Delta x} \]  

(5.50)

where \( \Delta x = x_1 - x_0 \) is the separation between wires (assumed to be uniform).

Next, the elements of the method-of-moments matrices are populated. The elements of the \( Z \) matrix are given by

\[ Z_{nm} = i\Delta x \begin{cases} \frac{H_0^{(1)}(k\rho_{nm})}{1+2i(\log(\gamma k\Delta x \sqrt{1+dz/dx})-1)/\pi} & n \neq m \\ \frac{1}{4} & n = m \end{cases} \]  

(5.51)

where \( \gamma = 1.78107 \) is a constant and the distance vector is given by

\[ \rho_{nm} = \sqrt{(x_n - x_m)^2 + (z(x_n) - z(x_m))^2} \]  

(5.52)
The elements of the incident field vector \( \mathbf{v} \) are

\[
v_n = E_y^i(x_n)
\]

(5.53)

where \( x_n \) are the \( x \) locations of the wires.

At this point, the currents \( \mathbf{p} \) induced on the wires are calculated by solving the matrix equation using Gaussian elimination or any matrix solution technique of interest

\[
\mathbf{p} = \mathbf{Z}^{-1} \mathbf{v}
\]

(5.54)

Finally, the scattered field elements can be determined by radiating all of the induced currents to the far-field and adding all \( N \) of them up

\[
E_y^s = \sum_{n=0}^{N} p_n \Delta x e^{-i(x_n k_z s + z(x_n) k_s)}
\]

(5.55)

and the radar cross-section can be determined as

\[
\sigma = \frac{4\pi |E_y^s|^2}{D}
\]

(5.56)

where \( D \) is a scaling parameter that counteracts the tapering applied to the incident field and is given by

\[
D = 8\pi k_z s g \sqrt{\frac{\pi}{2}} \left[ 1 - \frac{1 + 2 \tan^2(\theta)}{2g^2 k_z^2 s^2} \right]
\]

(5.57)

At this point, the output value \( \sigma \) and its derivatives have been obtained via the automatic differentiation procedure as carried out throughout the calculation starting from initial independent input variables \( y_n \). In order to obtain the expected value \( \langle \sigma \rangle \), the SPAD small-perturbation expansion is used, and methods as described in Chapters 2 or 3 are used to obtain the statistical quantities of interest.
An important advantage of the rough surface SPAD methodology is that it can be applied to mixed problems; for example, a static surface vessel on a randomly rough ocean surface or an arbitrarily curved rough surface. Classical solutions cannot support these types of problems.

5.5 Results and Comparisons

Results were computed using codes implementing the above solutions for the SPM, PO (labeled ka for Kirchhoff Approach in figures), and SPAD approaches as well as a Monte Carlo solution (which utilizes the same method-of-moments approach as described above for the SPAD rough surface code). The problem parameters are chosen to be within the valid range for a small-perturbation. So even though Physical Optics results are included, they are not valid since it is the wrong regime for that method.

The first comparison (Figure 5.4 is for a surface with a Gaussian spectrum roughness, which is given by [49]

\[ W(k) = \frac{h^2 l}{2 \sqrt{\pi}} \exp \left[ -\frac{k^2 l^2}{4} \right] \]  

(5.58)

where \( k \) is the wavenumber, \( h \) is the expected variance in the height of the surface, and \( l \) is the correlation length of the surface. Note that the unitless quantities \( kh \) and \( kl \) (the electrical length and height) will be used in examples for more generality. Also, an important restriction on the height variance in order for the SPM and SPAD methodologies to be valid is that it must be small with respect to the wavelength or \( kh << 1 \). For the example chosen (Figure 5.4), the parameters chosen were \( kh = 0.1\pi \), \( kl = 0.4\pi \), and incident illumination from elevation angle \( \theta_i = 30^\circ \). 128 input random variables were used to represent the surface in this case.
Figure 5.4: Gaussian spectrum rough surface scattering ($kh = 0.1\pi$, $kl = 0.4\pi$, $\theta_i = 30^\circ$)
The next comparison (Figure 5.5) is for a rough surface with a power-law (or Ocean surface like) spectrum. The power-law spectrum is given by [53]

\[
W(k) = \begin{cases} 
\frac{a_0}{k_l^4} & k_l < k < k_u \\
0 & \text{otherwise}
\end{cases}
\]  

(5.59)

where \(a_0\) is a scaling factor and \(k\) is the wavenumber. A scaling factor of \(a_0 = 6.573e^{-4}\) is often used to mimic the scaling needed to represent the Ocean surface. The \(k_l\) and \(k_u\) limits are imposed due to the singularity as \(k\) approaches zero. For the example chosen, the parameters are \(k_l = k/2\) and \(k_u = 2k\). The small-perturbation restriction is often considered valid for \(k_l > k/8\). This case also used \(N = 128\) wires (128 input random variables) to represent the surface.

The results demonstrate very close agreement between Monte Carlo and SPAD; although there are some subtle differences between SPAD and SPM results near grazing. This is an effect of the tapered illumination. It is well-publicized [52],[54] that the tapered incident field can have some inaccuracies near grazing, and there are some other additions to the tapered field formulation that can correct this effect [55], but this additional complexity was not included in this analysis.

Note that the above analyses used \(N = 128\) random variables to represent the surface. It was found that results obtained using \(N = 64\) random variables were close, but had a kink at certain angles. This was due to an under-discretized surface (an insufficient number of samples per wavelength). 128 random variables was found to be quite stressing on the forward-mode implementation used to produce results for these cases; requiring four days of runtime on a single processor and 32 GB of RAM.
Figure 5.5: Power-law spectrum rough surface scattering ($a_0 = 6.573e^{-4}$, $k_l = k/2$, $k_u = 2k$, $\theta_i = 30^\circ$)
CHAPTER 6

SPAD Numerical Error Analysis

An important question about the SPAD methodology is how well it performs with respect to the perturbation magnitude ($\Delta x / \langle x \rangle$) and the number of terms used in the Taylor series expansion ($N$). Unfortunately, the actual results for any such investigation will be very problem-dependent. Nevertheless, results for specific problems can still be very illustrative. An example chosen for this discussion is a two sphere DDA (Discrete Dipole Approximation) as described in Chapter 4.

The perturbation magnitude as well as the number of terms used in the expansion were varied for a few representative probability distributions. Figure 6.1 illustrates the error with respect to a Monte Carlo reference (using 10,000 samples) for a Gaussian random variable $d$ (the separation distance between two sphere particles), and Figure 6.2 demonstrates the error for a uniform random variable for the same DDA problem. The chosen frequency is $f = 1$GHz. Finally, figure 6.3 demonstrates the error for a log-normal random variable. For the log-normal distribution, the most straightforward way to calculate the central moments (i.e. the $A_n$ values) is by expanding the terms and using the simple formula for the log-normal distribution’s non-central higher-order moments. The first six central moments (for any distribution) can be written
as

\[ A_0 = \langle (y - \langle y \rangle)^0 \rangle = 1 \quad (6.1) \]

\[ A_1 = \langle (y - \langle y \rangle)^1 \rangle = 0 \quad (6.2) \]

\[ A_2 = \langle (y - \langle y \rangle)^2 \rangle = \langle y^2 \rangle - \langle y \rangle^2 \quad (6.3) \]

\[ A_3 = \langle (y - \langle y \rangle)^3 \rangle = \langle y^3 \rangle - 3 \langle y^2 \rangle \langle y \rangle + 2 \langle y \rangle^3 \quad (6.4) \]

\[ A_4 = \langle (y - \langle y \rangle)^4 \rangle = \langle y^4 \rangle - 4 \langle y^3 \rangle \langle y \rangle + 6 \langle y^2 \rangle \langle y \rangle^2 - 3 \langle y \rangle^4 \quad (6.5) \]

\[ A_5 = \langle (y - \langle y \rangle)^5 \rangle = \langle y^5 \rangle - 5 \langle y^4 \rangle \langle y \rangle + 10 \langle y^3 \rangle \langle y \rangle^2 - 10 \langle y^2 \rangle \langle y \rangle + 4 \langle y \rangle^5 \quad (6.6) \]

\[ A_6 = \langle (y - \langle y \rangle)^6 \rangle = \langle y^6 \rangle - 6 \langle y^5 \rangle \langle y \rangle + 15 \langle y^4 \rangle \langle y \rangle^2 - 20 \langle y^3 \rangle \langle y \rangle^3 + 15 \langle y^2 \rangle \langle y \rangle^4 - 5 \langle y \rangle^6 \quad (6.7) \]

