Establishing Global Error Bounds for Model Reduction in Combustion

by

Geoffrey Malcolm Oxberry

B. Ch. E., University of Delaware (2006)
M. Ch. E., University of Delaware (2006)

Submitted to the Department of Chemical Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Chemical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2013

Author ....
Department of Chemical Engineering
September 26, 2012

Certified by .. ..
Paul I. Barton
Lammot du Pont Professor of Chemical Engineering
Thesis Supervisor

Certified by ....
William H. Green
Hoyt C. Hottel Professor of Chemical Engineering
Thesis Supervisor

Accepted by ....
Patrick S. Doyle
Chairman, Committee on Graduate Students

DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

Some pages in the original document contain text that runs off the edge of the page.

Establishing Global Error Bounds for Model Reduction in Combustion

by

Geoffrey Malcolm Oxberry

Submitted to the Department of Chemical Engineering
on September 26, 2012, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Chemical Engineering

Abstract

In addition to theory and experiment, simulation of reacting flows has become important in policymaking, industry, and combustion science. However, simulations of reacting flows can be extremely computationally demanding due to the wide range of length scales involved in turbulence, the wide range of time scales involved in chemical reactions, and the large number of species in detailed chemical reaction mechanisms in combustion. To compensate for limited available computational resources, reduced chemistry is used. However, the accuracy of these reduced chemistry models is usually unknown, which is of great concern in applications; if the accuracy of a simplified model is unknown, it is risky to rely on the results of that model for critical decision-making.

To address this issue, this thesis derives bounds on the global error in reduced chemistry models. First, it is shown that many model reduction methods in combustion are based on projection; all of these methods can be described using the same equation. After that, methods from the numerical solution of ODEs are used to derive separate *a priori* bounds on the global error in the solutions of reduced chemistry models for both projection-based reduced chemistry models and non-projection-based reduced chemistry models. The distinguishing feature between the two sets of bounds is that bounds on projection-based reduced chemistry models are stronger than those on non-projection-based reduced chemistry models. In both cases, the bounds are tight, but tend to drastically overestimate the error in the reduced chemistry. The *a priori* bounds on the global error in the solutions of reduced chemistry models demonstrate that if the error in the time derivatives of the state variables in the reduced model is controlled, then the error in the reduced model solution is also controlled; this thesis proves that result for the first time. Source code is included for all results presented.

After presenting these results, the development of more accurate global error information is discussed. Using the error bounds above, in concert with more accurate global error information, it should be possible to assess better the accuracy and reliability of reduced chemistry models in applications.

Thesis Supervisor: Paul I. Barton
Title: Lammot du Pont Professor of Chemical Engineering

Thesis Supervisor: William H. Green
Title: Hoyt C. Hottel Professor of Chemical Engineering

Look at us. Look at what they make you give.

Jason Bourne, The Bourne Ultimatum

Acknowledgments

First, I would like to thank my parents, Brett and Kathleen, and my siblings, Mallory and Matthew, for their support over the course of my PhD. Without my mother's support, I certainly would not have been able to finish.

Second, I would like to thank my advisers, Prof. Paul I. Barton and Prof. William H. Green, for their advice and support. Professor Green has been a great resource in helping me understand the broader implications of my research, and helping me navigate the PhD process. Professor Barton has been instrumental in helping me better articulate my ideas. Both men are brilliant researchers, and I would not have been able to finish this thesis without them.

Third, I would like to thank the Department of Energy (DOE) Computational Science Graduate Fellowship (CSGF) program and the Krell Institute staff, especially Dr. Mary Ann Leung, Jeana Gingery, Michelle King, and Jim Corones, for all of their support and guidance during my PhD thesis. In addition to providing me with four years of very generous financial support, the DOE CSGF alumni (and their friends) have taught me countless lessons in what it means to be a computational scientist. In particular, Dr. Jaydeep Bardhan and Dr. Ahmed E. Ismail have been great career mentors and have helped me understand how to be a better scientist and human being. I would also like to thank Dr. Aron Ahmadia his insight and guidance regarding the Python programming language, and to Ethan Coon and countless other CSGFers who have promoted the language. Their influence on the implementation of the examples presented in this thesis will be evident to anyone who reads it. I would like to thank Dr. David Ketcheson for his comments on Chapters 3 and 4 of this thesis, because he provided the most valuable conceptual edits for this entire manuscript. Dr. Jed Brown, for me, will always be a role model for the breadth of mathematical techniques he has mastered, spanning several branches of the numerical solution of partial differential equations and numerical linear algebra, as well as for his programming (and rock-climbing) skill. He has been a great sounding board when it comes to tackling some of the algorithmic tasks I faced over the course of this thesis. I'd like to thank Dr. Jeff Hammond and Dr. Chris Rinderspacher for very lively discussions about computational science over the course of my thesis, and Dr. Matt Reuter for his camaraderie and great physical insight. In addition to all of the networking that the DOE CSGF program provided, and countless talks about computational science from staff scientists at the national labs and from CSGF alumni, I believe that being an alumnus helped me get a job at a national laboratory. Technical service in the national interest has always been a dream of mine, since I would like to give back to the community of people who helped me get to where I am today.

Fourth, I'd like to thank the staff of MIT Medical, especially Araceli Isenia, Dr. Sherry Bauman, and Dr. Jill Colman. Over the course of my thesis, I encountered a few serious medical issues, and without their help, I definitely would not have been able to finish. I wish them the best of luck with their future endeavors.

In addition, I'd like to thank several colleagues in the Barton and Green groups for their help and support. Dr. Ray Speth's work on Cantera was instrumental to helping me open-source the implementations of the examples in my thesis, and I look forward to working with him to contribute to Cantera in the future. Dr. Richard West, Josh Allen, and the rest of the RMG team have been an excellent example of what computational scientists should be doing when developing software; in particular, I would like to thank Dr. West for helping to convince people in the group to use Python and Git. In the Barton group, I'd like to thank Achim Wechsung, Spencer Schaber, and Matt Stuber for a lot of engaging conversations. I wish you all the best of luck in your future careers.

Finally, I'd like to thank my friends for helping me get out of the lab and relax (if only a little), particularly Ben Rosehart, Stephanie Anton, Jess Martin, Colleen Dunn, Jess Cochrane, Nicole Romano, Christine Tinker, Christy Petruczok (for keeping me sane), Kristin Vicari (for inviting me to my first Red Sox-Yankees game), and Rachel Howden (especially for organizing all of the social events).

This doctoral thesis has been examined by a Committee of the Department of Chemical Engineering as follows:

Professor William M. Deen . . . .
Chairman, Thesis Committee
Carbon P. Dubbs Professor of Chemical Engineering

Professor Paul I. Barton . . . .
Thesis Supervisor
Lammot du Pont Professor of Chemical Engineering

Professor William H. Green . . . .
Thesis Supervisor
Hoyt C. Hottel Professor of Chemical Engineering

Professor Martin Z. Bazant . . . .
Member, Thesis Committee
Professor of Chemical Engineering

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Along with theory and experiment, simulations of chemically reacting flows have become important tools in making policy, business decisions, and scientific discoveries. These simulations have been used to develop legislation like the Clean Air Act [1] and the Montréal Protocol [2]. In business, simulations are used to design engines, chemical reactors, and manufacturing processes. Simulations have also been used to explain experimentally observed behavior in homogeneous charge compression ignition (HCCI) engines, developing better explanations of unburned hydrocarbons in spark ignition (SI) engines [215], and determining the main cause of stabilization in a jet-lifted flame, among other applications [33].

However, simulations of chemically reacting flows are extremely computationally demanding. These computational demands can be attributed to a few factors. Computational fluid dynamics without chemical reactions is already computationally costly for many problems of practical interest (*e.g.*, engine design, atmospheric modeling, furnace design, *etc.*) due to the importance of turbulence in many of these simulations. To simulate turbulence requires sophisticated models of fluid flow (*i.e.*, averaging or filtering approaches like Reynolds-averaged Navier-Stokes (RANS) [222] or Large Eddy Simulation (LES) [161, 180, 70, 15], accompanied by appropriate closure relationships), or resolution of extremely fine length scales (*i.e.*, using direct numerical simulation (DNS) [138, 162, 33]), each of which tends to be used in computationally costly applications. Introducing chemical reactions to the fluid flow model only complicates matters further. Many chemical processes occur at time scales orders of magnitude both slower *and* faster than the characteristic time scales of fluid flow. In addition, different chemical processes often occur at time scales that differ by orders of magnitude [130], introducing stiffness that either requires resolving very small time scales (*i.e.*, using explicit methods), or sophisticated numerical methods (*i.e.*, implicit methods). In addition to the velocity and density of the fluid flow, each chemical species being modeled requires the solution of another (generally nonlinear) partial differential equation, requiring additional memory and floating-point operations to solve. The sophisticated models demanded, the wide range of length and time scales being modeled, and the multiple partial differential equations being solved all tax existing computational resources, limiting the number of species and reactions that can be tracked, even on large parallel computers. As a result, it is difficult to simulate reacting flows with detailed chemistry, characterized by large numbers (tens, hundreds, or even thousands) of species and hundreds or thousands of reactions.

Instead, simplified chemistry is used. These simplified models may be developed empirically, or they may be derived systematically from models of detailed chemistry. This process is called *model reduction*. In either model reduction approach, the goal is to model sufficiently accurately the chemistry and physics of a reacting flow problem, given resource constraints on computation. Despite meeting constraints on computational resources, these simplified models can sometimes fail to yield sufficiently accurate results. For instance, it is known from experience that simplified chemistry can fail to predict negative temperature coefficient (NTC) behavior in the ignition delay of hydrocarbons when low-temperature chemistry is omitted or oversimplified. Simplified chemistry, sometimes simplified without any error control, also can fail to yield quantitative predictions of the measurements (*i.e.*, temperature, species concentrations, *etc.*) made during experiments, which would be of use to scientists, engineers, and policymakers.

In order to make simplified chemistry models a more useful modeling tool, methods must be developed to quantify the error in the results of these simplified models. In particular, the *global* error must be quantified, which is the difference at *all* times between the results obtained by solving the simplified chemistry model, and the solution of a more detailed, reference chemistry model, under comparable initial conditions. If necessary, the solution of the simplified chemistry model is adjusted in order to make the comparison meaningful (that is, in order to make sure that the quantities being compared are indeed comparable). A more precise, technical explanation of the global error will be given in chapters 2, 3, and 4. The global error is essentially the approximation error in the simplified chemistry calculations at every point in time. Having this error information available informs scientists, engineers, and policymakers of the accuracy of their numerical results, enabling them to make more informed decisions.

The main contributions of this these to address this need are as follows:

First, in Chapter 2, the formalism of projection-based model reduction, common in electrical engineering, control systems, aeronautical engineering, and fluid mechanics, is used to show that multiple model reduction methods used in combustion are projection-based. This work makes more accessible to a non-specialist some of the model reduction methods used in combustion, and contains an extensive literature review. Consequently, the literature review and problem introduction traditionally written in the first chapter of a thesis is deferred here to Chapter 2. This analysis also forms the motivation and background for Chapter 3. By establishing that many model reduction methods are projection-based, analysis of the error in model reduction methods can be framed in terms of projection-based model reduction as a whole, rather than attempting analysis of each method individually, which would be much less efficient.

Second, in Chapter 3, traditional theory from the numerical solution of ordinary differential equations (ODEs) is used to establish an *a priori* bound on the global error in projection-based reduced order models. This work extends a previous similar result by Rathinam and Petzold [165] that applies to orthogonal projection-based reduced order models. This theoretical result establishes the first *a priori* bounds for oblique (*i.e.*, non-orthogonal) projection-based model reduction methods; some of the methods used in combustion are oblique, as discussed in Chapter 2. These bounds require quantities that are difficult to calculate for nonlinear chemistry models (more generally, for nonlinear ODEs), and while tight, often drastically overestimate the true global error. Despite these drawbacks, these bounds are important because they establish rigorously that controlling the local error due to model reduction (briefly, the error in the time derivatives of a simplified chemistry model; a precise technical definition is given in Chapter 3) implies that the global error due to model reduction is also controlled. Furthermore, this work establishes a foundation for more accurate methods for estimating or bounding the global error due to model reduction, discussed in Chapter 5.

Third, the work in Chapter 3 is extended further in Chapter 4 to apply to *all* model reduction methods. This result is important because some model reduction methods used in combustion, such as reaction elimination (discussed in Chapter 5), are *not* projection-based. The implications for this result are similar to those for the *a priori* global error bounds on model reduction error due to projection-based model reduction; this work also has similar drawbacks. The main distinguishing feature of this result, compared to the one presented in Chapter 3, is that the *a priori* bounds presented in Chapter 4 are weaker; the generality of these bounds, however, makes up for this apparent shortcoming.

Finally, great care is taken to present freely available, modified BSD-licensed source code [150, 216] that generates all of the figures and results in Chapters 2 through 4 of this thesis in Appendices A through C of this thesis. Both MATLAB [133] and Python [209] source files are available; each implementation calculates identical results (to within platform-dependent numerical error). The source code is presented to document the results of this thesis as completely as possible and ensure that they will withstand rigorous and thorough scrutiny. Presentation of the thoroughly documented source code also enables future students and researchers to avoid any unnecessary duplication of effort, so that the work in this thesis may be built upon more easily. A major obstacle in this thesis work was incompletely documented source code written by previous researchers using poor development practices, which required a great deal of effort to correct and overcome. Currently, a researcher is not judged by the code he writes, but the articles he publishes; bad code means more time spent programming and less time writing papers. By publishing the source code, future students will be able to write more papers, and the work in this thesis has greater impact (*e.g.*, in theory, people should cite it more because the code will be useful to them). Last, but not least, the purpose of publishing the source code is to ensure that the results in this thesis are unambiguously reproducible. The reproducible research movement aims to hold computational science research to the same standard of reproducibility as experimental science research [110, 196, 65, 197, 157, 158, 134, 68, 111, 198, 50, 71, 45, 88, 182, 210, 44, 91, 47, 173]. If the results of a computational science paper cannot be reproduced, the results of that paper should be considered suspect or wrong, as is common practice in the experimental community. It is incumbent upon the authors of a computational research article to demonstrate reproducibility.

To this end, a modified BSD-licensed, unit-tested Python implementation of reaction elimination and simultaneous reaction and species elimination is also provided. Some of the theory behind these model reduction methods is discussed in Mitsos, *et al.* [137], and Bhattacharjee, *et al.* [17], as well as in Chapter 5. Prior to writing this implementation, no open-source implementation of these methods existed. It is important to demonstrate their utility through reproducibility and enable potential future collaborators to use them. The source code for this implementation is listed in Appendix D.

Chapter 2

Projection-Based Model Reduction in Combustion

2.1 Introduction

Many practical problems in combustion involve spatially inhomogeneous phenomena, and therefore require the use of numerical methods that solve large systems of coupled, nonlinear partial differential equations. Further complicating matters, the relevant physics of these phenomena involve a wide range of time and/or length scales, sometimes over ten orders of magnitude. It is not uncommon for simulations in these application areas to require hundreds of thousands of CPU-hours [33, 215] on the world's fastest supercomputers. If a researcher is willing to sacrifice some accuracy in their simulations, use of a model reduction method [203, 151, 125] may be a viable option to reduce the computational requirements.

Several model reduction methods are available for generating reduced models from detailed chemical models. However, these different methods originate from different theoretical backgrounds. A partial listing of major model reduction methods in combustion includes three major themes: exploiting the reaction-based structure of the chemical kinetics, exploiting the physics encoded by the chemical kinetics, or exploiting mathematical structure.

To exploit the reaction-based structure of the chemical kinetics, some model reduction methods operate on the chemical reaction mechanism representation of the source term directly. These methods then eliminate reactions (and usually species also) from the original, input chemical reaction mechanism to create a reduced chemical reaction mechanism, which is then converted into a reduced source term. Examples of this approach include detailed reduction [212], DRG [126, 127, 121], DRGASA [221], DRGEP [159], SEM-CM [145], integer programming approaches [160, 5, 53, 18, 154, 153, 137], and others.

To exploit the physics of chemical kinetics, some model reduction methods use arguments from classical thermodynamics to construct a manifold in state space that contains the dynamics of the reduced source term. Examples of approaches that construct physical manifolds include ICE-PIC [166, 168], RCCE [97, 96, 94], MIM [74, 75], and reaction invariants [211, 194, 69]. POD [120, 14, 165] also constructs a manifold derived from physical structure, but the physics represented by this manifold is encoded implicitly through the data points selected as inputs to this method.

To exploit the mathematical structure of chemical kinetics, some model reduction methods employ time-scale arguments to construct a manifold in state space that approximates well the dynamics of the original system occurring in the time scale range of interest; this manifold is typically called the "slow manifold". (Sometimes, it may not include the slowest dynamics.) Although these time scale arguments may arise due to physical reasoning, these methods can sometimes also be formulated using purely mathematical reasoning so that they are application-agnostic. Examples of this approach include CSP [103, 104], ILDM [130, 144], QSSA [28, 19, 21, 172], LQSSA [124, 122], functional iteration methods [67, 174, 175, 41, 191], and lumping-based approaches [213, 112, 113, 204, 89].

As the preceding discussion indicates, many model reduction methods attempt to accomplish the same goal through varying means. Despite the proliferation of these methods, one problem with the current state of model reduction in combustion is that no standard terminology or framework exists to describe model reduction methods, making it difficult to communicate about or to compare different model reduction methods. Due to the lack of standard terminology, model reduction methods are typically compared pairwise for specific applications [41, 95, 219, 76, 119, 34]; these comparisons cannot be generalized easily. In order to better understand the workings of model reduction methods, it would be helpful to develop standardized terminology to describe these methods, which would facilitate broader comparisons of these methods and the development of more general results. Here, we propose a standardized formalism for projection-based methods in combustion, building upon previous work done outside the combustion community in model reduction and iterative methods in linear algebra [23, 179, 9, 186, 27, 35].