where the non-central moments for the log-normal distribution [56] are simply

\[ \langle y^n \rangle = \exp \left[ n \langle x \rangle + \frac{n^2 \Delta x^2}{2} \right] \quad (6.8) \]

where \( \langle x \rangle \) and \( \Delta x \) are the expected value and standard deviation of a Gaussian random variable \( x \). Note that a direct application of the previous sets of equations will result in various destructive floating point issues when implemented in a computer program (due to the large values taken on by the exponential function). In order to overcome this problem, the analytical central moments specifically for the log-normal
distribution should be used instead. These are given by

\[ A_0 = 1 \]  \hspace{1cm} (6.9)
\[ A_1 = 0 \]  \hspace{1cm} (6.10)
\[ A_2 = e^{2(x)} \left[ e^{2\Delta x^2} - e^{\Delta x} \right] \]  \hspace{1cm} (6.11)
\[ A_3 = e^{3(x)} \left[ e^{5\Delta x^2/2} - 3e^{5\Delta x^2/2} + 2e^{3\Delta x^2/2} \right] \]  \hspace{1cm} (6.12)
\[ A_4 = e^{4(x)} \left[ e^{8\Delta x^2} - 4e^{5\Delta x^2} + 6e^{3\Delta x^2} - 3e^{2\Delta x^2} \right] \]  \hspace{1cm} (6.13)
\[ A_5 = e^{5(x)} \left[ e^{25\Delta x^2/2} - 5e^{17\Delta x^2/2} + 10e^{11\Delta x^2/2} - 10e^{7\Delta x^2/2} + 4e^{5\Delta x^2/2} \right] \]  \hspace{1cm} (6.14)
\[ A_6 = e^{6(x)} \left[ e^{18\Delta x^2} - 6e^{13\Delta x^2} + 15e^{9\Delta x^2} - 20e^{6\Delta x^2} + 15e^{4\Delta x^2} - 5e^{3\Delta x^2} \right] \]  \hspace{1cm} (6.15)

Note that the variable \( y \) is being used to denote log-normal variables and \( x \) is being used to denote Gaussian variables in order to illustrate the intimate relationship between the two, which is simply \( y = e^x \). This is a very important property, which will be utilized and discussed in further detail in the next chapter.

As can be seen in Figures 6.1-6.3, the accuracy does improve as more terms are included in the expansion, but not by a significant amount beyond the first four terms (at least for the problem under consideration). It can be seen that for these cases (with an expected value of \( \langle d \rangle = 1 \text{cm} \)) a perturbation magnitude (\( \Delta d/\langle d \rangle \)) of about 0.2 produces good estimates.

On the other hand, Figure 6.4 demonstrates the same analysis for a uniform random variable with expected value \( \langle d \rangle = 1 \text{m} \) (instead of 1cm). As can be seen in this case the displacement magnitude needs to be much smaller in order to achieve similar accuracy to the 1cm case. This case also better demonstrates the ability of the SPAD method to obtain improved accuracy with a larger number of terms in the Taylor series. Ultimately, all of these cases show that the error cannot be described
simply in terms of the perturbation magnitude itself. This has to do with the fact
that the error in this method is fundamentally dependent upon the derivatives of the
output function, and those values will be specific to the problem under evaluation.

As a suggestion, a convergence analysis would be a much better way to determine
the general effectiveness of the SPAD methodology since such an approach would
provide very informative error quantification for the specific problem of interest. In
order to perform this analysis, consider the Taylor series expansion for the expected
value of the function of interest as given by

\[
\langle \phi(x) \rangle \approx \sum_{n=0}^{N} \frac{\langle (x - \langle x \rangle)^n \rangle}{n!} \left. \frac{d^n \phi(x)}{dx^n} \right|_{x=\langle x \rangle} \tag{6.16}
\]

One can calculate the derivatives (via Automatic-Differentiation) for the output func-
tion for any particular problem of interest with \(N\) terms followed by a calculation of
the \((N + 1)\)-th term. Then, given those last two derivatives, the ratio

\[
\frac{\langle (x - \langle x \rangle)^{N+1} \rangle}{(N + 1) \langle (x - \langle x \rangle)^N \rangle} \left. \frac{d^{N+1} \phi(x)}{dx^{N+1}} \right|_{x=\langle x \rangle} \left/ \left. \frac{d^N \phi(x)}{dx^N} \right|_{x=\langle x \rangle} \right) \tag{6.17}
\]

can be examined to determine whether the statistics were sufficiently captured (i.e.
whether this quantity is approaching zero). Depending on the level of fidelity re-
quired, this would enable one to have high confidence in the applicability of the
SPAD methodology for the problem at hand.
Figure 6.1: SPAD error for DDA problem: Gaussian distribution
Figure 6.2: SPAD error for DDA problem: uniform distribution
Figure 6.3: SPAD error for DDA problem: log-normal distribution
Figure 6.4: SPAD error for DDA problem: uniform distribution (1m expected value)
CHAPTER 7

Conclusions

This dissertation proposed a new Small-Perturbation Automatic-Differentiation for computational problems involving uncertainties. The discussion included a background on fundamentals, development of the approach, and application to many problems. As a consequence, SPAD was seen to be useful on a broad set of problems each of which involving uncertain input parameters and/or statistical processes.

Chapter 1 included a background on existing uncertainty methodologies as well as an introduction to SPAD approach itself and to the underlying automatic-differentiation methodology. Also as a background, this Chapter included results and discussion of the numerical complexity of other fundamental and basic uncertainty approaches.

Chapter 2 pursued the development and discussion of the forward-mode SPAD methodology. It was found that this approach offers a few advantages over other stochastic methods. First, it can be applied to any existing object-oriented deterministic CEM code with minimal changes; only modification of variable declarations are needed. This feature is achieved via operator overloading since operations such as addition and multiplication can be overridden in object-oriented languages. It was also found that the increased computational complexity (compared to a deterministic solution) is small when the number of terms chosen in the expansion is small,
which is feasible when uncertainties of the input parameters are small. Finally, it was found the result of the expansion is “separable” in the sense that once the derivatives are calculated for a specific expected value, it is rather fast to recompute it for any change in the probability distribution of the input variable(s) since those two parts of the resulting equation are separate. Stated another way, the probabilistic aspects of the problem have been separated from the core of the approach, which differentiates SPAD from other methods and may be an attractive feature depending the goals of the research at hand. It was also found that SPAD is a valid approach as all results present agree with Monte Carlo references. This Chapter also included a discussion of the numerical complexity of this approach. It was found that it is possible to obtain $O(d)$ complexity, where $d$ is the number of input random variables, using a fourth-order approach when the input variables are uncorrelated. For correlated input variables, an increased $O(d^2)$ is required.

Chapter 3 analysed the development and discussion of a reverse-mode SPAD approach. It was found that there are a number of trade-offs between the reverse-mode SPAD implemented here, the forward-mode SPAD, and the various other techniques for evaluating uncertainties. In terms of reverse versus forward-mode SPAD implementations, the computational complexities behave quite different. The base complexity of the forward mode scales as $O(d^N)$ where $d$ is the number of input random variables, and $N$ depends on the complexity chosen. For uncorrelated variables in a fourth-order approach, $N = 1$, and for correlated variables $N = 2$ is a minimum. On the other hand, the reverse-mode scales with respect to the number of steps $c$ in a given calculation chain, or as $O(c^M)$ where $M$ is the Taylor series expansion order. In order to obtain the expected value of an output $M = 1$ is sufficient, but in order to
obtain an estimate of the standard deviation $M = 2$ or better is needed. This scaling is achieved via sparse matrix techniques. Practically speaking, calculations of interest can have very large number of steps, so $c$ is potentially very large and needs to be stored; thus memory and hard disk space are important considerations. However, in cases where the values can be kept in memory and for low $M$ (preferably $M = 1$), the $O(c^M)$ scaling obtained via sparse matrix techniques becomes attractive.

Chapters 4 and 5 used the SPAD developments laid out in the previous Chapters to analyse two particular regimes where uncertainty is an important consideration. Chapter 4 applied SPAD to a particle scattering problem involving two dipole-like spheres. It was found that SPAD is applicable to these type of problems as long as the small-perturbation limitation is appropriately taken into consideration. Chapter 5 applied SPAD to rough surface scattering problems mostly as a validation exercise, but also as a stepping point to be able to tackle mixed problems (involving both rough and static components).

Chapter 6 included a brief numerical error analysis of the SPAD methodology to better quantify its limitations. Again, SPAD was applied to the particle scattering problem, and the error was studied with respect to the uncertainty as well as number of terms used in the SPAD expansion. For this particular problem, it was seen that for small uncertainty (with respect to the wavelength) the number of SPAD terms needed was minimal (two often being enough). It was also seen that with larger uncertainty more SPAD terms increased the accuracy of the results.

It bears repeating that the primary limitation of the SPAD methodology is that it does (as its name implies) require small perturbations in the uncertain input quantities, which must be taken into consideration when deciding whether SPAD is the
best choice to tackle the problem at hand. Other uncertainty approaches are far more appropriate when the small-perturbation condition is violated. However, as mentioned in the Introduction, most practical scenarios entail small perturbations and the well-known behavior of the Taylor series makes it tractable to understand its limitations very well. Finally, it should be pointed out that the SPAD methodology is not limited to the Method-of-Moments approach as was used for all examples. In principle, it can be applied to any other CEM algorithm as well.

Some future potential avenues of research related to SPAD include full-wave analysis of objects with uncertainties using a 3-D EFIE 3-D Method-of-Moments solver. Also, Appendix B includes some preliminary findings on input distributions with fat tail probability distributions. For the cases analysed, it was found that it was possible to translate a short tail distribution (such as the Gaussian) into one with a fat tail to use as the uncertain input variables. It would be useful to expand on that work. Finally, it would be very important to further study application of the SPAD methodology to problems involving both static and statistical components such as a scattering target located beneath a rough interface. Also, as described in Chapter 5, it may be interesting to apply the SPAD methodology to other rough surface methodologies such as Physical Optics and the Small-Slope Approximation.
APPENDIX A

Computational divided differencing

One very interesting recent publication investigates an application of AD methodology to finite-differencing [57] using a method denoted as “computational divided differencing.” The paper contends that this technique can be viewed as a generalization of computational (i.e. finite) differencing itself. The approach proceeds by recognizing that each new derivative computation can be determined via information about derivatives of the inputs (the same philosophy as that applied in automatic differentiation). The fundamental set of properties for any operation (addition, multiplication, etc.) can be applied subsequently applied to determine derivative values of any following operation.

As an example that demonstrates a strong potential for this approach to be utilized in computational electromagnetics problems, assume that a system has an FDTD update equation given by

\[
a(x_n, t_{m+1}) = a(x_n, t_m) + \frac{1}{k} \left. \frac{da(x, t_m)}{dx} \right|_{x=x_n} \tag{A.1}
\]

which would normally be implemented (for example via central-differencing) as

\[
a(x_n, t_{m+1}) = a(x_n, t_m) + \frac{1}{k} \frac{a(x_{n+1}, t_m) - a(x_{n-1}, t_m)}{x_{n+1} - x_{n-1}} \tag{A.2}
\]
Instead, introducing a computational divided differencing notation (note reference [57] uses a slightly different notation) as

\[
\frac{da(x, t_m)}{dx} \bigg|_{[x_{n+1}, x_{n-1}]} = \frac{a(x_{n+1}, t_m) - a(x_{n-1}, t_m)}{x_{n+1} - x_{n-1}}
\]  

(A.3)

then (A.2) can be written as

\[
a(x_n, t_{m+1}) = a(x_n, t_m) + \frac{1}{k} \frac{da(x, t_0)}{dx} \bigg|_{[x_{n+1}, x_{n-1}]}
\]  

(A.4)

and its derivative can be written as

\[
\frac{da(x, t_{m+1})}{dx} \bigg|_{[x_{n+1}, x_{n-1}]} = a(x_n, t_m) + \frac{1}{k} \frac{d^2 a(x, t_m)}{dx^2} \bigg|_{[x_{n+1}, x_{n-1}]}
\]  

(A.5)

which can be simplified to

\[
\frac{da(x, t_{m+1})}{dx} \bigg|_{[x_{n+1}, x_{n-1}]} = a(x_n, t_m) \left[ 1 + \frac{2}{k} \frac{da(x, t_m)}{dx} \bigg|_{[x_{n+1}, x_{n-1}]} \right]
\]  

(A.6)

after applying the chain rule. Thus, the derivative values of \(a\) for all \(x_n\) at every new time step can be obtained solely from the deterministic value and first derivative of \(a\) at the previous time step. Note that as a starting point, the initial derivative value can be obtained via a standard finite-differencing approach.

It appears that this particular consequence has not yet been noticed or applied in the field of computational electromagnetics, so this may be something of significant interest to study a bit more for the upcoming dissertation. However, as a practical matter, it is important to note that computational divided differencing is protected by United States patent number US2004/0015830 [58]. Unfortunately, this means that anyone interested in implementing this technique will need to negotiate licensing rights, which should be done upfront to avoid expending wasted effort on an implementation that will end up litigated later by the patent holder. However, a federal
appeals court very recently ruled that mental processes do not fall under the scope of patentability (in Cybersource Corporation v. Retail Decisions, Inc. [59]). One could argue that any mathematical algorithm is simply a set of mental processes, and thus this and other pure math patents in play today are invalid; however, the current patent system as it presently exists requires millions of dollars to pursue a challenge. Effectively, the issuance of this patent will likely lead potential implementers (at universities and small/mid-sized corporations without huge patent portfolios) to decide that the wisest course of action is to simply wait it out until 2024 when the patent expires. The current state of the patent system is unfortunately acting as an impediment to academic freedom, scientific openness, and technological advancement. Ideally, the patent system should be reverted to something resembling the initial system implemented by Thomas Jefferson at the start of the American experiment [60] where patentable materials had to be fully justified, non-obvious, and demonstrated to work.
As has been readily observed in many references, certain probability distributions have higher-order central moments that do not converge. These are often described as “fat tail” distributions due to the large probability of values being observed in far away from the expected value (often simply called the tail). A couple examples of “fat tails” include the log-normal and Cauchy distributions. The log-normal distribution does have analytic moments, but unfortunately they diverge; whereas, the Cauchy distribution does not have any analytic moments at all (they are all indeterminate).

There are various methods employed in the literature to obtain estimates of these parameters such as the maximum likeliness method [61], truncation methods [62], and asymptotic methods [63].

“Fat tail” distributions are certainly relevant in the field of computational electromagnetics. For example, it is known that the log-normal distribution can be used to characterize ceramic particle sizes [64], which may be useful in observing the smoke-stack output of a ceramics processing plant from a distance. Rain drop sizes [65] also follow a log-normal distribution, which can be useful for weather radar and climate analysis.
A fundamental requirement of the SPAD methodology is that the higher-order central moments converge (in order for the Taylor series expansion to converge). “Fat tail” distributions violate this requirement. Fortunately any of the above described methods could be used to obtain estimates of the input variables’ central moments, which could then be used in the SPAD methodology. However, each method is an approximation, and has its own limitations. Any one of these methods are candidates to solve the “fat tail” problem; however, there is a much simpler and straightforward approach.

This approach will take advantage of the fact that both the log-normal and Cauchy distributions can be written as a function of Gaussian-distributed random variables. For example, given a Gaussian random variable \( x \) with the following the transformation applied
\[
a = e^x
\]  
(B.1)
it can be observed that the new random variable \( a \) has a log-normal probability distribution. From there, \( a \) can be used as the log-normally distributed input variable in an algorithm of interest. Then, at the end of the computation, the well-behaved higher-order Gaussian central moments can be used to determine the output function’s statistical properties. As a demonstration, the DDA problem (as described in the previous and following chapters) was evaluated using the log-normal random variable as well as a Gaussian random variable that as a first step was transformed via \( y = e^x \). The relative error between the two methods is plotted in Figure B.1. As can be seen the two approaches produce almost equivalent results; thus validating the methodology.
Displacement statistics ratio, $\frac{\Delta d}{\langle d \rangle}$ [unitless]

Number of Taylor series terms, $N$

Backscattered RCS: $d_0 = 2\text{cm}$, $\langle d \rangle = 1\text{cm}$

Figure B.1: SPAD difference between Gaussian and log-normal random variables
A similar approach can be applied to a Cauchy-distributed input random variable. Given two independent Gaussian random variables $x$ and $y$, then applying the transformation

$$b = \frac{x}{y}$$  \hspace{1cm} (B.2)

it can be observed that the new random variable $b$ has a Cauchy distribution. Thus in order to handle such “ill-behaved” or “fat tailed” distributions, the simplest solution is to start with a well-behaved/well-understood random variable and simply transform that parameter into the ill-behaved distribution of interest. Then, any subsequent analysis can take advantage of the well-behaved nature of the underlying “fundamental” random distribution.