In addition, it is useful to discuss projection-based model reduction in a geometric fashion. Having a geometric interpretation of the objects in model reduction can leverage the superior capacity of human beings to analyze visual data in comparison to numerical and text data. Previous work in this spirit includes the work done by Fraser and Roussel [67, 174, 175] to interpret the QSSA geometrically, and work done by Ren *et al.* both to develop ICE-PIC [166] and to explain effects that pull trajectories off the slow manifold in reaction-diffusion systems [167]. A better understanding of the geometry of model reduction helps researchers to understand the implications of using model reduction methods and combining them, as in [22] and [123].

This article addresses the aforementioned problems as follows. First, a terminology is developed to define what is meant by a projection-based model reduction method, which will facilitate the discussion and comparison of methods.

After that, the properties of constant projection-based model reduction methods will also be discussed, using linear algebra and geometry where possible. One main result of this article will be to elucidate that projection-based model reduction methods have three representations: a projector representation, a Petrov-Galerkin representation (also known as a lumping), and an affine invariant representation. The mathematical relationships among these representations will provide researchers with standard, method-agnostic language for the discussion and comparison of projection-based model reduction methods.

Next, to demonstrate the applicability of the projection-based model reduction formalism, examples will be given of methods that are classically presented in each of the three representations of constant projection-based model reduction methods. In particular, it is shown that POD and MIM are classically presented in the projector representation; CSP, linear species lumping [113], and reaction invariants are classically presented in the affine lumping representation; and LQSSA is classically presented in the affine invariant representation. When presenting examples of constant projection-based model reduction methods, simplifying assumptions required for the theoretical development will be discussed. Briefly, this article assumes an underlying linear manifold structure, or equivalently, it assumes that all of the matrices used in the methods are constant over the entire state space region of interest. If the manifold constructed by the method is nonlinear, it will be linearized (by taking the tangent space at a point on the manifold). Adaptive model reduction is outside the scope of this article, and will not be considered here. The relationships between the linear manifold structure and the matrices in each method will be elucidated as the exposition develops.

Finally, the limitations of the linear manifold assumption will be discussed, as well as how this formalism can be leveraged in future work.

2.2 Defining "Projection-Based Model Reduction Method"

Projection-based model reduction in combustion typically arises in ODE setting (*e.g.*, the chemistry ODE obtained by Strang splitting [199] or Godunov splitting [72] a PDE governing the state variables in a reacting flow; for examples, see [184] and references therein), where the ODE corresponds to an adiabatic-isobaric batch reactor:

$$\dot{\mathbf{y}}(t) = \mathbf{\Gamma}(\mathbf{y}(t)), \quad \mathbf{y}(0) = \mathbf{y}^*, \tag{2.1}$$

where $\mathbf{y}(t) \in \mathbb{R}^{N_S}$ represents the original state variables, specifying the thermodynamic state of the system, N_S is the number of state variables, $\mathbf{y}^* \in \mathbb{R}^{N_S}$, and $\mathbf{\Gamma} : \mathbb{R}^{N_S} \to \mathbb{R}^{N_S}$ is a continuously differentiable function describing changes in the state variables due to chemistry.

From the full model ODE (2.1), a projection-based model reduction method constructs a projected reduced model that can be expressed as

$$\dot{\mathbf{x}}(t) = \mathbf{P}\mathbf{\Gamma}(\mathbf{x}(t)), \quad \mathbf{x}(0) = \mathbf{P}(\mathbf{y}^* - \mathbf{y}_0) + \mathbf{y}_0. \tag{2.2}$$

The projected reduced model is defined by the projection matrix $\mathbf{P} \in \mathbb{R}^{N_S \times N_S}$ and the point in state space $\mathbf{y}_0 \in \mathbb{R}^{N_S}$, called the *origin* of the projected reduced model. The state variables of the projected reduced model, $\mathbf{x}(t) \in \mathbb{R}^{N_S}$, have the same physical interpretation as $\mathbf{y}(t)$.

Some model reduction methods discuss the concept of projection onto the tangent bundle of a smooth manifold as part of their development (such as CSP [219] and MIM [74]; for background on smooth manifolds, see [108, 142]). The manifolds defined by these methods are used to calculate $\mathbf{P}$, which varies with $\mathbf{x}(t)$ in these methods. To simplify the exposition, $\mathbf{P}$ (and all related matrices) will be assumed constant over a region of interest in state space, which is equivalent to assuming that the corresponding manifold is linear (i.e., an affine subspace) in that region. Consequently, any nonlinear manifold encountered in this paper will be linearized at a point by taking the tangent space.

The concept of a smooth manifold inspires the three representations of projection-based model reduction, although knowledge of manifolds is not necessary to read this paper. The projector representation of model reduction has already been presented in (2.2), and corresponds to projection onto a tangent space of the manifold at a point (as in POD). The Petrov-Galerkin projection representation of projection-based model reduction corresponds to the observation that a manifold is locally diffeomorphic to a Euclidean space of lower dimension (as in CSP, or species lumping). A smooth manifold can also be defined locally by an algebraic equation, which corresponds to the affine invariant representation of model reduction (as in LQSSA). These three representations will be discussed in the following section along with their geometric properties. It will be shown that the projector and Petrov-Galerkin projection representations are equivalent, and that both of these representations can be converted to an affine invariant representation. It will also be shown that under certain conditions, the affine invariant representation can be expressed as a projector representation.

2.3 Three Representations of Constant Projection-Based Model Reduction

From the discussion of manifolds in the previous section, the three representations of projection-based model reduction can be formulated concretely. First, the projector representation will be discussed, since it has already been presented, then the Petrov-Galerkin projection representation, followed by the affine invariant representation.

2.3.1 Projector Representation

Before presenting the projector representation, a brief aside is necessary to discuss notation. For the remainder of this paper, let $\mathcal{R}(\cdot)$ and $\mathcal{N}(\cdot)$ denote the range and nullspace of a matrix, respectively. If $A, B \subset \mathbb{R}^n$ are vector spaces, then $A + B = \{\mathbf{u} + \mathbf{v} : \mathbf{u} \in A, \mathbf{v} \in B\}$; this operation is called the sum of vector spaces A and B. If in addition, $A \cap B = \{\mathbf{0}\}$, then the sum of vector spaces is denoted $A \oplus B$ and called the direct sum of A and B instead. If $\mathbf{v} \in \mathbb{R}^n$, then $A+\mathbf{v} = \{\mathbf{u}+\mathbf{v} : \mathbf{u} \in A\}$. If A is a subspace, then $A + \mathbf{v}$ is an affine subspace. The orthogonal complementary subspace of A is denoted $A^{\perp} = \{\mathbf{v} : \mathbf{u}^{\mathrm{T}}\mathbf{v} = 0, \forall \mathbf{u} \in A\}$. If $\mathbf{A}$ is a matrix such that $\mathcal{R}(\mathbf{A}) = A$, then let $\mathbf{A}_{\perp}$ denote a matrix such that $\mathcal{R}(\mathbf{A}_{\perp}) = A^{\perp}$. Note that $\mathbf{A}_{\perp}^{\mathrm{T}}\mathbf{A} = \mathbf{0}$, and that the columns of $\mathbf{A}$ and $\mathbf{A}_{\perp}$ then form a basis for $\mathbb{R}^{n}$. Having stated this notation, we can proceed to discuss the projector representation.

As stated earlier, the projector representation obtained by reducing the full model ODE (2.1) takes the form in (2.2):

$$\dot{\mathbf{x}}(t) = \mathbf{P}\boldsymbol{\Gamma}(\mathbf{x}(t)), \quad \mathbf{x}(0) = \mathbf{P}(\mathbf{y}^{*} - \mathbf{y}_{0}) + \mathbf{y}_{0}. \tag{2.2}$$

Here, $\mathbf{P}$ is a projection matrix, so by definition, $\mathbf{P}^{2} = \mathbf{P}$. (For background on projection matrices, see [13, 10].)

A graphical depiction of the projector representation can be seen in Figure 2-1; results were generated using the ozone mechanism in [132, 189] in an adiabatic-isobaric batch reactor as a model problem. The point $\mathbf{y}_0$ was chosen to be a point $\mathbf{y}^{*}$ in the solution of the original model; $\mathbf{y}_0$ is the point of tangency between the dashed line (the original model solution) and the plane. The projector $\mathbf{P}$ was chosen so that $\mathcal{R}(\mathbf{P}) + \mathbf{y}_0$ is contained within the plane defined by the point $\mathbf{y}_0$ and the normal vector $(0, 3.552617158102808 \cdot 10^{-2}, 9.993687463257971 \cdot 10^{-1})$. This choice of projector can be seen in the shaded plane contains $\mathcal{R}(\mathbf{P}) + \mathbf{y}_0$. Mass conservation reduces this plane to a line, so that $\mathcal{R}(\mathbf{P}) + \mathbf{y}_0$ is a one-dimensional affine subspace. In more complicated cases, the reduced model solution will be curved because it will not be restricted to a one-dimensional affine subspace.Note that the solution of the original model, shown as a dashed line, diverges from that of the reduced model, shown as a solid line, at the point of tangency between the dashed line and the plane. The difference between the reduced model solution and the full model solution is due to approximation error inherent in most reduced models. Also note that the reduced model is completely contained in $\mathcal{R}(\mathbf{P}) + \mathbf{y}_0$; it will be shown later that the reduced model solution must always be contained in this linear manifold.

The matrix $\mathbf{P}$ also has the property $\mathcal{R}(\mathbf{P}) \oplus \mathcal{N}(\mathbf{P}) = \mathbb{R}^{N_S}$, which implies that

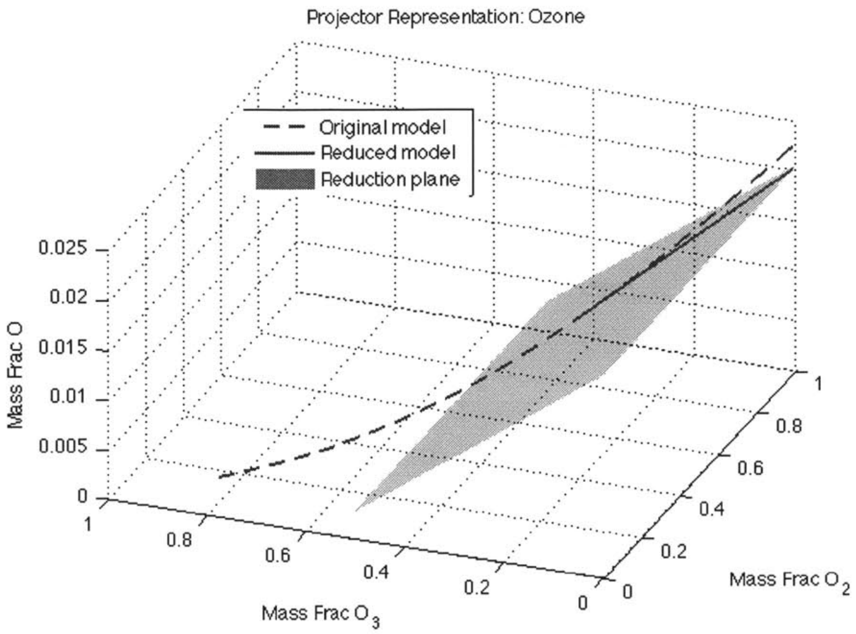

Figure 2-1: Graphical depiction of projector representation for adiabatic O_3 decomposition. Here, the model is reduced from 3 variables to 2 by projecting orthogonally onto the plane shown. The resulting reduced model solution is a line, due to mass conservation.

any vector $\mathbf{w} \in \mathbb{R}^{N_S}$ can be decomposed uniquely into $\mathbf{w} = \mathbf{P}\mathbf{w} + (\mathbf{I} - \mathbf{P})\mathbf{w}$ such that $\mathbf{P}\mathbf{w} \in \mathcal{R}(\mathbf{P})$ and $(\mathbf{I} - \mathbf{P})\mathbf{w} \in \mathcal{N}(\mathbf{P})$. Consequently, if $\mathbf{w} \in \mathcal{R}(\mathbf{P})$, then $\mathbf{w} = \mathbf{P}\mathbf{w}$. A graphical depiction of this decomposition can be seen in Figure 2-2.

From this decomposition, it also follows that the solution of (2.2) must be contained in $\mathcal{R}(\mathbf{P}) + \mathbf{y}_0$, which is a linear manifold of dimension $N_L = \mathrm{tr}(\mathbf{P})$. It also follows that if the solution $\mathbf{y} : \mathbb{R} \to \mathbb{R}^{N_S}$ of (2.1) satisfies $\mathbf{\Gamma}(\mathbf{y}(t)) \in \mathcal{R}(\mathbf{P})$ for all t and $\mathbf{y}_0 = \mathbf{y}^*$, then $\mathbf{y}$ is also a solution of (2.2), and the reduced model ODE (2.2) is exact (no approximation error).

More commonly, the solution $\mathbf{x} : \mathbb{R} \to \mathbb{R}^{N_S}$ of the projected reduced model (2.2) is not exact. Consider the difference between the right-hand sides of (2.2) and (2.1), $\mathbf{P}\mathbf{\Gamma}(\mathbf{x}(t)) - \mathbf{\Gamma}(\mathbf{y}(t))$. This right-hand side error can also be decomposed:

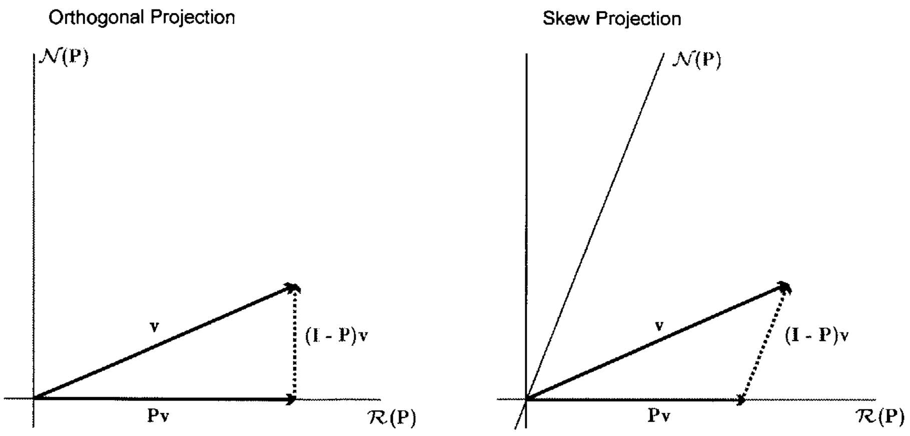

Figure 2-2: Two examples illustrating the decomposition of a vector into components along the range and nullspace of a projection matrix $\mathbf{P}$. In the orthogonal case, $\mathbf{P} = \mathbf{P}^{\mathrm{T}}$.

$$\mathbf{P}\mathbf{\Gamma}(\mathbf{x}(t)) - \mathbf{\Gamma}(\mathbf{y}(t)) = \mathbf{P}(\mathbf{\Gamma}(\mathbf{x}(t)) - \mathbf{\Gamma}(\mathbf{y}(t))) - (\mathbf{I} - \mathbf{P})\mathbf{\Gamma}(\mathbf{y}(t)). \qquad (2.3)$$

The second term on the right-hand side of (2.3) is the error component in $\mathcal{N}(\mathbf{P})$ due to projecting the right-hand side of (2.1). The first term on the right-hand side of (2.3) is the error component in $\mathcal{R}(\mathbf{P})$ that accumulates because $\mathbf{x}(t) \neq \mathbf{y}(t)$. There can also be error associated with projecting the initial conditions onto the affine subspace $\mathcal{R}(\mathbf{P}) + \mathbf{y}_0$; this error must be in $\mathcal{N}(\mathbf{P})$ because this projection is along that subspace.

Any error control in projection-based model reduction must control both components of the right-hand side error. A final consequence of the decomposition property of projectors is that the solution $\mathbf{x}$ of (2.2) must satisfy the affine invariant

$$(\mathbf{I} - \mathbf{P})(\mathbf{x}(t) - \mathbf{y}_0) = \mathbf{0}, \forall t, \qquad (2.4)$$

because $\mathbf{x}(t) \in \mathcal{R}(\mathbf{P}) + \mathbf{y}_0$ and $\mathcal{R}(\mathbf{P}) \cap \mathcal{N}(\mathbf{P}) = \{\mathbf{0}\}$. Differentiating the previous equation also yields

$$(\mathbf{I}-\mathbf{P})\dot{\mathbf{x}}(t) = \mathbf{0}, \forall t. \tag{2.5}$$

2.3.2 Affine Lumping/Petrov-Galerkin Projection Representation

An equivalent representation may be obtained by lumping state variables [113]; this approach is common in industrial simulations of chemical kinetics. The basic idea is to replace (2.1) with k similar-looking autonomous "lumped" equations

$$\dot{\tilde{\mathbf{y}}}(t) = \tilde{\mathbf{f}}(\tilde{\mathbf{y}}(t)), \tag{2.6}$$

where $\tilde{\mathbf{y}}(t) \in \mathbb{R}^k$, and $k < n$ is the number of lumped state variables. A simple and common method to relate $\tilde{\mathbf{y}}(t)$ to an approximation $\mathbf{x}(t) \in \mathbb{R}^n$ of $\mathbf{y}(t) \in \mathbb{R}^n$ is affine lumping:

$$\tilde{\mathbf{y}}(t) = \mathbf{W}^{\mathrm{T}}(\mathbf{x}(t) - \mathbf{y}_0), \tag{2.7}$$

$$\mathbf{x}(t) = \mathbf{V}\tilde{\mathbf{y}}(t) + \mathbf{y}_0, \tag{2.8}$$

where $\mathbf{V}, \mathbf{W} \in \mathbb{R}^{n\times k}$ are full rank. For the definition of the lumping operation in (2.7) to be a left inverse of the unlumping operation in (2.8), $\mathbf{V}$ and $\mathbf{W}$ must satisfy

$$\mathbf{W}^{\mathrm{T}}\mathbf{V} = \mathbf{I}. \tag{2.9}$$

In the theory of generalized inverses, the rectangular matrices $\mathbf{V}$ and $\mathbf{W}^{\mathrm{T}}$ are called $\{1,2\}$-inverses of each other [13]. Note that multiple possible choices of $\mathbf{V}$ and $\mathbf{W}$ satisfy both the full rank constraint and the biorthogonality constraint in (2.9) unless $k = n$.