A possible downfall to this approach is that not all distributions can be represented as a transformation of an existing known well-behaved distribution. In these cases one of the various approximate methods (maximum likelihood, asymptotic, etc.) described before could be used to estimate the central moments. Another alternative approach may be to devise a transformation starting from a well-behaved distribution that gives approximately the desired statistical qualities.
APPENDIX C

Code Listing

The backend classes for the forward- and reverse-mode SPAD methodology are listed below. These codes were written in the Python computer language [66] and make use of the numpy [67] and matplotlib [68] libraries.

class array4:

    def __init__( self , ndimensions ):
        self.ndimensions = ndimensions
        self.zeroth = decimal_float( '0.0' )

        self.first = numpy.zeros( ndimensions , numpy.object )
        for c in range( 0 , ndimensions ):
            self.first[c] = numpy.zeros( 4 , numpy.object )
            for n in range( 0 , 4 ):
                self.first[c][n] = decimal_float( '0.0' )

        self.cross = triangular_matrix( ndimensions )
        for cy in range( 0 , ndimensions ):
            for cx in range( cy + 1 , ndimensions ):
                self.cross[cy,cx] = numpy.zeros( 6 , numpy.object )
                for n in range( 0 , 6 ):
                    self.cross[cy,cx][n] = decimal_float( '0.0' )

    def __neg__( self ):
        new = array4( self.ndimensions )
        new.zeroth = -self.zeroth
        for c in range( 0 , self.ndimensions ):
            for n in range( 0 , 4 ):
                new.first[c][n] = -self.first[c][n]
        for cy in range( 0 , self.ndimensions ):
            for cx in range( cy + 1 , self.ndimensions ):
                for n in range( 0 , 6 ):
                    new.cross[cy,cx][n] = -self.cross[cy,cx][n]
        return new

    def __abs__( self ):
        new = array4( self.ndimensions )
        new.zeroth = abs( self.zeroth )
        for c in range( 0 , self.ndimensions ):
            for n in range( 0 , 4 ):
                new.first[c][n] = abs( self.first[c][n] )
for cy in range(0, self.ndimensions):
    for cx in range(cy + 1, self.ndimensions):
        for n in range(0, 6):
            new.cross[cy,cx][n] = abs(self.cross[cy,cx][n])

return new

def __add__(self, other):
    if isinstance(other, array4):
        new = array4(self.ndimensions)
        new.zeroth = self.zeroth + other.zeroth
        for c in range(0, self.ndimensions):
            for n in range(0, 4):
                new.first[c][n] = self.first[c][n] + other.first[c][n]
        for cy in range(0, self.ndimensions):
            for cx in range(cy + 1, self.ndimensions):
                for n in range(0, 6):
                    new.cross[cy,cx][n] = self.cross[cy,cx][n] + other.cross[cy,cx][n]
        return new
    elif isinstance(other, decimal_float) or isinstance(other, decimal_complex) or isinstance(other, int):
        new = self.copy()
        new.zeroth += other
    else:
        raise AttributeError('%s + %s not supported' % (self.__class__, other.__class__))

return new

def __mul__(self, other):
    if isinstance(other, array4):
        factorial = numpy.array([decimal_float('1.0'), decimal_float('1.0'),
                                  decimal_float('2.0'),
                                  decimal_float('6.0'),
                                  decimal_float('24.0')])

        new = array4(self.ndimensions)
        new.zeroth = self.zeroth*other.zeroth
        for c in range(0, self.ndimensions):
            for n in range(0, 4):
                new.first[c][n] += self.zeroth*other.first[c][n]
                new.first[c][n] += self.first[c][n]*other.zeroth
                for m in range(0, n):
                    binom = factorial[n+1]/factorial[m+1]/factorial[n-m]
                    new.first[c][n] += binom*self.first[c][m]*other.first[c][n-m-1]
        for cy in range(0, self.ndimensions):
            for cx in range(cy + 1, self.ndimensions):
                new.cross[cy,cx][0] = self.zeroth*other.cross[cy,cx][0]
                + self.first[cy][0]*other.first[cx][0]
                + self.first[cx][0]*other.first[cy][0]
                + self.cross[cy,cx][0]*other.zeroth
                new.cross[cy,cx][1] = self.zeroth*other.cross[cy,cx][1]
                + 2*self.first[cy][0]*other.cross[cy,cx][0]
                + self.first[cy][1]*other.first[cx][0]
                + self.cross[cy,cx][1]*other.zeroth
                new.cross[cy,cx][2] = self.zeroth*other.cross[cy,cx][2]
                + 2*self.first[cy][0]*other.cross[cy,cx][0]
                + self.first[cy][1]*other.first[cx][0]
                + self.first[cx][0]*other.first[cy][1]
                + 2*self.cross[cy,cx][0]*other.first[cx][0]
                + self.cross[cy,cx][2]*other.zeroth
                new.cross[cy,cx][3] = self.zeroth*other.cross[cy,cx][3]
                + 2*self.first[cy][0]*other.cross[cy,cx][2]
                + self.first[cy][1]*other.first[cx][1]
                + 2*self.first[cy][0]*other.cross[cy,cx][1]
                + 4*self.cross[cy,cx][0]*other.cross[cy,cx][0]
                + 2*self.cross[cy,cx][1]*other.first[cy][0]
\[\begin{align*}
+ \text{self.first}[\text{cy}][1] \cdot \text{other.first}[\text{cx}][1] \\
+ 2 \cdot \text{self.cross}[\text{cy},\text{cx}][2] \cdot \text{other.first}[\text{cx}][0] \\
+ \text{self.cross}[\text{cy},\text{cx}][3] \cdot \text{other.zeroth}
\end{align*}\]

\[\begin{align*}
\text{new.cross}[\text{cy},\text{cx}][4] &= \text{self.zeroth} \cdot \text{other.cross}[\text{cy},\text{cx}][4] \\
+ 3 \cdot \text{self.first}[\text{cx}][0] \cdot \text{other.cross}[\text{cy},\text{cx}][1] \\
+ 3 \cdot \text{self.first}[\text{cx}][1] \cdot \text{other.cross}[\text{cy},\text{cx}][0] \\
+ \text{self.first}[\text{cx}][2] \cdot \text{other.first}[\text{cy}][0] \\
+ \text{self.first}[\text{cy}][0] \cdot \text{other.first}[\text{cx}][2] \\
+ 3 \cdot \text{self.first}[\text{cy},\text{cx}][0] \cdot \text{other.first}[\text{cx}][1] \\
+ 3 \cdot \text{self.first}[\text{cy},\text{cx}][1] \cdot \text{other.first}[\text{cx}][0] \\
+ \text{self.cross}[\text{cy},\text{cx}][4] \cdot \text{other.zeroth}
\end{align*}\]

\[\begin{align*}
\text{new.cross}[\text{cy},\text{cx}][5] &= \text{self.zeroth} \cdot \text{other.cross}[\text{cy},\text{cx}][5] \\
+ 3 \cdot \text{self.first}[\text{cy}][0] \cdot \text{other.cross}[\text{cy},\text{cx}][2] \\
+ 3 \cdot \text{self.first}[\text{cy}][1] \cdot \text{other.cross}[\text{cy},\text{cx}][0] \\
+ \text{self.first}[\text{cy}][2] \cdot \text{other.first}[\text{cx}][0] \\
+ \text{self.first}[\text{cx}][0] \cdot \text{other.first}[\text{cy}][2] \\
+ 3 \cdot \text{self.first}[\text{cy},\text{cx}][0] \cdot \text{other.first}[\text{cy}][1] \\
+ 3 \cdot \text{self.first}[\text{cy},\text{cx}][1] \cdot \text{other.first}[\text{cy}][0] \\
+ \text{self.cross}[\text{cy},\text{cx}][5] \cdot \text{other.zeroth}
\end{align*}\]

```python
either isinstance(other, int) or isinstance(other, decimal_float):
    new = array4(self.ndimensions)
    new.zeroth = other * self.zeroth
    for c in range(self.ndimensions):
        for n in range(0, 4):
            new.first[c][n] = other * self.first[c][n]
    for cy in range(self.ndimensions):
        for cx in range(cy + 1, self.ndimensions):
            for n in range(0, 6):
                new.cross[cy, cx][n] = other * self.cross[cy, cx][n]
else:
    raise AttributeError('%s * %s not implemented' % (self.__class__, other.__class__))
return new
```