Differentiating the definition in (2.8) with respect to t and substituting the definition (2.6) in for $\dot{\tilde{\mathbf{y}}}(t)$ shows that

$$\dot{\mathbf{x}}(t) = \mathbf{V}\widetilde{\mathbf{f}}(\tilde{\mathbf{y}}(t)). \tag{2.10}$$

Defining $\widetilde{\mathbf{f}} : \mathbb{R}^k \to \mathbb{R}^k$ such that

$$\widetilde{\mathbf{f}}(\tilde{\mathbf{y}}) = \widetilde{\mathbf{f}}(\mathbf{W}^{\mathrm{T}}(\mathbf{x}(t) - \mathbf{y}_0)) = \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{x}(t)), \tag{2.11}$$

(2.10) becomes

$$\dot{\mathbf{x}}(t) = \mathbf{V}\mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{x}(t)) = \mathbf{P}\mathbf{f}(\mathbf{x}(t)), \tag{2.12}$$

equivalent to (2.2), where $\mathbf{P} = \mathbf{V}\mathbf{W}^{\mathrm{T}}$ is a projection matrix. With this definition of $\widetilde{\mathbf{f}}$, (2.6) becomes

$$\dot{\tilde{\mathbf{y}}}(t) = \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{V}\tilde{\mathbf{y}}(t) + \mathbf{y}_0). \tag{2.13}$$

One common choice of initial condition for (2.6) and (2.13) is

$$\tilde{\mathbf{y}}(0) = \mathbf{W}^{\mathrm{T}}(\mathbf{y}(0) - \mathbf{y}_0) \tag{2.14}$$

under the assumption that $\mathbf{x}(0) = \mathbf{y}(0)$. However, combining (2.14) with (2.8) yields the equation

$$\mathbf{x}(0) = \mathbf{V}\tilde{\mathbf{y}}(0) + \mathbf{y}_0 = \mathbf{V}\mathbf{W}^{\mathrm{T}}(\mathbf{y}(0) - \mathbf{y}_0) + \mathbf{y}_0 = \mathbf{P}(\mathbf{y}(0) - \mathbf{y}_0) + \mathbf{y}_0, \tag{2.15}$$

which may or may not satisfy the assumption that $\mathbf{x}(0) = \mathbf{y}(0)$; this potential inconsistency illustrates that there is some approximation error in the initial condition of (2.12), and consequently also in the initial condition of (2.6) and (2.13). One way to avoid such inconsistency is to set $\mathbf{y}(0) = \mathbf{y}_0$. However, other choices of $\mathbf{y}_0$ may be used for the purposes of accuracy, such as choosing $\mathbf{y}_0$ so that $\mathbf{y}(t)$ decays onto $\mathcal{R}(\mathbf{P}) + \mathbf{y}_0$.

Petrov-Galerkin projection and lumping are identical. Petrov-Galerkin projection seeks an approximate solution of (2.1) that takes the form

$$\mathbf{x}(t) = \mathbf{y}_0 + \mathbf{V}\tilde{\mathbf{y}}(t), \tag{2.16}$$

where $\tilde{\mathbf{y}}(t) \in \mathbb{R}^k$, $k < n$, and $\mathbf{V} \in \mathbb{R}^{n \times k}$ is full rank. Differentiating both sides of (2.16) with respect to t implies that

$$\dot{\mathbf{x}}(t) = \mathbf{V}\dot{\tilde{\mathbf{y}}}(t). \tag{2.17}$$

As is customary in Galerkin-type methods, $\mathbf{W} \in \mathbb{R}^{n \times k}$ is defined so that its columns are orthogonal to a residual, $\mathbf{d}(t)$, defined as

$$\mathbf{d}(t) = \dot{\mathbf{x}}(t) - \mathbf{f}(\mathbf{x}(t)). \tag{2.18}$$

Expanding the orthogonality relation

$$\mathbf{W}^{\mathrm{T}}\mathbf{d}(t) = \mathbf{0} \tag{2.19}$$

in terms of (2.16) and (2.17) yields

$$\mathbf{W}^{\mathrm{T}}[\mathbf{V}\dot{\tilde{\mathbf{y}}}(t) - \mathbf{f}(\mathbf{y}_0 + \mathbf{V}\tilde{\mathbf{y}}(t))] = \mathbf{0}. \tag{2.20}$$

The matrix $\mathbf{W}$ is also chosen such that $\mathbf{W}^{\mathrm{T}}\mathbf{V} = \mathbf{I}$ (as in (2.9)), so that

$$\dot{\tilde{\mathbf{y}}}(t) = \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\tilde{\mathbf{y}}(t)) \tag{2.21}$$

as in (2.13).

The reason this method is also called a "projection" follows from the observation that the matrix $\mathbf{V}\mathbf{W}^{\mathrm{T}}$ is a projection matrix; similarly, any projection matrix $\mathbf{P}$ can be decomposed into the product of the form $\mathbf{V}\mathbf{W}^{\mathrm{T}}$ using a full rank decomposition. This decomposition ensures that $\mathbf{V}$ and $\mathbf{W}$ have properties consistent with Petrov-Galerkin projection; it can also be shown that $\mathcal{R}(\mathbf{P}) = \mathcal{R}(\mathbf{V})$ and $\mathcal{N}(\mathbf{P}) = \mathcal{N}(\mathbf{W}^{\mathrm{T}}) = \mathcal{R}(\mathbf{W})^{\perp}$. Using this decomposition, it can be shown that the Petrov-Galerkin and projector representations are equivalent: Multiplying both sides of (2.13) by $\mathbf{V}$ and plugging in (2.8) yields (2.2), demonstrating that the two representations correspond exactly.

A graphical depiction of the affine lumping (Petrov-Galerkin projection) representation can be seen in Figure 2-3. In this case, the initial conditions, $\mathbf{P}$, and $\mathbf{y}_0$ are the same as those for the reduced system shown in Figure 2-1. The point $\mathbf{y}_0$ corresponds to the lowermost point of the dashed and solid curves in the lower left-hand corner of Figure 2-3; note that the x-axis corresponds to a lumped variable. Temperature is not lumped.

The lumping matrices for Figure 2-3 were obtained by singular value decom-

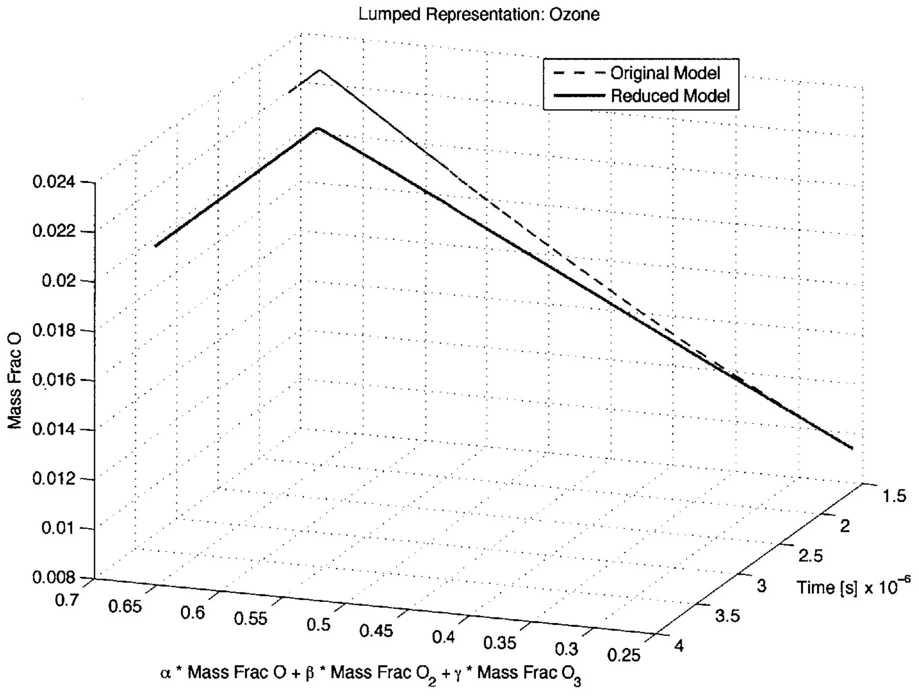

Figure 2-3: Graphical depiction of Petrov-Galerkin/affine lumping representation for the same adiabatic O_3 decomposition in Figure 2-1. Here, the lumped variable is on the x-axis, and is an affine combination of the mass fractions of O, O_2 and O_3; the coefficients of this relationship are the first column of $\mathbf{V}$ in (2.23). Note that $\mathbf{y}_0$ is the lowermost point of both curves in the lower left-hand corner; the sharp bend in the upper left-corner indicates that the mass fraction of O and lumped variable have both achieved steady state.

position. Here,

$$\mathbf{P} = \begin{bmatrix} 6.5428 \cdot 10^{-4} & 1.7751 \cdot 10^{-2} & -1.8405 \cdot 10^{-2} \\ 1.7751 \cdot 10^{-2} & 4.8159 \cdot 10^{-1} & -4.9935 \cdot 10^{-1} \\ -1.8405 \cdot 10^{-2} & -4.9935 \cdot 10^{-1} & 5.1775 \cdot 10^{-1} \end{bmatrix}, \tag{2.22}$$

$$\mathbf{P} = \mathbf{V}\mathbf{W}^T = \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \begin{bmatrix} \alpha & \beta & \gamma \end{bmatrix}, \tag{2.23}$$

$$\alpha = 2.5579 \cdot 10^{-2}, \tag{2.24}$$

$$\beta = 6.9397 \cdot 10^{-1}, \tag{2.25}$$

$$\gamma = -7.1955 \cdot 10^{-1}. \tag{2.26}$$

where **V** and **W** are as in (2.13). Since **V** = **W** for this example, the associated **P** is both symmetric and an orthogonal projector.

In Figures 2-1 and 2-3, the initial conditions $\mathbf{y}^*$ and the value of $\mathbf{y}_0$ are:

$$(y_O^*, y_{O_2}^*, y_{O_3}^*, T^*) = (0, 0.15, 0.85, 1000\,\mathrm{K}), \tag{2.27}$$

$$\begin{aligned}(y_{O,0}, y_{O_2,0}, y_{O_3,0}, T_0) = (&9.5669 \cdot 10^{-3}, 6.8325 \cdot 10^{-1}, \\ &3.0718 \cdot 10^{-1}, 2.263 \cdot 10^3\,\mathrm{K}),\end{aligned} \tag{2.28}$$

where all calculations are carried out in MATLAB r2012a [133] and Cantera 2.0 [73]; calculations were repeated using Python 2.7.3 [209] and Cantera 2.0 [73]. Details, source code, and input files can be found in Appendix A. In Figure 2-1, the full model solution is plotted starting from $\mathbf{y}^*$, whereas the reduced model solution is plotted starting from $\mathbf{y}_0$. In Figure 2-3, the solutions of both models are plotted starting from $\mathbf{y}_0$.

It is worth noting that both the computational cost and numerical accuracy of the reduced model solution are dependent on the representation of the reduced model. Solving (2.13) requires fewer operations than solving (2.2), neglecting the influence of matrix multiplies. For N_L sufficiently small, each evaluation of both the right-hand side and the Jacobian of (2.13) requires fewer matrix multiply operations (for $\mathbf{V}$ and $\mathbf{W}^{\mathrm{T}}$) than the same quantities for (2.2) (for $\mathbf{P}$). For cases of Petrov-Galerkin projection and projection without special structure, see [165] for an analysis of computational cost; although POD is considered, results generalize to oblique projectors, and focus primarily on matrix multiplies and function evaluations. Stiffness may also be a factor in comparing the computational costs of solving (2.13) and (2.2). Generally, (2.13) is no more stiff than (2.2). If (2.13) is much less stiff than (2.2), it may be possible to use an explicit method to integrate (2.13), in which case the computational costs of solving (2.13) are much less than solving (2.2). For examples of this approach using computational singular perturbation, see [207, 107]. It is important to note that generalizing the conclusions of this paragraph to the adaptive case is <u>***not***</u> straightforward. In particular, in the adaptive case, N_L changes with the current system state, changing the sizes of the matrices $\mathbf{V}$ and $\mathbf{W}$, which complicates the preceding discussion considerably, and will be deferred to future work.

For $N_L < N_S/2$, less memory is required to store the matrix pair $(\mathbf{V}, \mathbf{W})$ than $\mathbf{P}$, and for $N_L < N_S$, less memory is required to store values of the solution $\widetilde{\mathbf{y}}$ to (2.13) than is required to store the same number of values of the solution $\mathbf{x}$ to (2.2). Therefore, it is likely that solving (2.13) will require less memory than solving (2.2), which could be valuable in memory-limited applications, such as in 3-D reacting flow simulations.

When $\mathbf{W}$ consists of standard unit vectors in $\mathbb{R}^{N_S}$, computational costs decrease, as seen in POD-DEIM [30]. For a fixed $\mathbf{P}$, $\mathbf{V}$ and $\mathbf{W}^{\mathrm{T}}$ are not unique; replacing them with $\mathbf{VQ}$ and $\mathbf{Q}^{-1}\mathbf{W}^{\mathrm{T}}$, where $\mathbf{Q} \in \mathbb{R}^{N_S \times N_S}$ is invertible, works equally well from an analytical standpoint, though numerical results may differ. Good choices of $\mathbf{Q}$ can reduce the CPU time needed to solve the reduced model (2.13) and/or the numerical error in the reduced model solution. Theoretically, $\mathcal{R}(\mathbf{P}) = \mathcal{R}(\mathbf{V})$ and $\mathcal{N}(\mathbf{P}) = \mathcal{N}(\mathbf{W}^{\mathrm{T}})$ are the important objects, and are unchanged by such a transformation; they merely yield different diffeomorphisms on the manifold $\mathcal{R}(\mathbf{V}) + \mathbf{y}_0$. However, this type of transformation leaves the underlying projector unchanged, since $\mathbf{V}\mathbf{Q}\mathbf{Q}^{-1}\mathbf{W}^{\mathrm{T}} = \mathbf{V}\mathbf{W}^{\mathrm{T}} = \mathbf{P}$; it cannot convert an orthogonal projector to an oblique one, or vice versa. Therefore, if the numerical error is negligible, the same $\mathbf{x}(t)$ will be computed for each choice of $\mathbf{Q}$ and given t.

However, the numerical error may not be negligible. Given bases for $\mathcal{R}(\mathbf{P})$ and $\mathcal{N}(\mathbf{P})$, calculating $\mathbf{P}$, $\mathbf{V}$, and $\mathbf{W}$ accurately in floating point arithmetic is highly nontrivial; care should be taken to preserve numerical accuracy. See [195] for recommendations on how to calculate $\mathbf{P}$, $\mathbf{V}$, and $\mathbf{W}$. In order to reduce the numerical error in calculating the projector-vector product $\mathbf{P}\mathbf{v}$ in floating point arithmetic for any vector $\mathbf{v} \in \mathbb{R}^n$, Stewart [195] recommends calculating $\mathbf{P}\mathbf{v}$ as $\mathbf{V}\mathbf{W}^{\mathrm{T}}\mathbf{v}$, and setting $\mathbf{V}$ and $\mathbf{W}$ such that $||\mathbf{V}|| = 1$. If $||\mathbf{V}|| ||\mathbf{W}||$ is greater than $||\mathbf{P}||$, then calculating $\mathbf{P}\mathbf{v}$ in floating point arithmetic using $\mathbf{V}\mathbf{W}^{\mathrm{T}}\mathbf{v}$ can lead to a loss in accuracy compared to naïvely calculating $\mathbf{P}\mathbf{v}$ in floating point arithmetic. Stewart also recommends an alternate method for calculating $\mathbf{P}\mathbf{v}$ that is at least as accurate because it does not involve explicitly forming the matrix $\mathbf{P}$. Under certain technical conditions, this alternate method is more accurate than calculating $\mathbf{P}\mathbf{v}$ as $\mathbf{V}\mathbf{W}^{\mathrm{T}}\mathbf{v}$; these technical conditions are rarely satisfied. Furthermore, the numerical error in calculating $\mathbf{P}\mathbf{v}$ using floating point arithmetic increases as $||\mathbf{P}||$ increases, regardless of calculation method. Since $\mathbf{P}$ is singular, condition number is not a useful metric for numerical error; instead, it is recommended that modelers treat $||\mathbf{P}||$ for projection in the way that they treat the condition number for linear systems, and be alert for potentially error-prone projection operations. For a thorough analysis of numerical errors associated with calculating oblique projectors and projector-vector products, see [195].

2.3.3 Affine Invariant/Linear Manifold Representation

As noted earlier, a solution $\mathbf{x} : \mathbb{R} \to \mathbb{R}^{N_S}$ of the reduced model (2.2) satisfies the overdetermined system

$$\dot{\mathbf{x}}(t) = \mathbf{P}\boldsymbol{\Gamma}(\mathbf{x}(t)), \quad \mathbf{x}(0) = \mathbf{P}(\mathbf{y}^* - \mathbf{y}_0) + \mathbf{y}_0, \tag{2.29a}$$

$$\mathbf{0} = (\mathbf{I} - \mathbf{P})(\mathbf{x}(t) - \mathbf{y}_0). \tag{2.29b}$$

Since $\mathbf{I} - \mathbf{P}$ is also a projection matrix, a full rank decomposition into $\mathbf{I} - \mathbf{P} = \mathbf{W}_\perp \mathbf{V}_\perp^\mathrm{T}$ yields a pair of full rank $\{1, 2\}$-inverses such that $\mathbf{V}_\perp^\mathrm{T} \mathbf{W}_\perp = \mathbf{I}$, $\mathbf{V}_\perp, \mathbf{W}_\perp \in \mathbb{R}^{N_S \times (N_S - N_L)}$. The matrices $\mathbf{V}$ and $\mathbf{W}$ are the same as in the previous section. Using this information, an equivalent overdetermined system can be formed by premultiplying (2.29b) by $\mathbf{V}_\perp^\mathrm{T}$

$$\dot{\mathbf{x}}(t) = \mathbf{P}\boldsymbol{\Gamma}(\mathbf{x}(t)), \quad \mathbf{x}(0) = \mathbf{P}(\mathbf{y}^* - \mathbf{y}_0) + \mathbf{y}_0, \tag{2.30a}$$

$$\mathbf{0} = \mathbf{V}_\perp^\mathrm{T}(\mathbf{x}(t) - \mathbf{y}_0). \tag{2.30b}$$

Since $\mathbf{V}_\perp^\mathrm{T} \in \mathbb{R}^{(N_S - N_L) \times N_S}$ is a full rank matrix, there exists a permutation matrix $\mathbf{E} \in \mathbb{R}^{N_S \times N_S}$ such that $\mathbf{V}_\perp^\mathrm{T}\mathbf{E}$ can be partitioned into $\mathbf{V}_\perp^\mathrm{T}\mathbf{E} = [\mathbf{L}\ \mathbf{R}]$ such that $\mathbf{R} \in \mathbb{R}^{(N_S - N_L) \times (N_S - N_L)}$ is invertible, yielding

$$\dot{\mathbf{x}}(t) = \mathbf{P}\boldsymbol{\Gamma}(\mathbf{x}(t)), \quad \mathbf{x}(0) = \mathbf{P}(\mathbf{y}^* - \mathbf{y}_0) + \mathbf{y}_0, \tag{2.31a}$$

$$\mathbf{0} = \begin{bmatrix} \mathbf{L} & \mathbf{R} \end{bmatrix} \mathbf{E}^{-1}(\mathbf{x}(t) - \mathbf{y}_0). \tag{2.31b}$$

The entire system can be rewritten by defining

$$\begin{bmatrix} \mathbf{s}(t) \\ \mathbf{f}(t) \end{bmatrix} = \mathbf{E}^{-1}\mathbf{x}(t), \tag{2.32}$$

where $\mathbf{s}(t) \in \mathbb{R}^{N_L}$ represents "slow" state variables (*e.g.*, longer-lived, reactive species compositions) and $\mathbf{f}(t) \in \mathbb{R}^{(N_S - N_L)}$ represents algebraically determined state variables (*e.g.*, radical species compositions from linearized steady-state like approximations, inert species compositions, and species compositions from mass conservation):

$$\dot{\mathrm{s}}(t) = \begin{bmatrix} \mathbf{I}_{N_L} & \mathbf{0} \end{bmatrix} \mathbf{E}^{-1}\mathbf{P}\boldsymbol{\Gamma}\left(\mathbf{E}\begin{bmatrix} \mathrm{s}(t) \\ \mathrm{f}(t) \end{bmatrix}\right), \tag{2.33a}$$

$$\dot{\mathrm{f}}(t) = \begin{bmatrix} 0 & \mathbf{I}_{(N_S-N_L)} \end{bmatrix} \mathbf{E}^{-1}\mathbf{P}\boldsymbol{\Gamma}\left(\mathbf{E}\begin{bmatrix} \mathrm{s}(t) \\ \mathrm{f}(t) \end{bmatrix}\right), \tag{2.33b}$$

$$\mathbf{0} = \begin{bmatrix} \mathbf{L} & \mathbf{R} \end{bmatrix}\left(\begin{bmatrix} \mathrm{s}(t) \\ \mathrm{f}(t) \end{bmatrix} - \begin{bmatrix} \mathrm{s}(0) \\ \mathrm{f}(0) \end{bmatrix}\right), \tag{2.33c}$$

$$\begin{bmatrix} \mathrm{s}(0) \\ \mathrm{f}(0) \end{bmatrix} = \mathbf{E}^{-1}\mathbf{x}(0) = \mathbf{E}^{-1}[\mathbf{P}(\mathbf{y}^* - \mathbf{y}_0) + \mathbf{y}_0]. \tag{2.33d}$$

Since the algebraic equation (2.33c) can be solved explicitly for $\mathrm{f}(t)$ in place of (2.33b), ignoring (2.33b) yields the affine invariant representation:

$$\dot{\mathrm{s}}(t) = \begin{bmatrix} \mathbf{I}_{N_L} & \mathbf{0} \end{bmatrix} \mathbf{E}^{-1}\mathbf{P}\boldsymbol{\Gamma}\left(\mathbf{E}\begin{bmatrix} \mathrm{s}(t) \\ \mathrm{f}(t) \end{bmatrix}\right), \tag{2.34a}$$

$$\mathbf{0} = \begin{bmatrix} \mathbf{L} & \mathbf{R} \end{bmatrix}\left(\begin{bmatrix} \mathrm{s}(t) \\ \mathrm{f}(t) \end{bmatrix} - \begin{bmatrix} \mathrm{s}(0) \\ \mathrm{f}(0) \end{bmatrix}\right), \tag{2.34b}$$

$$\begin{bmatrix} \mathrm{s}(0) \\ \mathrm{f}(0) \end{bmatrix} = \mathbf{E}^{-1}\mathbf{x}(0). \tag{2.34c}$$

This representation of the reduced model as a differential-algebraic equation (DAE) system will be called the affine invariant representation. Since the algebraic equations are linear and $\mathbf{R}$ is invertible, (2.34b) can be solved for $\mathrm{f}(t)$ in terms of $\mathrm{s}(t)$:

$$\mathbf{f}(t) = \mathbf{f}(0) - \mathbf{R}^{-1}\mathbf{L}(\mathbf{s}(t) - \mathbf{s}(0)). \tag{2.35}$$

This equation can be substituted into (2.34a) to yield

$$\dot{\mathbf{s}}(t) = \begin{bmatrix} \mathbf{I}_{N_L} & \mathbf{0} \end{bmatrix} \mathbf{E}^{-1}\mathbf{P}\mathbf{\Gamma}\left(\mathbf{E}\begin{bmatrix} \mathbf{s}(t) \\ \mathbf{f}(0) - \mathbf{R}^{-1}\mathbf{L}(\mathbf{s}(t) - \mathbf{s}(0)) \end{bmatrix}\right), \tag{2.36}$$
$$\mathbf{s}(0) = \begin{bmatrix} \mathbf{I}_{N_L} & \mathbf{0} \end{bmatrix} \mathbf{E}^{-1}\mathbf{x}(0),$$

so the reduced model can also be solved as a systems of N_L ODEs; compare (2.36) with (2.13). This derivation essentially uses the implicit function theorem [142, 108].

A graphical depiction of the affine invariant representation can be seen in Figure 2-4. In this case, the initial conditions and $\mathbf{y}_0$ for the affine invariant system are the same as those for the reduced system shown in Figure 2-1, but the projection is chosen such that the mass fraction of O_2 is held constant at $y_{O_2} = 6.83252318 \cdot 10^{-1}$. This type of approximation (which is obviously inexact, because the sum of species mass fractions no longer equals one) is commonly used in atmospheric chemistry when O_2 is present in great excess. Consequently, only the mass fractions O and O_3 are plotted. The point $\mathbf{y}_0$ corresponds to the intersection of the reduced model solution and the original model solution in the lower right-hand corner of Figure 2-4; the mass fraction of O_2 can be found from this point by subtracting the mass fractions of O and O_3 from one. Time increases from right to left.

Reduced models written in this representation, shown in (2.34), are natural if the modeler knows some conserved or nearly-conserved quantities, *e.g.*, from conservation laws or linearized quasi-steady state-like methods. This representation also gives modelers the option to express their reduced models as DAEs that may have advantageous structure (such as sparsity, which could make them easier to solve than (2.2) or (2.13) [183]). However, often, the DAE system is not easy to

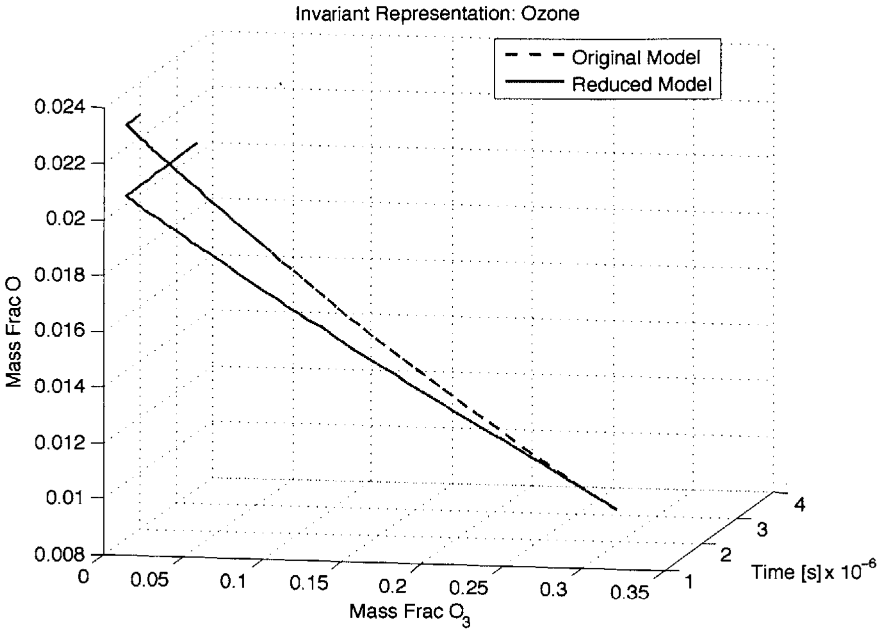

Figure 2-4: Graphical depictions of affine invariant representation for adiabatic O_3 decomposition; note that this case is different than those in Figures 2-1 and 2-3 in order to yield a more illustrative plot. Here, the mass fraction of O_2 is held constant at $y_{O_2} = 6.83252318 \cdot 10^{-1}$, and the point y_0 is the intersection of the two curves, found lower right. The sharp bend in the plot corresponds to the establishment of $O_3 = O_2 + O$ equilibrium.

solve, and the representation (2.34) is a bit unwieldy due to the number of matrices involved. Typically, the modeler has chosen $\mathbf{E}$, $\mathbf{L}$, $\mathbf{R}$, and y_0, which specify the quantities the modeler wishes to treat as conserved. If one is converting from one of the other two representations, $\mathbf{P}$ is known. Otherwise, if a model reduction method is originally expressed as a DAE, the modeler will see:

$$\dot{\mathrm{s}}(t) = \widetilde{\Gamma}(\mathrm{s}(t), \mathrm{f}(t)), \tag{2.37a}$$

$$0 = \mathrm{g}(\mathrm{s}(t), \mathrm{f}(t)). \tag{2.37b}$$

If g is affine, if $\mathrm{f}(t)$ can be solved in (2.37b) as a function of $\mathrm{s}(t)$ for all $(\mathrm{s}(t), \mathrm{f}(t))$, if there exists a permutation matrix $\mathbf{E}$ and a projection matrix $\mathbf{P}$ such that $\widetilde{\Gamma}(\mathrm{s}(t), \mathrm{f}(t)) = [\mathbf{I}_{N_L}\ \mathbf{0}]\mathbf{E}^{-1}\mathbf{P}\Gamma(\mathbf{E}(\mathrm{s}(t), \mathrm{f}(t))$, and if $\mathcal{R}(\mathbf{P}) = \{\mathbf{E}(\mathrm{s}(t), \mathrm{f}(t)) : \mathrm{g}(\mathrm{s}(t), \mathrm{f}(t)) = \mathbf{0}\}$ (that is, the algebraic equation defines exactly the range of the projection matrix, after permuting the variables so that they have the same order and interpretation as $\mathbf{x}(t)$) all hold, then (2.37) is an affine invariant representation of a projection-based reduced model. These four conditions implicitly restrict the values that $\mathbf{L}$ and $\mathbf{R}$ can take, via the implicit function theorem. Also, these conditions are not easily satisfied (or easy to check), and may admit multiple projectors and multiple permutation matrices. Consequently, it is not easy to determine if (2.37) is an affine invariant representation. However, an important special case is the linearized quasi-steady state approximation, which is an affine invariant representation because it can be expressed in the form of (2.34) with

$$\mathbf{P} = \mathbf{E}\begin{bmatrix} \mathbf{I}_{N_L} & \mathbf{0} \\ -\mathbf{R}^{-1}\mathbf{L} & \mathbf{0} \end{bmatrix}\mathbf{E}^{-1}. \tag{2.38}$$

For more details on the linearized quasi-steady state approximation, see Section 2.4.3.

2.4 Examples of Projection-Based Model Reduction Methods

Having established three representations of projection-based model reduction, examples of methods used in combustion will be presented, categorized by their classical representation in the literature.

2.4.1 Projector Representation

Two projection-based model reduction methods with classical projector representations are proper orthogonal decomposition (POD) and the method of invariant manifolds (MIM).

Proper Orthogonal Decomposition

In the ODE context, POD [120, 165] constructs a projected reduced model for (2.1) by assembling a collection of data points, classically called snapshots, $\{\mathbf{y}_i\}_{i=1}^{N_{ref}}$ such that $\mathbf{y}_i \in \mathbb{R}^{N_S}$ for all i. These snapshots are assembled into a matrix

$$\mathbf{Y} = \left[\begin{array}{ccc} (\mathbf{y}_1 - \mathbf{y}_0) & \dots & (\mathbf{y}_{N_{ref}} - \mathbf{y}_0) \end{array} \right], \tag{2.39}$$

where $\mathbf{y}_0$ is usually chosen so that $\mathbf{y}_0 = \frac{1}{N_{ref}} \sum_{i=1}^{N_{ref}} \mathbf{y}_i$. Snapshots may be data points from the solution of the original model (2.1), relevant experimental data points, or other physically realizable points. From this matrix, the SVD (singular value decomposition; see [200, 205]) of $\mathbf{Y} = \mathbf{U}\Sigma\mathcal{V}^{\mathrm{T}}$ is used to construct the reduced model. (Here, $\mathcal{V}$ is used to distinguish the Hermitian matrix that is part of the output of SVD from the $\mathbf{V}$ matrix of the affine invariant representation.) The rank N_L of the projection matrix is chosen to satisfy an error criterion (see [7] for details). POD defines a projected reduced model as in (2.2) by $\mathbf{P} = \mathbf{U}_{N_L}\mathbf{U}_{N_L}^{\mathrm{T}}$, where $\mathbf{U}_{N_L}$ is the submatrix consisting of the first N_L columns of $\mathbf{U}$; this result assumes that the singular values in Σ are arranged in descending order from left to right, which is the typical convention for numerical calculations. Note also that for POD, $\mathbf{V} = \mathbf{W} = \mathbf{U}_{N_L}$ in (2.8), (2.19), and (2.13), implying that $\mathbf{P}$ is an orthogonal projector.

Method of Invariant Manifolds

MIM [74] is motivated by the observation that when (2.1) arises from chemical kinetics, its solution $\mathbf{y} : \mathbb{R} \to \mathbb{R}^{N_S}$ initially passes through a rapid transient before it appears to be attracted to a lower-dimensional manifold $\mathcal{M} \subset \mathbb{R}^{N_S}$. For sufficiently large t, the authors of [74] posit that $\mathbf{y}(t) \in \mathcal{M}$.

MIM uses thermodynamic criteria and an iterative procedure to construct an N_L-dimensional approximation of $\mathcal{M}$ called $\mathcal{M}^{MIM}$. The remainder of the description of this method requires basic familiarity with smooth manifolds, and is independent of the rest of the paper. For any point $\mathbf{p} \in \mathcal{M}^{MIM}$, there exists a local neighborhood of p, $\mathcal{U}_{\mathrm{p}} \subset \mathbb{R}^{N_S}$, a full rank matrix $\mathbf{M} \in \mathbb{R}^{N_S \times N_L}$, a neighborhood of $\mathbf{M}^{\mathrm{T}}\mathrm{p}$, $\mathcal{V}_{\mathbf{M}^{\mathrm{T}}\mathrm{p}} \subset \mathbb{R}^{N_L}$ and a smooth ($\mathcal{C}^{\infty}$) function $\mathrm{g} : \mathcal{V}_{\mathbf{M}^{\mathrm{T}}\mathrm{p}} \rightarrow \mathcal{U}_{\mathrm{p}} \cap \mathcal{M}^{MIM}$ such that $\mathbf{M}^{\mathrm{T}}$ maps points in $\mathcal{U}_{\mathrm{p}} \cap \mathcal{M}^{MIM}$ to local coordinates on the manifold (in $\mathbb{R}^{N_L}$), and g locally defines the manifold in terms of these local coordinates. The function g and the matrix M are both defined by MIM [74]. Using these functions, MIM calculates a projector using the formula

$$\mathbf{P}(\mathbf{w}) = \mathrm{Dg}(\mathbf{M}^{\mathrm{T}}\mathbf{w})\mathbf{M}^{\mathrm{T}} \tag{2.40}$$

for $\mathbf{w} \in \mathbb{R}^{N_S}$, where Dg is the function defining the Jacobian matrix of g. Since linear manifolds are assumed, using this formalism requires that the projector function be evaluated at some point $\mathbf{y}_0 \in \mathcal{M}^{MIM}$ and treated as a constant, in which case the projector is evaluated at $\mathbf{w} = \mathbf{y}_0$. To express MIM in a affine lumping (or Petrov-Galerkin projection) representation, set $\mathbf{V} = \mathrm{Dg}(\mathbf{M}^{\mathrm{T}}\mathbf{y}_0)$ and $\mathbf{W} = \mathbf{M}$ in (2.8), (2.19), and (2.13). Nothing restricts $\mathbf{P}$ to be an orthogonal projector in this method; it is typically oblique.