def __pow__(self, power):
    if isinstance(power, int):
        power = decimal_float(power)
    elif isinstance(power, decimal_float):
        pass
    else:
        raise AttributeError('<value>**%s not implemented' % power.__class__)
    if power.is_zero():
        new = array4(self.ndimensions)
        new.zeroth = decimal_float('1.0')
    elif self.zeroth.is_zero():
        factor = decimal_float('1.0')
        pow_derivatives = numpy.zeros(5, numpy.object)
        pow_derivatives[0] = decimal_float('0.0')
        for n in range(1, 5):
            factor *= n
            if (n - power).is_zero():
                pow_derivatives[n] = factor
            else:
                pow_derivatives[n] = decimal_float('0.0')
        new = self.chainrule(pow_derivatives)
    else:
        new = self.chainrule(self.pow_derivatives(power, 5))
    return new

def __div__(self, other):
    if isinstance(other, int):
        value = decimal_float(other)
    else:
value = other
return self*(value**decimal_float( '-1.0' ))

def __sub__(self, other):
    return self + -other

def __radd__(self, other):
    if isinstance(other, int):
        return decimal_float(other) + self
    else:
        return other + self

def __rsub__(self, other):
    return other + -self

def __rmul__(self, other):
    return self*other

def __rdiv__(self, other):
    return self/other

def __str__(self):
    out = ''
    A = numpy.zeros((5,5), numpy.object)
    A[0,0] = self.zeroth
    for cx in range(1, 5):
        for cy in range(1, 5):
            A[cx,cy] = decimal_float('0.0')
    for cy in range(0, self.ndimensions):
        for cx in range(cy+1, self.ndimensions):
            A[1,1] = self.cross[cx,cy][0]
            A[1,2] = self.cross[cx,cy][1]
            A[2,1] = self.cross[cx,cy][2]
            A[2,2] = self.cross[cx,cy][3]
    for m in range(0, 4):
        A[0,m+1] = self.first[cy][m]
        A[m+1,0] = self.first[cx][m]
    out += str(A)
    return out

def chainrule(self, outer_derivatives):
    f = outer_derivatives
    new = array4(self.ndimensions)
    new.zeroth = f[0]
    for c in range(0, self.ndimensions):
        new.first[c][0] = self.first[c][0]*f[1]
        new.first[c][1] = self.first[c][1]*f[1]
        new.first[c][2] = self.first[c][2]*f[1]
        new.first[c][3] = self.first[c][3]*f[1]
        new.cross[c][0] = self.cross[c][0]*f[1]
        new.cross[c][1] = self.cross[c][1]*f[1]
        new.cross[c][2] = self.cross[c][2]*f[1]
        new.cross[c][3] = self.cross[c][3]*f[1]
        new.cross[cy,cx][0] = self.cross[cy,cx][0]*f[1]
        new.cross[cy,cx][1] = self.cross[cy,cx][1]*f[1]
        new.cross[cy,cx][2] = self.cross[cy,cx][2]*f[1]
        new.cross[cy,cx][3] = self.cross[cy,cx][3]*f[1]
    return new
new.cross[cy,cx][3] = self.cross[cy,cx][3]*f[1] 
+ 2*self.cross[cy,cx][2]*self.first[cy][0]*f[2] 
+ 2*self.cross[cy,cx][1]*self.first[cy][0]*f[2] 
+ 2*new.cross[cy,cx][0]*self.first[cy][0]*f[2] 
+ self.first[cy][1]*self.first[cy][0]*f[3] 
+ self.first[cy][1]*self.first[cy][0]*self.first[cy][0]*f[3] 
+ 4*new.cross[cy,cx][0]*self.first[cy][0]*self.first[cy][0]*f[3] 
+ self.first[cy][0]*self.first[cy][0]*self.first[cy][0]*f[4] 

new.cross[cy,cx][4] = self.cross[cy,cx][4]*f[1] 
+ 3*new.cross[cy,cx][3]*self.first[cy][0]*f[2] 
+ 3*self.cross[cy,cx][2]*self.first[cy][0]*f[2] 
+ 3*new.cross[cy,cx][1]*self.first[cy][0]*f[2] 
+ 3*new.cross[cy,cx][0]*self.first[cy][0]*f[2] 
+ self.first[cy][0]*self.first[cy][0]*self.first[cy][0]*f[3] 
+ 3*new.cross[cy,cx][0]*self.first[cy][0]*self.first[cy][0]*f[3] 
+ 3*new.cross[cy,cx][0]*self.first[cy][0]*self.first[cy][0]*f[3] 

new.cross[cy,cx][5] = self.cross[cy,cx][5] 
+ 3*new.cross[cy,cx][4]*self.first[cy][0]*f[2] 
+ 3*self.cross[cy,cx][3]*self.first[cy][0]*f[2] 
+ 3*new.cross[cy,cx][2]*self.first[cy][0]*f[2] 
+ 3*new.cross[cy,cx][1]*self.first[cy][0]*f[2] 
+ 3*new.cross[cy,cx][0]*self.first[cy][0]*self.first[cy][0]*f[3] 
+ 3*new.cross[cy,cx][0]*self.first[cy][0]*self.first[cy][0]*f[3] 

return new

def exp( self ):
    exp_derivatives = numpy.zeros( 5 , numpy.object )
    for n in range( 0 , 5 ) :
        exp_derivatives[n] = exp( self.zeroth )
    return self.chainrule( exp_derivatives )

def log( self ):
    log_derivatives = numpy.zeros( 5 , numpy.object )
    log_derivatives[0] = log( self.zeroth )
    log_derivatives[1:5] = self.pow_derivatives( -1 , 4 )
    return self.chainrule( log_derivatives )

def sqrt( self ):
    sqrt_derivatives = self.pow_derivatives( decimal_float( '0.5' ) , 5 )
    return self.chainrule( sqrt_derivatives )

def hankel1( self , order ):
    if self.zeroth.is_zero():
        raise AttributeError( 'unhandled singularity in hankel derivatives' )
    factor = decimal_float( '1.0' )
    f = factorial( 5 , decimal_float )
    hankel_derivatives = numpy.zeros( 5 , numpy.object )
    hankel_derivatives[0] = hankel1( self.zeroth , order )
    for n in range( 1 , 5 ) :
        m = 1
        factor **= decimal_float( '0.5' )
        hankel_derivatives[n] = decimal_complex()
        for k in range( 0 , n + 1 ) :
            binom = f[n]/f[k]/f[n-k]
            hankel_derivatives[n] += m*factor*binom*( hankel1( self.zeroth , order + 2*k - n ) )
            m *= -1
        return self.chainrule( hankel_derivatives )

def hankel2( self , order ):
    if self.zeroth.is_zero():
        raise AttributeError( 'unhandled singularity in hankel derivatives' )
    factor = decimal_float( '1.0' )
f = factorial( 5 , decimal_float )
hankel_derivatives = numpy.zeros( 5 , numpy.object )
hankel_derivatives[0] = hankel2( self.zeroth , order )
for n in range( 1 , 5 ) :
    m = 1
    factor = decimal_float( '0.5' )
    hankel_derivatives[n] = decimal_complex()
    for k in range( 0 , n + 1 ) :
        binom = f[n]/f[k]/f[n-k]
        hankel_derivatives[n] += m*factor*binom*( hankel2( self.zeroth , order + 2*k - n ) )
        m *= -1
return self.chainrule( hankel_derivatives )

def pow_derivatives( self , power , nterms ) :
    factor = decimal_float( '1.0' )/self.zeroth
    pow_derivatives = numpy.zeros( nterms , numpy.object )
    pow_derivatives[0] = self.zeroth**power
    for n in range( 1 , nterms ) :
        pow_derivatives[n] = ( power - n + 1 )*factor*pow_derivatives[n-1]
    return pow_derivatives

def copy( self ) :
    new = array4( self.ndimensions )
    new.zeroth = self.zeroth
    for c in range( 0 , self.ndimensions ) :
        for n in range( 0 , 4 ) :
            new.first[c][n] = self.first[c][n]
        for cy in range( 0 , self.ndimensions ) :
            for cx in range( cy + 1 , self.ndimensions ) :
                for n in range( 0 , 6 ) :
                    new.cross[cy,cx][n] = self.cross[cy,cx][n]
    return new