2.4.2 Affine Lumping/Petrov-Galerkin Projection Representation

Three projection-based model reduction methods with classical affine lumping (or Petrov-Galerkin projection) representations are computational singular perturbation (CSP), linear species lumping (LSL), and reaction invariants (RI).

Computational Singular Perturbation

CSP [103, 104] constructs a reduced model by using a set of vectors called the CSP basis to determine the range and nullspace of a projection matrix. Let $\mathbf{A}^{CSP} \in \mathbb{R}^{N_S \times N_S}$ be the CSP basis matrix. It must be invertible, and is calculated from an initial guess (typically eigenvectors of the Jacobian of Γ evaluated at a reference point), followed by optional iterative refinement. Let $\mathbf{B}^{CSP} = (\mathbf{A}^{CSP})^{-1}$ be the CSP reciprocal basis matrix. The number of reduced state variables, N_L, is calculated using the error criteria defined by the method. The method also partitions $\mathbf{A}^{CSP}$ and $\mathbf{B}^{CSP}$ (again, using the error criteria) in the following way:

$$\mathbf{A}^{CSP} = \begin{bmatrix} \mathbf{A}^{CSP}_{fast} & \mathbf{A}^{CSP}_{slow} \end{bmatrix}, \tag{2.41}$$

$$\mathbf{B}^{CSP} = \begin{bmatrix} \mathbf{B}^{CSP}_{fast} \\ \mathbf{B}^{CSP}_{slow} \end{bmatrix}, \tag{2.42}$$

where $\mathbf{A}^{CSP}_{slow} \in \mathbb{R}^{N_S \times N_L}$ and $\mathbf{B}^{CSP}_{slow} \in \mathbb{R}^{N_L \times N_S}$. From these matrices, one constructs a reduced model using Petrov-Galerkin projection according to (2.8), (2.19), and (2.13), with $\mathbf{V} = \mathbf{A}^{CSP}_{slow}$ and $\mathbf{W}^{\mathrm{T}} = \mathbf{B}^{CSP}_{slow}$. A projector representation follows by taking $\mathbf{P} = \mathbf{V}\mathbf{W}^{\mathrm{T}} = \mathbf{A}^{CSP}_{slow}\mathbf{B}^{CSP}_{slow}$; this projector is typically oblique. Although $\mathbf{A}^{CSP}_{slow}$ and $\mathbf{B}^{CSP}_{slow}$ are typically matrix-valued functions over $\mathbb{R}^{N_S}$, for this analysis, these functions would be replaced with their values at a reference point $\mathbf{y}_0$ on the CSP manifold. In practical applications, the CSP matrices are constructed as piecewise constant functions over $\mathbb{R}^{N_S}$.

Linear Species Lumping

Historically, species lumping has been employed to reduce the computational effort needed to simulate processes that involve large numbers of species. The general idea in linear species lumping [213, 112, 113, 114, 115, 116, 117] is to define "pseudocomponents" or "lumps" that are linear combinations of species compositions. These lumps are defined either due to their physical significance (such as grouping together chemically similar species, or species that react on the same time scale) or due to their favorable mathematical properties (reducing stiffness, increasing sparsity).

Linear species lumping uses the mapping $\tilde{\mathbf{y}}(t) = \mathbf{M}^{LSL}\mathbf{y}(t)$ to lump species, and the map $\mathbf{x}(t) = \bar{\mathbf{M}}^{LSL}\tilde{\mathbf{y}}(t)$ to unlump species, where $\mathbf{M}^{LSL}$ and $\bar{\mathbf{M}}^{LSL}$ are a pair of full rank $\{1,2\}$-inverses such that $\mathbf{M}^{LSL} \in \mathbb{R}^{N_L \times N_S}$ and $\bar{\mathbf{M}}^{LSL} \in \mathbb{R}^{N_S \times N_L}$. It can be seen by inspection that linear species lumping is a Petrov-Galerkin projection representation as in (2.8), (2.19), and (2.13), where $\mathbf{V} = \bar{\mathbf{M}}^{LSL}$, $\mathbf{W}^{\mathrm{T}} = \mathbf{M}^{LSL}$, and $\mathbf{y}_0 = \mathbf{0}$; as the previous discussion indicates, there is no reason to restrict $\mathbf{y}_0$ to be $\mathbf{0}$. Again, $\mathbf{P}$ is probably an oblique projector, since it is unlikely that $\mathbf{V}$ and $\mathbf{W}$ can be made equal, even with a change of basis.

Reaction Invariants

The method of reaction invariants has been suggested by [211, 64] as a way to reduce the computational requirements of simulating chemical reactor systems with large numbers of species using a change of variables. This change of variables yields a new set of state variables that can be partitioned into time-varying quantities called *variants* and time-invariant quantities called *invariants*. Only the information contained in the variants needs to be preserved to reconstruct the solution of (2.1).

Reaction invariants assumes that (2.1) models chemical kinetics and has the form $\mathbf{\Gamma}(\mathbf{y}(t)) = \mathbf{N}\mathbf{r}(\mathbf{y}(t))$, where $\mathbf{N} \in \mathbb{R}^{N_S \times N_R}$ is the stoichiometry matrix, $\mathbf{r} : \mathbb{R}^{N_S} \to \mathbb{R}^{N_R}$ is a function returning a vector of reaction rates, N_R is the number of chemical reactions being modeled, and $\mathbf{y} : \mathbb{R} \to \mathbb{R}^{N_S}$ describes species concentrations.

Noting that vectors in $\mathcal{N}(\mathbf{N}^{\mathrm{T}})$ correspond to conservation relationships that hold for this reacting system, let $(\mathbf{D}^{RI})^{\mathrm{T}} \in \mathbb{R}^{N_S \times (N_S - N_L)}$ be a matrix whose columns are a basis for $\mathcal{N}(\mathbf{N}^{\mathrm{T}})$, where $N_S - N_L = \dim(\mathcal{N}(\mathbf{N}^{\mathrm{T}}))$. To complete the change of basis transformation, choose a matrix $\mathbf{L}^{RI} \in \mathbb{R}^{N_L \times N_S}$ such that the change-of-basis matrix $\mathbf{B}^{RI} \in \mathbb{R}^{N_S \times N_S}$ defined by

$$\mathbf{B}^{RI} = \begin{bmatrix} \mathbf{D}^{RI} \\ \mathbf{L}^{RI} \end{bmatrix} \tag{2.43}$$

is invertible. Then the functions $\mathbf{v} : \mathbb{R} \to \mathbb{R}^{N_L}$ and $\mathbf{w} : \mathbb{R} \to \mathbb{R}^{N_S - N_L}$ such that $\mathbf{v}(t) = \mathbf{L}^{RI}\mathbf{y}(t)$ and $\mathbf{w}(t) = \mathbf{D}^{RI}\mathbf{y}(t)$ define the variants and invariants of (2.1). Let

$\mathbf{Q}^{RI} \in \mathbb{R}^{N_S \times (N_S - N_L)}$ and $\mathbf{T}^{RI} \in \mathbb{R}^{N_S \times N_L}$ be matrices such that

$$(\mathbf{B}^{RI})^{-1} = \begin{bmatrix} \mathbf{Q}^{RI} & \mathbf{T}^{RI} \end{bmatrix}. \tag{2.44}$$

Setting $\mathbf{V} = \mathbf{T}^{RI}$, $\mathbf{W}^{\mathrm{T}} = \mathbf{L}^{RI}$, and $\mathbf{y}_0 = \mathbf{0}$ in (2.8), (2.19), and (2.13) illustrates how the matrices from reaction invariants can be used to carry out model reduction through Petrov-Galerkin projection. It can also be shown that $\mathbf{V}_{\perp} = (\mathbf{D}^{RI})^{\mathrm{T}}$, from which the affine invariant representation

$$\dot{\mathbf{s}}(t) = \begin{bmatrix} \mathbf{I}_{N_L} & \mathbf{0} \end{bmatrix} \mathbf{E}^{-1}\mathbf{T}^{RI}\mathbf{L}^{RI}\mathbf{\Gamma}\left(\mathbf{E}\begin{bmatrix} \mathbf{s}(t) \\ \mathbf{f}(t) \end{bmatrix}\right), \tag{2.45a}$$

$$\mathbf{0} = \mathbf{D}^{RI}\mathbf{E}\left(\begin{bmatrix} \mathbf{s}(t) \\ \mathbf{f}(t) \end{bmatrix} - \begin{bmatrix} \mathbf{s}(0) \\ \mathbf{f}(0) \end{bmatrix}\right), \quad \begin{bmatrix} \mathbf{s}(0) \\ \mathbf{f}(0) \end{bmatrix} = \mathbf{E}^{-1}\mathbf{y}(0), \tag{2.45b}$$

where

$$\begin{bmatrix} \mathbf{s}(t) \\ \mathbf{f}(t) \end{bmatrix} = \mathbf{E}^{-1}\mathbf{x}(t), \tag{2.46}$$

and $\mathbf{E}$ is an appropriately chosen permutation matrix. It can be shown that reduced models calculated using reaction invariants are exact, so $\mathbf{x}(t) = \mathbf{y}(t)$. The projector $\mathbf{P}$ is not necessarily orthogonal, since it is unlikely that $\mathbf{V}$ and $\mathbf{W}$ can be set equal in this method, even with a change of basis.

2.4.3 Affine Invariant Representation

A projection-based model reduction method with a classical affine invariant representation is the linearized quasi-steady state approximation (LQSSA).

LQSSA was developed by Lu and Law [124] to reduce the computational expense of solving nonlinear equations in the quasi-steady state approximation (QSSA) by replacing them with (quasi-)linear approximations. Assume in (2.1) that $\mathbf{y}(t)$ can be partitioned such that

$$\mathbf{y}(t) = \begin{bmatrix} \mathbf{y}_{major}(t), \\ \mathbf{y}_{QSS}(t) \end{bmatrix}, \tag{2.47}$$

where $\mathbf{y}_{major}(t) \in \mathbb{R}^{N_L}$ is a collection of known major species and $\mathbf{y}_{QSS}(t) \in \mathbb{R}^{N_S - N_L}$ is a collection of known quasi-steady state (QSS) species. The QSSA of (2.1) is typically expressed as

$$\dot{\mathbf{x}}_{major}(t) = \begin{bmatrix} \mathbf{I}_{N_L} & \mathbf{0} \end{bmatrix} \boldsymbol{\Gamma}\left(\begin{bmatrix} \mathbf{x}_{major}(t) \\ \mathbf{x}_{QSS}(t) \end{bmatrix}\right), \tag{2.48a}$$

$$\mathbf{0} = \begin{bmatrix} \mathbf{0} & \mathbf{I}_{N_S - N_L} \end{bmatrix} \boldsymbol{\Gamma}\left(\begin{bmatrix} \mathbf{x}_{major}(t) \\ \mathbf{x}_{QSS}(t) \end{bmatrix}\right). \tag{2.48b}$$

The initial conditions are discussed at the end of this subsection. LQSSA replaces the nonlinear algebraic equation in the QSSA DAE (2.48) with a quasilinear algebraic equation:

$$\dot{\mathbf{x}}_{major}(t) = \begin{bmatrix} \mathbf{I}_{N_L} & \mathbf{0} \end{bmatrix} \boldsymbol{\Gamma}\left(\begin{bmatrix} \mathbf{x}_{major}(t) \\ \mathbf{x}_{QSS}(t) \end{bmatrix}\right), \tag{2.49a}$$

$$\mathbf{0} = \mathbf{C}_{major}^{LQSSA}\mathbf{x}_{major}(t) + (\mathbf{C}_{QSS}^{LQSSA} - \mathbf{D}^{LQSSA})\mathbf{x}_{QSS}(t) + \mathbf{c}_0, \tag{2.49b}$$

where $\mathbf{C}_{major}^{LQSSA} \in \mathbb{R}^{(N_S - N_L)\times N_L}$, $\mathbf{C}_{QSS}^{LQSSA} - \mathbf{D}^{LQSSA} \in \mathbb{R}^{(N_S-N_L)\times(N_S-N_L)}$ is invertible, and $\mathbf{c}_0 \in \mathbb{R}^{(N_S-N_L)}$. In LQSSA, these quantities are actually functions defined on $\mathbb{R}^{N_L}$ (corresponding to the major species), but must be treated as constants here to obtain a linear manifold; these functions are replaced by their values at $\mathbf{y}_{major,0} \in \mathbb{R}^{N_L}$ corresponding to some point $\mathbf{y}_0$ defined as

$$\mathbf{y}_0 = \begin{bmatrix} \mathbf{y}_{major,0} \\ \mathbf{y}_{QSS,0} \end{bmatrix}, \tag{2.50}$$

chosen by the users such that it is a solution to the LQSSA DAE system. (In practical numerical computations, these matrices are assumed piecewise constant.) The quantities $\mathbf{C}_{major}^{LQSSA}$, $\mathbf{C}_{QSS}^{LQSSA} - \mathbf{D}^{LQSSA}$, and $\mathbf{c}_0$ are the coefficients of QSS relationships linearized (in a manner specific to LQSSA [124], rather than using a Taylor series) at $\mathbf{y}_0$. It follows that under these assumptions, the LQSSA DAE system can be expressed as:

$$\dot{\mathbf{x}}_{major}(t) = \begin{bmatrix} \mathbf{I}_{N_L} & \mathbf{0} \end{bmatrix} \mathbf{\Gamma} \left(\begin{bmatrix} \mathbf{x}_{major}(t) \\ \mathbf{x}_{QSS}(t) \end{bmatrix} \right), \tag{2.51a}$$

$$\mathbf{0} = \begin{bmatrix} \mathbf{C}_{major}^{LQSSA} & (\mathbf{C}_{QSS}^{LQSSA} - \mathbf{D}^{LQSSA}) \end{bmatrix} \left(\begin{bmatrix} \mathbf{x}_{major}(t) \\ \mathbf{x}_{QSS}(t) \end{bmatrix} - \begin{bmatrix} \mathbf{x}_{major,0} \\ \mathbf{x}_{QSS,0} \end{bmatrix} \right), \tag{2.51b}$$

which is an affine invariant representation where $\mathbf{E} = \mathbf{I}$, $\mathbf{L} = \mathbf{C}_{major}^{LQSSA}$, and $\mathbf{R} = (\mathbf{C}_{QSS}^{LQSSA} - \mathbf{D}^{LQSSA})$. Since the product $\widetilde{\mathbf{P}}$ is of the form $[\mathbf{I}_{N_L}\ \mathbf{0}]\,\mathbf{E}^{-1}$, a projector representation can be constructed explicitly using (2.38). Let $\mathbf{P}$ be the projection matrix corresponding to this projector representation. If the initial condition of (2.1) is

$$\mathbf{y}(0) = \mathbf{y}^* = \begin{bmatrix} \mathbf{y}_{major}^* \\ \mathbf{y}_{QSS}^* \end{bmatrix}, \tag{2.52}$$

then the corresponding initial condition for (2.51) is

$$\mathbf{x}(0) = \begin{bmatrix} \mathbf{x}_{major}(0) \\ \mathbf{x}_{QSS}(0) \end{bmatrix} = \mathbf{P}(\mathbf{y}^* - \mathbf{y}_0) + \mathbf{y}_0. \tag{2.53}$$

In this method, $\mathbf{P}$ is probably oblique.

2.5 Discussion

Above, it has been shown that many model reduction methods that look superficially very different are all of the same mathematical form given by (2.2). The accuracy of the reduced models can differ only if they choose different $(\mathbf{P}, \mathbf{y}_0)$. The numerical efficiency can differ even if $\mathbf{P}$ and $\mathbf{y}_0$ (and thus, reduced model predictions) are identical, depending in part on which of the three formulations of the reduced model are used. Depending on the choice of the matrix $\mathbf{P}$ (or the pair of matrices $\mathbf{V}$ and $\mathbf{W}$), the solution $\mathbf{x}$ of the reduced model (2.2) may not satisfy conservation laws (such as conservation of elements, mass, or energy). However, it can still give sufficiently accurate results to be useful over time scales of interest.

The major technical obstacle in developing projection-based model reduction methods is determining a manifold (and a projector $\mathbf{P}$) that gives an accurate reduced model. Many researchers in combustion believe that there exist smooth nonlinear invariant manifolds that can be used to approximate accurately the dynamics of stiff ODE systems that arise in chemical kinetics. From a purely theoretical perspective, the theory of geometrical singular perturbation theory (GSPT) [92, 61, 62, 63, 219, 220] and the stable manifold theorem (see Theorem 1.3.2 in [78]) are cited as reasons that an invariant manifold should exist. However, each of these results is local in nature; while stable manifolds can be extended using the flow of an ODE, it is not necessarily clear that the local invariant manifolds of Fenichel can be extended globally. Furthermore, both GSPT and the stable manifold theorem requires that certain technical conditions be satisfied (see citations for details). It is not easy to check these conditions in most problems of practical interest. Many systems have some conserved quantities (*e.g.*, number of atoms in a closed system); these quantities imply exact invariant manifolds. Evidence suggesting the existence of less obvious invariant manifolds has been observed in some real-world problems [41, 43, 40, 42, 39] as well as simplified model problems [67, 66, 174, 175, 176, 171, 206, 208]. Therefore, existence of an invariant manifold is normally assumed, but not proven.

The manifolds used in model reduction in combustion are generally nonlinear (whether or not they are invariant) due to the significant curvature of trajectories in applications. Consequently, the linear manifold assumption in projection-based model reduction is restrictive for problems in chemical kinetics, despite yielding tractable analysis and useful conclusions; it is *very* important to remember this assumption when using these results. In order to be more useful for rigorous computations, the theory needs to be extended to account for piecewise linear or nonlinear manifolds. It is hoped that the streamlined mathematical notation presented here for linear manifolds will be helpful in that effort.

Although many model reduction methods have the same mathematical form, very different projection matrices may be used for different purposes. For example, most of the methods presented are aimed at projecting onto slow modes along fast-changing modes, but some methods, such as POD and reaction invariants, project onto fast modes along slow modes.

2.6 Conclusions

In this work, a class of model reduction methods called "projection-based model reduction methods" was defined, standardizing the language and mathematics underlying many different methods. It was shown that there are three representations of projection-based model reduction methods. Sources of instantaneous approximation error were described. All methods that calculate the same $(\mathbf{P}, \mathbf{y}_0)$ pair give the same projected reduced model, and thus, under the same initial conditions, the same reduced model solution. From an analytical standpoint the subspaces defined by the matrices in the projector and Galerkin representations are the important objects in determining the accuracy of the reduced model, rather than the specific matrices themselves. This observation suggests a geometric interpretation of projection-based model reduction. However, clever choices of the matrices **V** and **W** in the affine lumping (or Petrov-Galerkin projection) representation can reduce the CPU time needed to solve the reduced model, or improve the numerical accuracy of the reduced model solution. Similarly, clever choices of **E**, **L**, and **R** in the affine invariant representation could aid in the solution of the DAE obtained.

Furthermore, it was demonstrated that each of these three representations appear multiple times in the literature, provided that certain technical assumptions are made. Where applicable, it was shown how the matrices in each existing method relate to concepts in projection-based model reduction. The generalizations about projection-based model reduction methods presented here make it possible to draw analogies to similar objects in different projection-based model reduction methods, enabling more systematic comparisons of model reduction methods, and hopefully spurring more advances in model reduction in combustion.

It would also be useful to develop projection-based model reduction methods that control error in such a way that it is possible to bound the approximation error in the reduced model solution relative to the corresponding full model solution. Observations in this paper, combined with new error bounding results, should aid in the development of more such methods.

Chapter 3

State-Space Error Bounds For Projection-Based Reduced Model ODEs

3.1 Introduction

Projection-based model reduction is used in a variety of contexts, including fluid mechanics [14, 98, 128, 109, 129], control theory [101], atmospheric modeling [59, 48, 193], combustion modeling [103, 104, 124, 188, 206], circuit simulation [20, 169, 170], and other applications to reduce the computational requirements of carrying out CPU-intensive equation solves. In order to be used with confidence in applications with stringent accuracy requirements, accurate bounds on or estimates of the approximation error due to model reduction are needed.

Currently, error bounds for nonlinear ODE systems only exist for the case of orthogonal projection-based model reduction methods [165] such as proper orthogonal decomposition [14] and balanced truncation [8, 6, 7], as well as for the non-projection-based method POD-DEIM [30, 32, 29]. These error bounds are based on logarithmic norms of the Jacobian matrix of the ODE right-hand side and have their theoretical roots in Gronwall's inequality [77] and work by Dahlquist on bounding the error in numerical solutions of ODEs [38]. Although these bounds typically overestimate the approximation error, in both the numerical ODE context [192, 83, 12] and the context of model reduction of ODEs, they have provided the basis for work on much more accurate *a posteriori* estimates of error in both the numerical solution of ODEs [218, 190, 105] and the approximation error due to solving reduced model ODEs [86]; again, the estimate in [86] only applies to orthogonal projection-based model reduction methods.

Similar work on bounding the approximation error due to model reduction has been carried out by Haasdonk and collaborators [81, 79, 217]. In their work on model reduction of ODEs, Gronwall- and Dahlquist-like bounds are used to bound the approximation error in the reduced model solution of linear time-varying parameterized ODEs for both orthogonal and oblique projection-based model reduction methods. No effort is made to decompose the error into in-subspace and out-of-subspace components, as in [165].

Rozza, *et al.* [178] (and references therein) describe how to construct projection-based reduced models for affinely parameterized elliptic coercive PDEs with bounds on the energy norm of the error between a desired functional of the reduced model solution and the same functional evaluated at a solution obtained using a high-dimensional finite element approximation. Although the results are presented for one specific class of PDEs, the authors mention generalizations to affinely parameterized linear elliptic noncoercive problems, problems with nonaffine parametric variation, affine linear (stable) parabolic PDEs, and elliptic (or parabolic) PDEs with polynomial nonlinearities. However, no rigorous bounds of the same type exist for elliptic (or parabolic) PDEs with nonpolynomial nonlinearities. Using these methods, it is also possible to calculate bounds on the residual between the reduced model PDE solution and the PDE solution obtained using a high-dimensional finite element approximation [148] (and references therein).

Function norm error bounds on the reduced model solution of Navier-Stokes-like equations in fluid mechanics in two spatial dimensions were obtained in [102], assuming POD snapshots in space and backward Euler integration in time; the results require a number of technical inequalities be satisfied, and does not generalize easily.

For oblique projection-based model reduction methods, such as DEIM [30], computational singular perturbation [103, 104], linearized quasi-steady state approximation [124], and others [156], neither Dahlquist-like bounds nor accurate long time *a posteriori* error estimates exist for the approximation error due to solving reduced model ODEs with nonlinear right-hand sides. Here, the approach of Rathinam and Petzold [165] is extended from orthogonal projection-based methods to include all projection-based methods. Although these bounds will not be tight, as discussed later, they can be used as the basis for future work on *a posteriori* error estimation for oblique projection-based model reduction methods.

3.2 Projection-Based Model Reduction

Here, model reduction will be discussed in the ODE setting. Consider the initial value problem

$$\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t)), \quad \mathbf{y}(0) = \mathbf{y}^*, \tag{3.1}$$

where $\mathbf{y}(t) \in \mathbb{R}^n$ represents system state variables, $\mathbf{y}^* \in \mathbb{R}^n$, and $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ with $\mathrm{f} \in C^1$.

From (3.1), a projection-based model reduction method constructs a projected reduced model

$$\dot{\mathbf{x}}(t) = \mathbf{Pf}(\mathbf{x}(t)), \quad \mathbf{x}(0) = \mathbf{P}(\mathbf{y}^* - \mathbf{y}_0) + \mathbf{y}_0 \tag{3.2}$$

by calculating a projection matrix $\mathbf{P} \in \mathbb{R}^{n \times n}$, where $\mathbf{x}(t) \in \mathbb{R}^n$ represents the state variables of the projected reduced model (which have the same significance as $\mathbf{y}(t)$), and $\mathbf{y}_0 \in \mathbb{R}^n$.

An equivalent representation may be obtained through Petrov-Galerkin projection. In this representation, one seeks a function $\mathbf{x} : \mathbb{R} \to \mathbb{R}^n$ approximating the solution $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ of (3.1) that takes the form

$$\mathbf{x}(t) = \mathbf{y}_0 + \mathbf{V}\tilde{\mathbf{y}}(t), \tag{3.3}$$

where $\mathbf{V} \in \mathbb{R}^{n \times k}$ is a full rank matrix, $k < n$ is the number of reduced state variables, and $\tilde{\mathbf{y}}(t) \in \mathbb{R}^k$ represents the reduced state variables. The original model solution and the reduced model solution are related by

$$\mathbf{y}(t) = \mathbf{x}(t) - \mathbf{e}(t), \tag{3.4}$$

where the function $\mathbf{e} : \mathbb{R} \to \mathbb{R}^n$ is the approximation error in the reduced model solution. Note that $\mathbf{e}$ must be differentiable. If $\mathbf{x}$ were identically $\mathbf{y}$, then $\mathbf{e} = \mathbf{0}$, because there would be no errors in the reduced model. In practice, $\mathbf{e} \neq \mathbf{0}$. Substituting (3.4) into (3.1) and rearranging yields

$$\dot{\mathbf{x}}(t) - \mathbf{f}(\mathbf{x}(t) - \mathbf{e}(t)) = \dot{\mathbf{e}}(t); \tag{3.5}$$

typically, in the argument of $\mathbf{f}$ in (3.5), the error term is neglected, which will be the convention in this document. Replacing $\mathbf{x}(t)$ in (3.6) with the definition in (3.3), neglecting the error term in $\mathbf{f}$, and assuming that $\mathbf{V}$ is constant yields the definition of the residual, $\mathbf{d}(t)$, of the Petrov-Galerkin projection:

$$\mathbf{V}\dot{\tilde{\mathbf{y}}}(t) - \mathbf{f}(\mathbf{V}\tilde{\mathbf{y}}(t) + \mathbf{y}_0) = \mathbf{d}(t). \tag{3.6}$$

The residual is also defined orthogonal to the range of a full rank matrix $\mathbf{W} \in \mathbb{R}^{n \times k}$

$$\mathbf{W}^{\mathrm{T}}(\mathbf{V}\dot{\tilde{\mathbf{y}}}(t) - \mathbf{f}(\mathbf{V}\tilde{\mathbf{y}}(t) + \mathbf{y}_0)) = \mathbf{W}\mathbf{d}(t) = \mathbf{0}, \tag{3.7}$$

subject to the biorthogonality constraint

$$\mathbf{W}^{\mathrm{T}}\mathbf{V} = \mathbf{I}. \tag{3.8}$$

This constraint, along with equations (3.7) and (3.3), implies that

$$\tilde{\mathbf{y}}(t) = \mathbf{W}^{\mathrm{T}}(\mathbf{x}(t) - \mathbf{y}_0). \tag{3.9}$$

Differentiating (3.9) yields the lumped reduced model

$$\dot{\tilde{\mathbf{y}}}(t) = \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{V}\tilde{\mathbf{y}}(t) + \mathbf{y}_0), \quad \tilde{\mathbf{y}}(0) = \mathbf{W}^{\mathrm{T}}(\mathbf{y}^* - \mathbf{y}_0). \tag{3.10}$$

Note that in (3.10), the initial conditions of (3.2) can be used to obtain the proper initial conditions because $\mathbf{P} = \mathbf{V}\mathbf{W}^T$ is the corresponding projection matrix. In the case where $\mathbf{V} = \mathbf{W}$, this process is called Galerkin projection, and the corresponding projector is orthogonal, with $\mathbf{P} = \mathbf{P}^{\mathrm{T}}$. Otherwise, the corresponding projector is oblique; the emphasis here will be on the oblique case. For more information about both of these representations and their equivalence, see [156].

3.3 Mathematical Preliminaries

To bound the state space error in projection-based model reduction, the approach of this paper will be to bound the norm of a solution to a nonlinear ODE. Following the presentation of [165], consider the linear system

$$\dot{\mathbf{y}}(t) = \mathbf{A}\mathbf{y}(t) + \mathbf{r}(t), \quad \mathbf{y}(0) = \mathbf{y}^*, \tag{3.11}$$

for the purpose of illustration, where $\mathbf{A} \in \mathbb{R}^{n \times n}$. The solution of (3.11) takes the form

$$\mathbf{y}(t) = e^{\mathbf{A}t}\mathbf{y}^* + \int_0^t e^{\mathbf{A}(t-\tau)}\mathbf{r}(\tau)\, \mathrm{d}\tau. \tag{3.12}$$

From (3.12), bounds on the norm of $\mathbf{y}(t)$ may be derived using Gronwall's lemma [77] or Dahlquist-like inequalities involving the logarithmic norm of $\mathbf{A}$ [83, 192]. Following the approach of [165], bounds on the norm of the function $\mathbf{y} : [0, T] \to \mathbb{R}^n$ are derived instead, where $T > 0$. In this paper, for any function $\mathbf{g} : [0, T] \to \mathbb{R}^n$, $||\mathbf{g}(t)||$ is the norm of the point $\mathbf{g}(t) \in \mathbb{R}^n$, assumed to be the 2-norm unless otherwise stated. The function norm will be denoted $||\mathbf{g}||$ and will also be the 2-norm unless otherwise stated. Keeping function norms in mind, (3.12) may be written as

$$\mathbf{y} = \mathbf{F}(T, \mathbf{A})\mathbf{r} + \mathbf{G}(T, \mathbf{A})\mathbf{y}^*,$$

where $\mathbf{F}(T, \mathbf{A}) : L^2([0, T], \mathbb{R}^n) \to L^2([0, T], \mathbb{R}^n)$ and $\mathbf{G}(T, \mathbf{A}) : \mathbb{R}^n \to L^2([0, T], \mathbb{R}^n)$ are linear operators. The desired bound on $||\mathbf{y}||$ then takes the form

$$||\mathbf{y}|| \leq ||\mathbf{F}(T, \mathbf{A})||\,||\mathbf{r}|| + ||\mathbf{G}(T, \mathbf{A})||\,||\mathbf{y}_0||. \tag{3.13}$$

Sharp estimates for the operator norms of $\mathbf{F}(T, \mathbf{A})$ and $\mathbf{G}(T, \mathbf{A})$ are difficult to obtain. As can be seen from the form of (3.12), these estimates reduce to estimating the norm of the matrix exponential. The classical approach to this problem [192] yields

$$\|e^{t\mathbf{A}}\| \leq e^{t\mu(\mathbf{A})}, \quad t \geq 0,$$

where $\mu(\mathbf{A})$ is the logarithmic norm related to the induced 2-norm of the square matrix $\mathbf{A}$:

$$\mu(\mathbf{A}) = \lim_{h \to 0^+} \frac{\|\mathbf{I} + h\mathbf{A}\| - 1}{h}.$$

The logarithmic norm may be negative, and has the property

$$\max_i \operatorname{Re} \lambda_i \leq \mu(\mathbf{A}),$$

where $\{\lambda_i\}$ are the eigenvalues of $\mathbf{A}$. Bounding the norm of the solution of a nonlinear ODE follows similar reasoning; for a more detailed explanation of the nonlinear case, see [83, 192].

3.4 Error Analysis for Projection-Based Model Reduction

The development of error bounds in this section parallels the presentation in [165]. Consider approximating the solution $\mathbf{y} : [0, T] \to \mathbb{R}^n$ of (3.1) by the solution $\mathbf{x} : [0, T] \to \mathbb{R}^n$ of (3.2) constructed by a projection-based model reduction method. A bound on the error, $\mathbf{e}(t) = \mathbf{x}(t) - \mathbf{y}(t)$, will be derived. Since $\mathbb{R}^n = \mathcal{R}(\mathbf{P}) \oplus \mathcal{N}(\mathbf{P})$, $\mathbf{e}(t)$ may be decomposed uniquely into $\mathbf{e}(t) = \mathbf{e}_c(t) + \mathbf{e}_i(t)$, where $\mathbf{e}_i(t)$ denotes error within $\mathcal{R}(\mathbf{P})$ and $\mathbf{e}_c(t)$ denotes errors within the complementary subspace $\mathcal{N}(\mathbf{P})$. Unlike the previous work in [165], $\mathbf{e}_c(t)$ and $\mathbf{e}_i(t)$ are not necessarily orthogonal because $\mathbf{P}$ may be an oblique projector. These errors can be expressed as

$$\mathbf{e}_c(t) = (\mathbf{I} - \mathbf{P})(\mathbf{x}(t) - \mathbf{y}(t)) = -(\mathbf{I} - \mathbf{P})\mathbf{y}(t) + (\mathbf{I} - \mathbf{P})\mathbf{y}_0 \tag{3.14}$$

$$\mathbf{e}_i(t) = \mathbf{P}(\mathbf{x}(t) - \mathbf{y}(t)). \tag{3.15}$$

The component $\mathbf{e}_c(t)$ is the error between $\mathbf{y}(t)$ and its projection onto $\mathcal{R}(\mathbf{P})$ along $\mathcal{N}(\mathbf{P})$. It is assumed that $\mathbf{P}$ is calculated by a projection-based model reduction method that bounds $\mathbf{e}_c(t)$ in some norm to within a specified tolerance; this assumption will be revisited after error bounds are derived.

Typically, $\mathbf{e}_i(t)$ is not explicitly bounded by a method; it consists of errors in $\mathcal{R}(\mathbf{P})$ that accumulate over time as $\mathbf{e}_c(t)$ increases in norm.

An error estimate for $\mathbf{e}_i(t)$ can be derived in terms of $\mathbf{e}_c(t)$. Differentiating (3.15) and substituting (3.1) and (3.2) for the resulting time derivatives and initial conditions yields

$$\dot{\mathbf{e}}_i(t) = \mathbf{P}[\mathbf{f}(\mathbf{y}(t) + \mathbf{e}_c(t) + \mathbf{e}_i(t)) - \mathbf{f}(\mathbf{y}(t))], \quad \mathbf{e}_i(0) = \mathbf{0}. \tag{3.16}$$

Note that $\mathbf{e}_i(0) = \mathbf{0}$ because the initial conditions of (3.1) are projected onto $\mathcal{R}(\mathbf{P})$ along $\mathcal{N}(\mathbf{P})$ in (3.2). Therefore, $\mathbf{e}_i(t)$ is governed by (3.16), where $\mathbf{e}_c(t)$ and $\mathbf{y}(t)$ may be treated as forcing terms. For a graphical illustration of the relationships among $\mathbf{y}$, $\mathbf{x}$, $\mathbf{e}_c$, $\mathbf{e}_i$, and $\mathbf{e}$, see Figure 3-1.