def is_zero( self ) :
    iszero = True
    if not self.zeroth.is_zero() :
        iszero = False
    for c in range( 0 , self.ndimensions ) :
        for n in range( 0 , 4 ) :
            if not self.first[c][n].is_zero() :
                iszero = False
        for cy in range( 0 , self.ndimensions ) :
            for cx in range( cy + 1 , self.ndimensions ) :
                for n in range( 0 , 6 ) :
                    if not self.cross[cy,cx][n].is_zero() :
                        iszero = False
    return iszero

def real( self ) :
    new = array4( self.ndimensions )
    new.zeroth.value = self.zeroth.real
    for c in range( 0 , self.ndimensions ) :
        for n in range( 0 , 4 ) :
            new.first[c][n].value = self.first[c][n].real
        for cy in range( 0 , self.ndimensions ) :
            for cx in range( cy + 1 , self.ndimensions ) :
                for n in range( 0 , 6 ) :
                    new.cross[cy,cx][n].value = self.cross[cy,cx][n].real
    return new

def conj( self ) :
    new = array4( self.ndimensions )
    new.zeroth = conj( self.zeroth )
    for c in range( 0 , self.ndimensions ) :
        for n in range( 0 , 4 ) :
new.first[c][n] = conj( self.first[c][n] )
for cy in range( 0 , self.ndimensions ):
    for cx in range( cy + 1 , self.ndimensions ):
        for n in range( 0 , 6 ):
            new.cross[cy,cx][n] = conj( self.cross[cy,cx][n] )
return new

def imag( self ):
    new = array4( self.ndimensions )
    new.zeroth.value = self.zeroth.imag
    for c in range( 0 , self.ndimensions ):
        for n in range( 0 , 4 ):
            new.first[c][n].value = self.first[c][n].imag
        for cy in range( 0 , self.ndimensions ):
            for cx in range( cy + 1 , self.ndimensions ):
                for n in range( 0 , 6 ):
                    new.cross[cy,cx][n].value = self.cross[cy,cx][n].imag
    return new

def average( self , covariance ):
    if not isinstance( covariance , triangular_matrix ):
        raise AttributeError( 'covariance not triangular' )
    avg = self.zeroth
    for c in range( 0 , self.ndimensions ):
        factor = decimal_float( '1.0' )
        A = self_moments( covariance , c )
        for n in range( 0 , 4 ):
            factor /= n + 1
            avg += factor*A[n]*self.first[c][n]
        for cy in range( 0 , self.ndimensions ):
            for cx in range( cy + 1 , self.ndimensions ):
                A = cross_moments( covariance , cy , cx )
                avg += A[0]*self.cross[cy,cx][0]
                avg += A[1]*self.cross[cy,cx][1]/2
                avg += A[2]*self.cross[cy,cx][2]/2
                avg += A[3]*self.cross[cy,cx][3]/4
                avg += A[4]*self.cross[cy,cx][4]/6
                avg += A[5]*self.cross[cy,cx][5]/6
    return avg

def deviation( self , covariance ):
    if not isinstance( covariance , triangular_matrix ):
        raise AttributeError( 'covariance not triangular' )
    if isinstance( self.zeroth , decimal_float ):
        mod = self - self.average( covariance )
        var = ( mod*mod ).average( covariance )
        return sqrt( var )
    elif isinstance( self.zeroth , decimal_complex ):
        new = decimal_complex()
        new.real = self.real().deviation( covariance ).value
        new.imag = self.imag().deviation( covariance ).value
        return new
    else:
        raise AttributeError( 'only float and complex supported' )

class array2:
    def __init__( self , ndimensions , order=4 ):
        if order < 2 or order > 4:
            raise AttributeError( 'lowest order supported is 2 and max is 4' )
        self.order = order
        self.ndimensions = ndimensions
        self.zeroth = decimal_float( '0.0' )

142
self.first = numpy.zeros( ndimensions, numpy.object )
for c in range( 0, ndimensions ):
    self.first[c] = numpy.zeros( self.order, numpy.object )
for n in range( 0, self.order ):
    self.first[c][n] = decimal_float( '0.0' )

def __neg__( self ):
    new = array2( self.ndimensions, self.order )
    new.zeroth = -self.zeroth
    for c in range( 0, self.ndimensions ):
        for n in range( 0, self.order ):
            new.first[c][n] = -self.first[c][n]
    return new

def __abs__( self ):
    new = array2( self.ndimensions, self.order )
    new.zeroth = abs( self.zeroth )
    for c in range( 0, self.ndimensions ):
        for n in range( 0, self.order ):
            new.first[c][n] = abs( self.first[c][n] )
    return new

def __add__( self, other ):
    if isinstance( other, array2 ):
        new = array2( self.ndimensions, self.order )
        new.zeroth = self.zeroth + other.zeroth
        for c in range( 0, self.ndimensions ):
            for n in range( 0, self.order ):
                new.first[c][n] = self.first[c][n] + other.first[c][n]
    elif isinstance( other, decimal_float ) or isinstance( other, decimal_complex )
        elif isinstance( other, int )
            new = self.copy()
            new.zeroth += other
    else:
        raise AttributeError( '%s + %s not supported' % ( self.__class__, other.__class__ ) )
    return new

def __mul__( self, other ):
    new = array2( self.ndimensions, self.order )
    if isinstance( other, array2 ):
        factorial = numpy.array( [ decimal_float( '1.0' ), decimal_float( '1.0' ), decimal_float( '2.0' ), decimal_float( '6.0' ), decimal_float( '24.0' ) ] )
        new.zeroth = self.zeroth*other.zeroth
        for c in range( 0, self.ndimensions ):
            for n in range( 0, self.order ):
                new.first[c][n] += self.zeroth*other.first[c][n]
                new.first[c][n] += self.first[c][n]*other.zeroth
                for m in range( 0, n ):
                    binom = factorial[n+1]/factorial[m+1]/factorial[n-m]
                    new.first[c][n] += binom*self.first[c][m]*other.first[c][n-m-1]
    elif isinstance( other, int ) or isinstance( other, decimal_float )
        new.zeroth = other*self.zeroth
        for c in range( 0, self.ndimensions ):
            for n in range( 0, self.order ):
                new.first[c][n] = other*self.first[c][n]
    else:
        raise AttributeError( '%s * %s not implemented' % ( self.__class__, other.__class__ ) )
    return new

def __pow__( self, power ):
if isinstance(power, int):
    power = decimal_float(power)
elif isinstance(power, decimal_float):
    pass
else:
    raise AttributeError('<value>**%s not implemented' % power.__class__)

if power.is_zero():
    new = array2(self.ndimensions, self.order)
    new.zeroth = decimal_float('1.0')
elif self.zeroth.is_zero():
    factor = decimal_float('1.0')
    pow_derivatives = numpy.zeros(self.order + 1, numpy.object)
    pow_derivatives[0] = decimal_float('0.0')
    for n in range(1, self.order + 1):
        factor **= n
        if (n - power).is_zero():
            pow_derivatives[n] = factor
        else:
            pow_derivatives[n] = decimal_float('0.0')
    new = self.chainrule(pow_derivatives)
else:
    new = self.chainrule(self.pow_derivatives(power, self.order + 1))
return new

def __div__(self, other):
    if isinstance(other, int):
        value = decimal_float(other)
    else:
        value = other
    return self*(value**decimal_float('-1.0'))

def __sub__(self, other):
    return self + -other

def __radd__(self, other):
    if isinstance(other, int):
        return decimal_float(other) + self
    else:
        return other + self

def __rsub__(self, other):
    return other + -self

def __rmul__(self, other):
    return self*other

def __rdiv__(self, other):
    return self/other

def __str__(self):
    out = ''
    A = numpy.zeros((self.order + 1, self.order + 1), numpy.object)
    A[0, 0] = self.zeroth
    for cx in range(1, self.order + 1):
        for cy in range(1, self.order + 1):
            A[cx, cy] = decimal_float('0.0')
    out += str(A)
    return out