Before presenting error bounding results for the nonlinear ODE case, it is instructive to consider error bounding results for the linear case. Suppose that (3.1) takes the form $\dot{\mathbf{y}}(t) = \mathbf{A}\mathbf{y}(t)$ with $\mathbf{A} \in \mathbb{R}^{n \times n}$. Then (3.16) becomes

$$\dot{\mathbf{e}}_i(t) = \mathbf{P}\mathbf{A}\mathbf{e}_i(t) + \mathbf{P}\mathbf{A}\mathbf{e}_c(t), \quad \mathbf{e}_i(0) = \mathbf{0}.$$

It will be useful to define $\mathbf{V}_\perp$ and $\mathbf{W}_\perp$ as matrices whose columns span $\mathcal{R}(\mathbf{V})^\perp$

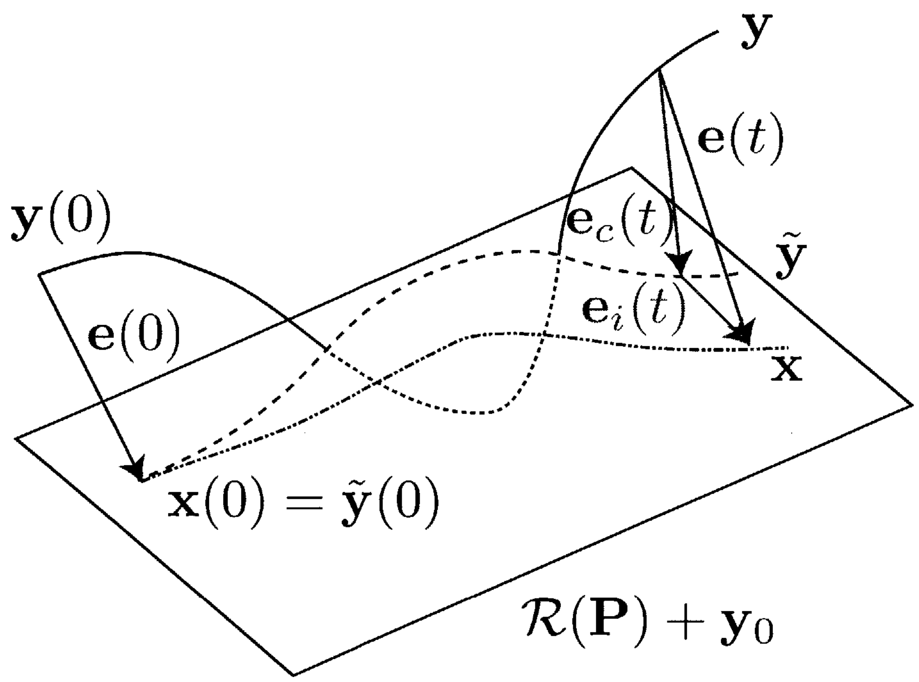

Figure 3-1: Illustrates the relationships among the full model solution $\mathbf{y}$, the projected solution $\widehat{\mathbf{y}}$ (see (3.22)), the reduced model solution $\mathbf{x}$, and the error $\mathbf{e}$. Note that the error is decomposed into a component in $\mathcal{R}(\mathbf{P})$ denoted $\mathbf{e}_i$, and a component in $\mathcal{N}(\mathbf{P})$ denoted $\mathbf{e}_c$.

and $\mathcal{R}(\mathbf{W})^{\perp}$, respectively, that also satisfy the relationship

$$\mathbf{V}_{\perp}^{\mathrm{T}}\mathbf{W}_{\perp} = \mathbf{I}. \tag{3.17}$$

These matrices will be used in developing error bounds for reduced order models. It follows that $\mathbf{V}_{\perp}, \mathbf{W}_{\perp} \in \mathbb{R}^{n\times(n-k)}$; these matrices also satisfy $\mathbf{I} - \mathbf{P} = \mathbf{W}_{\perp}\mathbf{V}_{\perp}^{\mathrm{T}}$, and can be obtained via full rank decomposition of $\mathbf{I} - \mathbf{P}$. This decomposition is not unique.

For convenience, let $\widetilde{\mathbf{A}} = \mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{V} \in \mathbb{R}^{k\times k}$, and $\widehat{\mathbf{A}} = \mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{W}_{\perp} \in \mathbb{R}^{k\times(n-k)}$. Then, using the result in (3.13) and a change of coordinates yields the bound

$$\|\mathbf{e}_i\| \leq \|\mathbf{F}(T, \widetilde{\mathbf{A}})\| \|\widehat{\mathbf{A}}\| \|\mathbf{V}\| \|\mathbf{V}_{\perp}\| \|\mathbf{e}_c\|,$$

so the total error is bounded by

$$\|\mathbf{e}\| \leq \left(\|\mathbf{F}(T, \widetilde{\mathbf{A}})\| \|\widehat{\mathbf{A}}\| \|\mathbf{V}\| \|\mathbf{V}_\perp\| + 1\right) \|\mathbf{e}_c\|.$$

Here, $\widetilde{\mathbf{A}} = \mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{V} \in \mathbb{R}^{k \times k}$, and $\widehat{\mathbf{A}} = \mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{W}_\perp \in \mathbb{R}^{k \times (n-k)}$, where $\mathbf{V}_\perp$ and $\mathbf{W}_\perp$ are matrices whose columns span $\mathcal{R}(\mathbf{V})^\perp$ and $\mathcal{R}(\mathbf{W})^\perp$, scaled so that $\mathbf{V}_\perp^{\mathrm{T}}\mathbf{W}_\perp = \mathbf{I}$. Then $\mathbf{V}_\perp, \mathbf{W}_\perp \in \mathbb{R}^{n \times (n-k)}$; these matrices also satisfy $\mathbf{I} - \mathbf{P} = \mathbf{W}_\perp \mathbf{V}_\perp^{\mathrm{T}}$, and can be obtained via full rank decomposition of $\mathbf{I} - \mathbf{P}$.

The nonlinear case proceeds in a similar fashion. Write the solution $\mathbf{y} : [0, T] \to \mathbb{R}^n$ of (3.1) and the solution $\mathbf{x} : [0, T] \to \mathbb{R}^n$ of (3.2) as

$$\mathbf{y}(t) = \mathbf{V}\mathbf{u}(t) + \mathbf{W}_\perp \mathbf{v}(t) + \mathbf{y}_0, \tag{3.18}$$

$$\mathbf{x}(t) = \mathbf{V}\mathbf{u}(t) + \mathbf{V}\mathbf{w}(t) + \mathbf{y}_0 = \mathbf{y}(t) + \mathbf{e}(t), \tag{3.19}$$

so that the errors $\mathbf{e}_c(t)$ and $\mathbf{e}_i(t)$ and the projected solution $\widehat{\mathbf{y}} : [0, T] \to \mathbb{R}^n$ are given by

$$\mathbf{e}_c(t) = -\mathbf{W}_\perp \mathbf{v}(t) = \widehat{\mathbf{y}}(t) - \mathbf{y}(t), \tag{3.20}$$

$$\mathbf{e}_i(t) = \mathbf{V}\mathbf{w}(t) = \mathbf{x}(t) - \widehat{\mathbf{y}}(t), \tag{3.21}$$

$$\widehat{\mathbf{y}}(t) = \mathbf{y}(t) + \mathbf{e}_c(t) = \mathbf{V}\mathbf{u}(t) + \mathbf{y}_0 = \mathbf{P}(\mathbf{y}(t) - \mathbf{y}_0) + \mathbf{y}_0. \tag{3.22}$$

Note that $\mathbf{u}(t) \in \mathbb{R}^k$, $\mathbf{w}(t) \in \mathbb{R}^k$, $\mathbf{v}(t) \in \mathbb{R}^{n-k}$, and $k = \operatorname{tr}(\mathbf{P})$. Recalling that $\mathbf{P} = \mathbf{V}\mathbf{W}^{\mathrm{T}}$ is a full rank decomposition of a projection matrix $\mathbf{P}$ such that $\mathbf{W}^{\mathrm{T}}\mathbf{V} = \mathbf{I}$, the linear case can be generalized in the following theorem:

Theorem 3.4.1. *Consider solving the initial value problem* (3.1) *using the projection-based reduced order model* (3.2) *in the interval* $[0, T]$. *Let* $\gamma \geq 0$ *be the Lipschitz constant of* $\mathbf{W}^{\mathrm{T}}\mathbf{f}$ *in the directions corresponding to* $\mathcal{N}(\mathbf{P}) = \mathcal{N}(\mathbf{W}^T) = \mathcal{R}(\mathbf{W})^\perp$ *in a region con-*

taining $\mathbf{y}([0,T])$ *and* $\widehat{\mathbf{y}}([0,T])$. *To be precise, suppose*

$$\|\mathbf{W}^T\mathbf{f}(\widehat{\mathbf{y}}(t)+\mathbf{W}_{\perp}\mathbf{v})-\mathbf{W}^T\mathbf{f}(\widehat{\mathbf{y}}(t))\| \leq \gamma\|\mathbf{v}\|$$

for all $(\mathbf{v},t)\in A\subset\mathbb{R}^{n-k}\times[0,T]$, *where the region* A *is such that the associated region* $\bar{A}=\{(\widehat{\mathbf{y}}(t)+\mathbf{W}_{\perp}\mathbf{v},t):(\mathbf{v},t)\in A\}$ *contains* $(\widehat{\mathbf{y}}(t),t)$ *and* $(\mathbf{y}(t),t)$ *for all* t *in* $[0,T]$. *Let* $\mu(\mathbf{W}^T\mathrm{D}\mathbf{f}(\mathbf{y}_0+\mathbf{V}\mathbf{z})\mathbf{V})\leq\bar{\mu}$ *for* $\mathbf{z}\in V\subset\mathbb{R}^k$, *where* $\{\mathbf{u}(t)+\lambda\mathbf{w}(t):t\in[0,T],\lambda\in[0,1]\}$ *is contained in* V, *and* $\mu(\cdot)$ *denotes the logarithmic norm related to the 2-norm.*

The function $\mathbf{e}_i$ *satisfies*

$$\inf\{C\geq 0:\|\mathbf{e}_i(t)\|_\infty\leq C \textit{ a.e. on } [0,T]\ \}=\|\mathbf{e}_i\|_\infty\leq\begin{cases}\varepsilon\gamma\left(\frac{e^{2\bar{\mu}T}-1}{2\bar{\mu}}\right)^{1/2}\|\mathbf{V}\|\|\mathbf{V}_{\perp}^T\|, & \bar{\mu}\neq 0,\\ \varepsilon\gamma T^{1/2}\|\mathbf{V}\|\|\mathbf{V}_{\perp}^T\|, & \bar{\mu}=0,\end{cases} \tag{3.23}$$

and the 2-norm of the function $\mathbf{e}$ *satisfies*

$$\left(\int_0^T\|\mathbf{e}(t)\|^2\,\mathrm{d}t\right)^{1/2}=\|\mathbf{e}\|\leq\begin{cases}\varepsilon\left(1+\gamma\left(\frac{e^{2\bar{\mu}T}-1-2\bar{\mu}T}{4\bar{\mu}^2}\right)^{1/2}\|\mathbf{V}\|\|\mathbf{V}_{\perp}^T\|\right), & \bar{\mu}\neq 0,\\ \varepsilon(1+2^{-1/2}\gamma T\|\mathbf{V}\|\|\mathbf{V}_{\perp}^T\|), & \bar{\mu}=0,\end{cases} \tag{3.24}$$

where

$$\varepsilon=\|\mathbf{e}_c\|=\left(\int_0^T\|\mathbf{e}_c(t)\|^2\,\mathrm{d}t\right)^{1/2}. \tag{3.25}$$

Proof. The proof follows the development of Proposition 4.2 in [165]. Since $\mathbf{e}_i(t)=\mathbf{V}\mathbf{w}(t)$ and $\mathbf{W}^{\mathrm{T}}\mathbf{V}=\mathbf{I}_k$, it follows that $\mathbf{W}^{\mathrm{T}}\mathbf{e}_i(t)=\mathbf{w}(t)$, so

$$\dot{\mathbf{w}}(t) = \mathbf{W}^{\mathrm{T}}\dot{\mathbf{e}}_i(t) = \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}(t)) - \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{x}(t)), \tag{3.26}$$

and

$$\dot{\mathbf{w}}(t) = \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t) + \mathbf{V}\mathbf{w}(t)) - \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t) + \mathbf{W}_{\perp}\mathbf{v}(t)). \tag{3.27}$$

Applying a Taylor expansion for $h > 0$, $\mathbf{w}(t+h) = \mathbf{w}(t) + h\dot{\mathbf{w}}(t) + O(h^2)$, which satisfies

$$\begin{aligned}\|\mathbf{w}(t+h)\| &= \|\mathbf{w}(t) + h\dot{\mathbf{w}}(t) + O(h^2)\|, \\ &= \|\mathbf{w}(t) + h\mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t) + \mathbf{V}\mathbf{w}(t)) - h\mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t) + \mathbf{W}_{\perp}\mathbf{v}(t)) + O(h^2)\|.\end{aligned} \tag{3.28}$$

Using the triangle inequality on the previous equation (3.28) yields

$$\begin{aligned}\|\mathbf{w}(t+h)\| \leq {} & \|\mathbf{w}(t) + h\mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t) + \mathbf{V}\mathbf{w}(t)) - h\mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t))\| \\ & + h\|\mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t) + \mathbf{W}_{\perp}\mathbf{v}(t)) - \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t))\| + O(h^2).\end{aligned} \tag{3.29}$$

Let $\mathbf{g} : \mathbb{R}^k \to \mathbb{R}^k$ be the function

$$\mathbf{g}(\boldsymbol{\eta}) = \boldsymbol{\eta} + h\mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\boldsymbol{\eta}). \tag{3.30}$$

Then

$$\|\mathbf{w}(t) + h\mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t) + \mathbf{V}\mathbf{w}(t)) - h\mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t))\| = \|\mathbf{g}(\mathbf{u}(t) + \mathbf{w}(t)) - \mathbf{g}(\mathbf{u}(t))\|. \tag{3.31}$$

Applying a multivariate mean value theorem (Exercise 2.5 from [52]) to g yields

$$\|\mathbf{g}(\mathbf{u}(t) + \mathbf{w}(t)) - \mathbf{g}(\mathbf{u}(t))\| \leq \kappa\|\mathbf{w}(t)\|, \tag{3.32}$$

for any $\kappa \in \mathbb{R}$ such that

$$\kappa \geq \sup_{\boldsymbol{\eta}\in[\mathbf{u}(t),\mathbf{u}(t)+\mathbf{w}(t)]} \|\mathrm{D}\mathbf{g}(\boldsymbol{\eta})\| = \sup_{\boldsymbol{\eta}\in[\mathbf{u}(t),\mathbf{u}(t)+\mathbf{w}(t)]} \|\mathbf{I}_k + h\mathbf{W}^{\mathrm{T}}\mathrm{D}\mathrm{f}(\mathbf{y}_0 + \mathbf{V}\boldsymbol{\eta})\mathbf{V}\|. \tag{3.33}$$

Here, for any two vectors $\boldsymbol{\eta}_1, \boldsymbol{\eta}_2 \in \mathbb{R}^k$, $[\boldsymbol{\eta}_1, \boldsymbol{\eta}_2]$ denotes the line segment joining the two. (Traditionally, this bracket notation refers to intervals; however, the convention used by Rathinam and Petzold [165] is followed here.) Since the line $[\mathbf{u}(t), \mathbf{u}(t) + \mathbf{w}(t)]$ is a compact subset of $\mathbb{R}^k$,

$$\sup_{\boldsymbol{\eta}\in[\mathbf{u}(t),\mathbf{u}(t)+\mathbf{w}(t)]} \|\mathbf{I}_k + h\mathbf{W}^{\mathrm{T}}\mathrm{D}\mathrm{f}(\mathbf{y}_0 + \mathbf{V}\boldsymbol{\eta})\mathbf{V}\| = \max_{\boldsymbol{\eta}\in[\mathbf{u}(t),\mathbf{u}(t)+\mathbf{w}(t)]} \|\mathbf{I}_k + h\mathbf{W}^{\mathrm{T}}\mathrm{D}\mathrm{f}(\mathbf{y}_0 + \mathbf{V}\boldsymbol{\eta})\mathbf{V}\|.$$

It follows from (3.31), (3.32), (3.33), and (3.29) that

$$\begin{aligned}\|\mathbf{w}(t+h)\| - \|\mathbf{w}(t)\| \leq{}& \left(\max_{\boldsymbol{\eta}\in[\mathbf{u}(t),\mathbf{u}(t)+\mathbf{w}(t)]} \|\mathbf{I}_k + h\mathbf{W}^{\mathrm{T}}\mathrm{D}\mathrm{f}(\mathbf{y}_0 + \mathbf{V}\boldsymbol{\eta})\mathbf{V}\| - 1\right)\|\mathbf{w}(t)\| \\ &+ h\|\mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t) + \mathbf{W}_{\perp}\mathbf{v}(t)) - \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\mathbf{u}(t))\| + O(h^2), \\ \leq{}& \left(\max_{\boldsymbol{\eta}\in[\mathbf{u}(t),\mathbf{u}(t)+\mathbf{w}(t)]} \|\mathbf{I}_k + h\mathbf{W}^{\mathrm{T}}\mathrm{D}\mathrm{f}(\mathbf{y}_0 + \mathbf{V}\boldsymbol{\eta})\mathbf{V}\| - 1\right)\|\mathbf{w}(t)\| \\ &+ h\gamma\|\mathbf{v}(t)\| + O(h^2),\end{aligned} \tag{3.34}$$

which implies that

$$\frac{\|\mathbf{w}(t+h)\| - \|\mathbf{w}(t)\|}{h} \leq \bar{\mu}\|\mathbf{w}(t)\| + \gamma\|\mathbf{v}(t)\| + O(h), \tag{3.35}$$

where the $O(h)$ term may be uniformly bounded independent of $\mathbf{w}(t)$ (see [83], Equations 10.17 and 10.18). Then it follows from Theorem 10.6 of [83] that

$$\|\mathbf{w}(t)\| \leq \gamma \int_0^t e^{\bar{\mu}(t-\tau)}\|\mathbf{v}(\tau)\| \,\mathrm{d}\tau. \tag{3.36}$$

Since $\mathbf{e}_i(t) = \mathbf{V}\mathbf{w}(t)$, it follows that

$$\|\mathbf{e}_i(t)\| \leq \|\mathbf{V}\|\|\mathbf{w}(t)\| \leq \|\mathbf{V}\|\gamma \int_0^t e^{\bar{\mu}(t-\tau)}\|\mathbf{v}(\tau)\| \,\mathrm{d}\tau. \tag{3.37}$$