def chainrule(self, outer_derivatives):
    f = outer_derivatives
    new = array2(self.ndimensions, self.order)
    new.zeroth = f[0]
    for c in range(0, self.ndimensions):
new.first[c][0] = self.first[c][0]*f[1]  
new.first[c][1] = self.first[c][1]*f[1]  
  + self.first[c][0]*self.first[c][0]*f[2]  
if self.order > 2:  
  new.first[c][2] = self.first[c][2]*f[1]  
  + 3* self.first[c][0]*self.first[c][1]*f[2]  
  + self.first[c][0]*self.first[c][0]*self.first[c][0]*f[3]  
if self.order > 3:  
  new.first[c][3] = self.first[c][3]*f[1]  
  + 4* self.first[c][0]*self.first[c][2]*f[2]  
  + 3* self.first[c][1]*self.first[c][1]*f[2]  
  + 6* self.first[c][0]*self.first[c][0]*self.first[c][1]*f[3]  
  + self.first[c][0]*self.first[c][0]*self.first[c][0]*self.first[c][0]*f[4]  
return new

def exp( self ):  
  exp_derivatives = numpy.zeros( 5 , numpy.object )  
  for n in range( 0 , 5 ):  
    exp_derivatives[n] = exp( self.zeroth )  
  return self.chainrule( exp_derivatives )

def log( self ):  
  log_derivatives = numpy.zeros( 5 , numpy.object )  
  log_derivatives[0] = log( self.zeroth )  
  log_derivatives[1:(self.order+1)] = self.pow_derivatives( -1 , self.order )  
  return self.chainrule( log_derivatives )

def sqrt( self ):  
  sqrt_derivatives = self.pow_derivatives( decimal_float( '0.5' ) , self.order + 1 )  
  return self.chainrule( sqrt_derivatives )

def hankel1( self , order ):  
  if self.zeroth.is_zero():  
    raise AttributeError( 'unhandled singularity in hankel derivatives' )  
  factor = decimal_float( '1.0' )  
  f = factorial( self.order + 1 , decimal_float )  
  hankel_derivatives = numpy.zeros( self.order + 1 , numpy.object )  
  hankel_derivatives[0] = hankel1( self.zeroth , order )  
  for n in range( 1 , self.order + 1 ):  
    m = 1  
    factor *= decimal_float( '0.5' )  
    hankel_derivatives[n] = decimal_complex()  
    for k in range( 0 , n + 1 ):  
      binom = f[n]/f[k]/f[n-k]  
      hankel_derivatives[n] += m*factor*binom*( hankel1( self.zeroth , order + 2*k - n ) )  
    m *= -1  
  return self.chainrule( hankel_derivatives )

def hankel2( self , order ):  
  if self.zeroth.is_zero():  
    raise AttributeError( 'unhandled singularity in hankel derivatives' )  
  factor = decimal_float( '1.0' )  
  f = factorial( self.order + 1 , decimal_float )  
  hankel_derivatives = numpy.zeros( self.order + 1 , numpy.object )  
  hankel_derivatives[0] = hankel2( self.zeroth , order )  
  for n in range( 1 , self.order + 1 ):  
    m = 1  
    factor *= decimal_float( '0.5' )  
    hankel_derivatives[n] = decimal_complex()  
    for k in range( 0 , n + 1 ):  
      binom = f[n]/f[k]/f[n-k]  
      hankel_derivatives[n] += m*factor*binom*( hankel2( self.zeroth , order + 2*k - n ) )  
    m *= -1  
  return self.chainrule( hankel_derivatives )

def pow_derivatives( self, power, nterms):
    factor = decimal_float( '1.0' )/self.zeroth
    pow_derivatives = numpy.zeros( nterms, numpy.object )
    pow_derivatives[0] = self.zeroth**power
    for n in range( 1, nterms):
        pow_derivatives[n] = (power - n + 1)*factor*pow_derivatives[n-1]
    return pow_derivatives

def copy(self):
    new = array2( self.ndimensions, self.order )
    new.zeroth = self.zeroth
    for c in range( 0, self.ndimensions ):
        for n in range( 0, self.order ):
            new.first[c][n] = self.first[c][n]
    return new

def is_zero(self):
    iszero = True
    if not self.zeroth.is_zero():
        iszero = False
    for c in range( 0, self.ndimensions ):
        for n in range( 0, self.order ):
            if not self.first[c][n].is_zero():
                iszero = False
    return iszero

def real(self):
    new = array2( self.ndimensions, self.order )
    new.zeroth.value = self.zeroth.real
    for c in range( 0, self.ndimensions ):
        for n in range( 0, self.order ):
            new.first[c][n].value = self.first[c][n].real
    return new

def conj(self):
    new = array2( self.ndimensions, self.order )
    new.zeroth = conj( self.zeroth )
    for c in range( 0, self.ndimensions ):
        for n in range( 0, self.order ):
            new.first[c][n] = conj( self.first[c][n] )
    return new

def imag(self):
    new = array2( self.ndimensions, self.order )
    new.zeroth.value = self.zeroth.imag
    for c in range( 0, self.ndimensions ):
        for n in range( 0, self.order ):
            new.first[c][n].value = self.first[c][n].imag
    return new

def average(self, covariance):
    if not isinstance( covariance, triangular_matrix ):
        raise AttributeError( 'covariance not triangular' )
    avg = self.zeroth
    for c in range( 0, self.ndimensions ):
        factor = decimal_float( '1.0' )
        A = self_moments( covariance, c )
        for n in range( 0, self.order ):
            factor /= n + 1
            avg += factor*A[n]*self.first[c][n]
    return avg

def deviation(self, covariance):
    if not isinstance( covariance, triangular_matrix ):
        raise AttributeError( 'covariance not triangular' )
if isinstance(self.zeroth, decimal_float):
    mod = self - self.average(covariance)
    var = (mod*mod).average(covariance)
    return sqrt(var)
elif isinstance(self.zeroth, decimal_complex):
    new = decimal_complex()
    new.real = self.real().deviation(covariance).value
    new.imag = self.imag().deviation(covariance).value
    return new
else:
    raise AttributeError('only float and complex supported')

class reversevalue:
    def __init__(self, trace, value, deviation=None):
        self.maxorder = 2  # hard-coded at 2 since derivative implementation is order 2 now
        self.value = value
        self.trace = trace
        self.index = trace.index

        trace.index += 1
        if deviation is not None:
            nvariables = trace.covariance.n
            newcovariance = triangular_matrix(nvariables + 1)
            for n in range(0, nvariables):
                for m in range(n, nvariables):
                    newcovariance[n,m] = trace.covariance[n,m]
            newcovariance[nvariables,nvariables] = deviation**2
            self.trace.covariance = newcovariance
            self.trace.indices.append(self.index)

    def __neg__(self):
        svalue = self.value
        new = reversevalue(self.trace, -svalue)
        self.trace.appendjacobian(new.index, self.index, 1, -1.0)
        return new

    # TODO: may need more complicated
    def __abs__(self):
        svalue = self.value
        new = reversevalue(self.trace, numpy.abs(svalue))
        self.trace.appendjacobian(new.index, self.index, 1, 1.0)
        return new

    def __add__(self, other):
        svalue = self.value
        if isinstance(other, reversevalue):
            self._checktrace(other)
            ovalue = other.value
            new = reversevalue(self.trace, svalue + ovalue)
            self.trace.appendjacobian(new.index, self.index, 1, 1.0)
            self.trace.appendjacobian(new.index, other.index, 1, 1.0)
        else:
            new = reversevalue(self.trace, svalue + other)
            self.trace.appendjacobian(new.index, self.index, 1, 1.0)
        return new

    def __sub__(self, other):
        return self + -other

    def __mul__(self, other):
        svalue = self.value
        if isinstance(other, reversevalue):
            if self.index == other.index:  # handle squaring of existing value
                # not implemented
            else:
                new = reversevalue(self.trace, svalue * other)
                self.trace.appendjacobian(new.index, self.index, 1, 1.0)
        return new
return self**2
self._checktrace( other )
value = other.value
new = reversevalue( self.trace , value*other )
self.trace.appendjacobian( new.index , self.index , 1 , value )
self.trace.appendjacobian( new.index , other.index , 1 , value )
self.trace.appendhessian( new.index , self.index , other.index , 1 , 1 , 1.0 )
else:
    new = reversevalue( self.trace , value*other )
    self.trace.appendjacobian( new.index , self.index , 1 , other )
return new
def __div__( self , other ):
    return self*( other**-1 )
def __pow__( self , power ):
    if isinstance( power , reversevalue ):
        raise AttributeError( 'not supported' )
    if self.is_zero(): # no-op for zero valued functions raised to a power
        return self
    nvalue = self.value**power
    new = reversevalue( self.trace , nvalue )
    for n in range( 0 , self.maxorder ):
        nvalue **= ( power - n )/self.value
    self.trace.appendjacobian( new.index , self.index , n + 1 , nvalue )
    return new
def __radd__( self , other ):
    return self + other
def __rsub__( self , other ):
    return other + -self
def __rmul__( self , other ):
    return self*other
def __rdiv__( self , other ):
    return other*( 1.0/self )
def __str__( self ):
    if isinstance( self.value , complex ):
        return '%i: %f+i%f' % ( self.index , self.value.real , self.value.imag )
    else:
        return '%i: %f' % ( self.index , self.value )
def _checktrace( self , other ):
    if self.trace != other.trace:
        err = 'traces differ: this library can only support one program trace'
        raise IOError( err )

def sin( self ):
    svalue = self.value
    sinvalue = numpy.sin( svalue )
    cosvalue = numpy.cos( svalue )
    new = reversevalue( self.trace , sinvalue )
    sign = 1
    for n in range( 0 , self.maxorder ):
        if n%2 == 0:
            nvalue = sign*cosvalue
        else:
            sign *= -1
            nvalue = sign*sinvalue
    self.trace.appendjacobian( new.index , self.index , n + 1 , nvalue )
    return new
def cos(self):
    svalue = self.value
    sinvalue = numpy.sin(svalue)
    cosvalue = numpy.cos(svalue)
    new = reversevalue(self.trace, cosvalue)
    sign = 1
    for n in range(0, self.maxorder):
        if n%2 == 0:
            sign *= -1
            nvalue = sign*sinvalue
        else:
            nvalue = sign*cosvalue
        self.trace.appendjacobian(new.index, self.index, n + 1, nvalue)
    return new

def log(self):
    svalue = self.value
    logvalue = numpy.log(svalue)
    new = reversevalue(self.trace, logvalue)
    nvalue = 1.0
    for n in range(0, self.maxorder):
        nvalue *= -(n + 1)/svalue
        self.trace.appendjacobian(new.index, self.index, n + 1, nvalue)
    return new

def exp(self):
    expvalue = numpy.exp(self.value)
    new = reversevalue(self.trace, expvalue)
    for n in range(0, self.maxorder):
        self.trace.appendjacobian(new.index, self.index, n + 1, expvalue)
    return new

# hardcoded at order 2 for now
def hankel2(self, order):
    svalue = self.value
    hvalue = scipy.special.hankel2(order, svalue)
    hvalue1 = 0.5*(scipy.special.hankel2(order - 1, svalue) - scipy.special.hankel2(order + 1, svalue))
    hvalue2 = 0.25*(scipy.special.hankel2(order - 2, svalue) - 2*hvalue + scipy.special.hankel2(order + 2, svalue))
    new = reversevalue(self.trace, hvalue)
    self.trace.appendjacobian(new.index, self.index, 1, hvalue1)
    self.trace.appendjacobian(new.index, self.index, 2, hvalue2)
    return new

def real(self):
    rvalue = self.value.real
    new = reversevalue(self.trace, rvalue)
    self.trace.appendjacobian(new.index, self.index, 1, 1.0)
    return new

def imag(self):
    ivalue = self.value.imag
    new = reversevalue(self.trace, ivalue)
    self.trace.appendjacobian(new.index, self.index, 1, -complex(0.0,1.0))
    return new

# TODO: needs more?
def is_zero(self):
    return self.value == 0.0

# TODO: needs more?
def copy(self):
    svalue = self.value
    new = reversevalue(self.trace, svalue)
return new

def average( self ):
    if self.maxorder > 2:
        raise AttributeError( 'order greater than 2 not yet supported' )
    avg = self.value
    #print 'reverse average:', avg
    for n in range( 0 , self.trace.covariance.n ):
        ref = self.trace.indices[n]
        A = self_moments( self.trace.covariance , n )
        avg += float( A[0] )*self.trace.derivative1( ref , self.index )
        #print n , 1 , self.trace.derivative1( n , self.index ) , avg
        avg += float( A[1] )*self.trace.derivative2( ref , ref , self.index )/2.0
        #print n , 2 , self.trace.derivative2( n , n , self.index ) , avg
    return avg

def deviation( self ):
    self2 = self**2
    var = self2.average() - self.average()**2
    return numpy.sqrt( var )

class reversetrace:
    def __init__( self ):
        self.index = 0
        self.indices = []
        self.hessian = hessianarray()
        self.jacobian = jacobianarray()
        self.covariance = triangular_matrix( 0 )
    def __str__( self ):
        return '%s
%s' % ( self.jacobian , self.hessian )
    def appendjacobian( self , n , m , orderm , value ):
        self.jacobian[n,m,orderm] = value
    def appendhessian( self , n , m , l , orderm , orderl , value ):
        self.hessian[n,m,l,orderm,orderl] = value
    def derivative1( self , k , n=None , d1={} ):
        if n is None:
            n = self.index - 1
        if n == k:
            return 1.0
        total = 0.0
        for m in range( k , n ):
            multiplier = self.jacobian[n,m,1]
            if multiplier != 0.0: # avoid further work on zero-valued terms
                if (k,n,m) not in d1.keys():
                    d1[(k,n,m)] = self.derivative1( k , m , d1 )
                total += multiplier*d1[(k,n,m)]
        return total
    def derivative2( self , k , l , n=None , d1={} , d2={} ):
        if n is None:
            n = self.index - 1
        if n == k:
            return 0.0
        if l < k:
            raise AttributeError( 'l cannot be less that k' )
        total = 0.0
        for m in range( k , n ):
            subtotal = 0.0
            for t in range( l , n ):
                if t == m:
multiplier = self.jacobian[n,m,2]
else:
    multiplier = self.hessian[n,m,t,1,1]
if multiplier != 0.0:
    if (l,n,t) not in d1.keys():
        di[(l,n,t)] = self.derivative1(l, t, d1)
    subtotal += multiplier*di[(l,n,t)]
if subtotal != 0.0:
    if (k,n,m) not in d1.keys():
        di[(k,n,m)] = self.derivative1(k, m, d1)
    total += subtotal*di[(k,n,m)]
multiplier = self.jacobian[n,m,1]
if multiplier != 0.0:
    if (n,k,l,m) not in d2.keys():
        d2[(n,k,l,m)] = self.derivative2(k, l, m, d1, d2)
    total += multiplier*d2[(n,k,l,m)]
return total
class jacobianarray:
    def __init__(self):
        self.array = {}
    def __setitem__(self, index, value):
        self._checkindex(index)
        self.array[index] = value
    def __getitem__(self, index):
        self._checkindex(index)
        if index in self.array.keys():
            return self.array[index]
        else:
            return 0.0
    def __str__(self):
        if len(self.array) == 0:
            return 'jacobian: <empty>
        string = 'jacobian:
        for key in self.array.keys():
            value = self.array[key]
            if isinstance(value, complex):
                string += ' [%i,%i,%i]: %f+i%f
            else:
                string += ' [%i,%i,%i]: %f
        return string[:-1]
    def _checkindex(self, index):
        if len(index) != 3:
            raise AttributeError('index not correct length')
class hessianarray:
    def __init__(self):
        self.array = {}
    def __setitem__(self, index, value):
        index = self._checkindex(index)
        self.array[index] = value
    def __getitem__(self, index):
        index = self._checkindex(index)
        if index in self.array.keys():
            return self.array[index]
        else:
            return 0.0
    def __str__(self):
        if len(self.array) == 0:
            return 'hessian: <empty>
        for key in self.array.keys():
            value = self.array[key]
            if isinstance(value, complex):
                string += ' [%i,%i,%i]: %f+i%f
            else:
                string += ' [%i,%i,%i]: %f
        return string[:-1]
    def _checkindex(self, index):
        if len(index) != 3:
            raise AttributeError('index not correct length')
else:
    return 0.0

def __str__( self ) :
    if len( self.array ) == 0:
        return 'hessian: <empty>'
    string = 'hessian:
    for key in self.array.keys() :
        string += ' [%i,%i,%i,%i,%i]: %f
        % (key[0], key[1], key[2], key[3], key[4], self.array[key])
    return string[:-1]

def _checkindex( self, index ) :
    if len( index ) != 5:
        raise AttributeError( 'index not correct length' )
    if index[1] < index[2]:
        return (index[0], index[2], index[1], index[4], index[3])
    elif index[1] > index[2]:
        return index
    else:
        err = 'hessian matrix index with duplicate indices: [%i,%i,%i,%i,%i] ' \
        % (index[0],index[1],index[2],index[3],index[4])
        raise AttributeError( err )


155