After applying the Cauchy-Schwarz inequality on the right-hand side, it follows that

$$\|\mathbf{e}_i(t)\| \leq \begin{cases} \|\mathbf{V}\|\gamma\Big(\frac{e^{2\bar{\mu}t}-1}{2\bar{\mu}}\Big)^{1/2}\Big(\int_0^t \|\mathbf{v}(\tau)\|^2 \,\mathrm{d}\tau\Big)^{1/2}, & \bar{\mu} \neq 0, \\ \|\mathbf{V}\|\gamma t^{1/2}\Big(\int_0^t \|\mathbf{v}(\tau)\|^2 \,\mathrm{d}\tau\Big)^{1/2}, & \bar{\mu} = 0. \end{cases} \tag{3.38}$$

Since $\mathbf{v}(t) = -\mathbf{V}_\perp^{\mathrm{T}}\mathbf{e}_c(t)$, it follows that

$$\|\mathbf{e}_i(t)\| \leq \begin{cases} \|\mathbf{V}\|\|\mathbf{V}_\perp^{\mathrm{T}}\|\gamma\Big(\frac{e^{2\bar{\mu}t}-1}{2\bar{\mu}}\Big)^{1/2}\Big(\int_0^t \|\mathbf{e}_c(\tau)\|^2 \,\mathrm{d}\tau\Big)^{1/2}, & \bar{\mu} \neq 0, \\ \|\mathbf{V}\|\|\mathbf{V}_\perp^{\mathrm{T}}\|\gamma t^{1/2}\Big(\int_0^t \|\mathbf{e}_c(\tau)\|^2 \,\mathrm{d}\tau\Big)^{1/2}, & \bar{\mu} = 0, \end{cases} \tag{3.39}$$

from which it follows that

$$\|\mathbf{e}_i\|_\infty \le \begin{cases} \varepsilon\gamma\Big(\frac{e^{2\bar{\mu}t}-1}{2\bar{\mu}}\Big)^{1/2}\|\mathbf{V}\|\|\mathbf{V}_\perp^{\mathrm{T}}\|, & \bar{\mu}\neq 0, \\ \varepsilon\gamma T^{1/2}\|\mathbf{V}\|\|\mathbf{V}_\perp^{\mathrm{T}}\|, & \bar{\mu}=0. \end{cases} \tag{3.40}$$

Substituting (3.25) then squaring (3.39), integrating, and taking the square root to pass to the L^2-norm yields the bound

$$\|\mathbf{e}_i\| \le \begin{cases} \varepsilon\gamma\Big(\frac{e^{2\bar{\mu}T}-1-2\bar{\mu}T}{4\bar{\mu}^2}\Big)^{1/2}\|\mathbf{V}\|\|\mathbf{V}_\perp^{\mathrm{T}}\|, & \bar{\mu}\neq 0, \\ 2^{-1/2}\varepsilon\gamma T\|\mathbf{V}\|\|\mathbf{V}_\perp^{\mathrm{T}}\|, & \bar{\mu}=0. \end{cases} \tag{3.41}$$

Applying the triangle inequality yields

$$\|\mathbf{e}\| \le \|\mathbf{e}_i\| + \|\mathbf{e}_c\| \le \begin{cases} \varepsilon\left(1+\gamma\Big(\frac{e^{2\bar{\mu}T}-1-2\bar{\mu}T}{4\bar{\mu}^2}\Big)^{1/2}\|\mathbf{V}\|\|\mathbf{V}_\perp^{\mathrm{T}}\|\right), & \bar{\mu}\neq 0, \\ \varepsilon(1+2^{-1/2}\gamma T\|\mathbf{V}\|\|\mathbf{V}_\perp^{\mathrm{T}}\|), & \bar{\mu}=0. \end{cases} \tag{3.42}$$

□

Remark 3.4.2. When $\bar{\mu} < 0$, uniform bounds (independent of T) can obtained from Theorem 3.4.1 by noting that

$$\frac{e^{2\bar{\mu}t}-1}{\bar{\mu}} \le \frac{1}{|\bar{\mu}|}, \tag{3.43}$$

in which case

$$\|\mathbf{e}_i\|_\infty \le \varepsilon\gamma|2\bar{\mu}|^{-1/2}\|\mathbf{V}\|\|\mathbf{V}_\perp^{\mathrm{T}}\|, \tag{3.44}$$

$$\|\mathbf{e}\| \le \varepsilon\Big(1+\frac{\gamma}{2|\bar{\mu}|}\|\mathbf{V}\|\|\mathbf{V}_\perp^{\mathrm{T}}\|\Big). \tag{3.45}$$

Remark 3.4.3. It is worth noting that if Theorem 3.4.1 is applied to an orthogonal projector, the resulting bounds are weaker than those derived in [165] because orthogonality is not used in the proof above. Also note that the since full rank decompositions of P and $\mathbf{I}-\mathbf{P}$ are not unique, the error bounds derived in Theorem 3.4.1 are not unique, and depend on these full rank decompositions. Some choice of these decompositions will yield the tightest possible error bounds. However, worst-case error bounds may be derived by changing the approach above slightly.

Write instead the solution $\mathbf{x} : [0,T] \to \mathbb{R}^n$ of (3.2) in terms of the solution $\mathbf{y} : [0,T] \to \mathbb{R}^n$ of (3.1):

$$\mathbf{x}(t) = \mathbf{y}(t) + \mathbf{e}_c(t) + \mathbf{e}_i(t).$$

Write the projected solution $\hat{\mathbf{y}} : [0,T] \to \mathbb{R}^n$ as

$$\hat{\mathbf{y}}(t) = \mathbf{y}(t) + \mathbf{e}_c(t).$$

Then error bounds can be obtained from the following corollary:

Corollary 3.4.4. *Assume the hypotheses of Theorem 3.4.1. Let $\gamma' \geq 0$ be the Lipschitz constant of* $\mathbf{Pf}$ *in the directions corresponding to $\mathcal{N}(\mathbf{P})$ in a region containing $\mathbf{y}([0,T])$ and $\hat{\mathbf{y}}([0,T])$. To be precise, suppose*

$$||\mathbf{Pf}(\hat{\mathbf{y}}(t) + \mathbf{W}_\perp \mathbf{v}) - \mathbf{Pf}(\hat{\mathbf{y}}(t))|| \leq \gamma' ||\mathbf{v}||$$

for all $(\mathbf{v}, t) \in A' \subset \mathbb{R}^{n-k} \times [0,T]$, where the region A' is such that the associated region $\bar{A}' = \{(\hat{\mathbf{y}}(t) + \mathbf{W}_\perp \mathbf{v}, t) : (\mathbf{v}, t) \in A'\}$ contains $(\hat{\mathbf{y}}(t), t)$ and $(\mathbf{y}(t), t)$ for all t in $[0,T]$, and $\mathbf{W}_\perp$ is orthonormal.. Let $\mu(\mathbf{PDf}(\mathbf{z})) \leq \bar{\mu}'$ for $\mathbf{z} \in V' \subset \mathbb{R}^n$, $\{\lambda\hat{\mathbf{y}}(t) + (1-\lambda)\mathbf{x}(t) : t \in [0,T], \lambda \in [0,1]\}$ is contained in V', and $\mu(\cdot)$ denotes the logarithmic norm related to the 2-norm.

The function $\mathbf{e}_i$ satisfies

$$\inf\{C \geq 0 : |\mathbf{e}_i(t)| \leq C \textit{ a.e. on } [0,T]\ \} = \|\mathbf{e}_i\|_\infty \leq \begin{cases} \varepsilon\gamma'\left(\frac{e^{2\bar{\mu}'T}-1}{2\bar{\mu}'}\right)^{1/2}, & \bar{\mu}' \neq 0, \\ \varepsilon\gamma' T^{1/2}, & \bar{\mu}' = 0, \end{cases} \tag{3.46}$$

and the 2-norm of the function e *satisfies*

$$\left(\int_0^T \|\mathbf{e}(t)\|^2 \,\mathrm{d}t\right)^{1/2} = \|\mathbf{e}\| \leq \begin{cases} \varepsilon\left(1 + \gamma'\left(\frac{e^{2\bar{\mu}'T}-1-2\bar{\mu}'T}{4(\bar{\mu}')^2}\right)^{1/2}\right), & \bar{\mu}' \neq 0, \\ \varepsilon(1 + 2^{-1/2}\gamma' T), & \bar{\mu}' = 0. \end{cases} \tag{3.47}$$

Proof. Only a sketch proof will be provided; the proof follows the logic of Theorem 3.4.1, but uses the decomposition

$$\begin{aligned} \|\mathbf{e}_i(t+h)\| &= \|\mathbf{e}_i(t) + h\mathbf{P}\mathbf{f}(\mathbf{y}(t) + \mathbf{e}_c(t) + \mathbf{e}_i(t)) - h\mathbf{P}\mathbf{f}(\mathbf{y}(t))\| + O(h^2) \\ &\leq \|\mathbf{e}_i(t) + h\mathbf{P}\mathbf{f}(\mathbf{y}(t) + \mathbf{e}_c(t) + \mathbf{e}_i(t)) - h\mathbf{P}\mathbf{f}(\mathbf{y}(t) + \mathbf{e}_c)\| \\ &\quad + h\|\mathbf{P}\mathbf{f}(\mathbf{y}(t)) - h\mathbf{P}\mathbf{f}(\mathbf{y}(t) + \mathbf{e}_c)\| + O(h^2) \end{aligned} \tag{3.48}$$

instead of (3.29) to derive bounds. □

Remark 3.4.5. The bounds in Corollary 3.4.4 do not correspond to the bounds in Theorem 3.4.1, because $\bar{\mu}$ and $\bar{\mu}'$ cannot be compared directly, since they bound the logarithmic norm of square matrices of differing size. Coordinate changes cannot be used to relate the logarithmic norm in the hypotheses of Theorem 3.4.1 to the logarithmic norm in the hypotheses of Corollary 3.4.4; these coordinate changes would not be norm-preserving. Bounds corresponding to Corollary 3.4.4 were not considered in [165]; the likely explanation for not considering this approach in that work is that $\|\mathbf{P}\| = 1$ for the case where $\mathbf{P}$ is an orthogonal projector, and $\mathbf{V}$, $\mathbf{W}$, $\mathbf{V}_\perp$, and $\mathbf{W}_\perp$ can all be chosen such that their 2-norms are all one. Corollary

3.4.4 yields worst-case bounds on the error for a projection-based model reduction method, since these bounds are unique, unlike the bounds in Theorem 3.4.1. However, these bounds are generally expected to be weaker than Theorem 3.4.1, because the hypotheses involve taking the logarithmic norm of a larger square matrix, making it more likely in practice to yield a large logarithmic norm bound, compared to Theorem 3.4.1.

Remark 3.4.6. By considering in-subspace errors ($\mathbf{e}_i(t)$, which is in $\mathcal{R}(\mathbf{P})$) and out-of-subspace errors ($\mathbf{e}_c(t)$, which is in $\mathcal{N}(\mathbf{P})$) separately, this analysis provides a bound on the norm of the total error function, $\mathbf{e}$, in terms of ε, a bound on the norm of the out-of-subspace error function, $\mathbf{e}_c$. The value of ε depends on the solution $\mathbf{y} : [0, T] \to \mathbb{R}^n$ of (3.1) and on $\mathbf{P}$ and $\mathbf{y}_0$ in (3.2). In general, $||\mathbf{e}_c||$ is not known precisely unless the solution of (3.1) is calculated; this observation holds even for the analysis of POD in [165]. Typically, a bound on $||\mathbf{e}_c||$ is estimated using any error control results provided by a model reduction method. If such results are unavailable, substituting a known solution to (3.1) with different initial conditions that approximates $\mathbf{y}$ is another way to obtain such an estimate. Using an estimate of ε in Theorem 3.4.1 or Corollary 3.4.4 would only yield estimates of bounds on the function norm of the total error at best; if the function used in place of $\mathbf{y}$ in the definition of (3.14) differs significantly from $\mathbf{y}$, these estimates may be inaccurate. Consequently, such estimates must be used with caution.

3.5 Case Study

To illustrate the factors affecting bounds on the norms of $\mathbf{e}_i$ and $\mathbf{e}$ given $\mathbf{e}_c$ (or bounds on $||\mathbf{e}_c||_2$), consider the linear time invariant ODE

$$\dot{\mathbf{y}}(t) = \mathbf{A}\mathbf{y}(t), \quad \mathbf{y}(0) = \mathbf{y}^*, \tag{3.49}$$

where $\mathbf{A}$ takes the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{A}_2 \end{bmatrix}, \tag{3.50}$$

with the blocks of A first taking the values

$$\mathbf{A}_1 = \begin{bmatrix} -2.1 & 0 & 0 \\ 0 & -2.1732 & -2 \\ 0 & -2 & -2.1732 \end{bmatrix}, \tag{3.51}$$

$$\mathbf{A}_{12} = \begin{bmatrix} 0.3893 & 0.5179 & -1.543 \\ 1.39 & 1.3 & 0.8841 \\ 0.06293 & -0.9078 & -1.184 \end{bmatrix}, \tag{3.52}$$

$$\mathbf{A}_2 = \begin{bmatrix} -3 & 0 & 0 \\ 0 & -3.226 & -0.708 \\ 0 & -0.708 & -3.226 \end{bmatrix}, \tag{3.53}$$

so that $n = 6$. Three values of A were considered; modifications of (3.51), (3.52), and (3.53) will be discussed later in this section. This example is inspired by [165]; the matrix A in this paper differs from the matrix A in that paper by negating all positive entries of $\mathbf{A}_1$ and $\mathbf{A}_2$ in [165], then subtracting 2I from the resulting matrix. These manipulations can be seen in the accompanying MATLAB [133] and Python [209] code in Appendix B.

Unlike the case of orthogonal projectors, it is impossible to use a norm-preserving change of basis to decouple the effects of ε, γ, and $\bar{\mu}$, because $\mathcal{R}(\mathbf{P})$ is *not* orthogonal to $\mathcal{N}(\mathbf{P})$ when P is an oblique projector. Consequently, when changing one of $\{\varepsilon, \gamma, \bar{\mu}\}$, at least one other parameter changes. In light of this observation, the effect of changing γ and ε at constant $\bar{\mu}$, and the effect of changing $\bar{\mu}$ and ε at constant γ were studied by altering the blocks of A. The parameters γ and $\bar{\mu}$ were calculated using $\gamma = \|\mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{W}_{\perp}\|_2$ and $\bar{\mu} = \mu(\mathbf{W}^{\mathrm{T}}\mathbf{A}\mathbf{V})$; both formulates derive from

Theorem 3.4.1. The parameters γ' and $\bar{\mu}'$ were calculated using $\gamma' = \|\mathbf{PAW}_{\perp}\|_2$ and $\bar{\mu}' = \mu(\mathbf{PA})$, where $\mathbf{W}_{\perp}$ is orthonormal. The parameter ε and all other function 2-norms were calculated by approximating definite integrals using the trapezoidal rule.

Note that $\mathbf{A}_1$ and $\mathbf{A}_2$ in (3.51) and (3.53), respectively, are symmetric and have negative real eigenvalues. Furthermore, the spectrum of $\mathbf{A}$ is the union of the spectra of $\mathbf{A}_1$ and $\mathbf{A}_2$. For each value of $\mathbf{A}$ considered, the projector $\mathbf{P}$ is chosen so that $\mathcal{R}(\mathbf{P})$ consists of the three eigenvectors of $\mathbf{A}$ that are *not* contained in the span of $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$, where $\mathbf{e}_j$ is the jth standard Euclidean unit vector; consequently, the number of reduced variables is $k = 3$. However, for all three values of $\mathbf{A}$ considered, $\mathcal{N}(\mathbf{P})$ will always be spanned by $\{\mathbf{e}_4, \mathbf{e}_5, \mathbf{e}_6\}$, where again, $\mathbf{e}_j$ is the jth standard Euclidean unit vector. As a result, when $\mathbf{A}$ changes, the projector $\mathbf{P}$ will change with it, because the range of the projector changes, even though the null space of the projector will stay the same.

To illustrate the effect of changing $\mathbf{V}$, $\mathbf{W}$, $\mathbf{V}_{\perp}$, and $\mathbf{W}_{\perp}$ on the error bounds given by Theorem 3.4.1, two sets of values for these matrices will be considered. In the first set of values, $\mathbf{W}$ and $\mathbf{W}_{\perp}$ will have orthonormal columns so that $\|\mathbf{W}\|_2 = \|\mathbf{W}_{\perp}\|_2 = 1$; the remaining matrices are determined so that they satisfy (3.8) and (3.17). In the second set of values, $\mathbf{V}$ and $\mathbf{V}_{\perp}$ have orthonormal columns so that $\|\mathbf{V}\|_2 = \|\mathbf{V}_{\perp}\|_2 = 1$; the remaining matrices are determined so that they satisfy (3.8) and (3.17). The error bounds obtained from Theorem 3.4.1 were compared to error bounds obtained from Corollary 3.4.4.

The matrices $\mathbf{V}$, $\mathbf{W}$, $\mathbf{V}_{\perp}$, and $\mathbf{W}_{\perp}$ were calculated from matrices whose columns spanned $\mathcal{R}(\mathbf{P})$ and $\mathcal{N}(\mathbf{P})$, respectively, using the algorithm suggested in equations (5.1) and (5.2) of [195]. Projector-vector products were also calculated using this algorithm. While explicit calculation of $\mathbf{P}$ is *not* recommended [195], it was calculated explicitly only to calculate $\|\mathbf{P}\|_2$ to get an idea of numerical errors due to projector-vector products. The quantity $\|\mathbf{P}\|_2$ plays a role in the calculation of projector-vector products similar to that of the condition number when solving systems of linear equations [195]. All ODEs were integrated using an explicit

Runge-Kutta (4,5) Dormand-Prince pair using a relative tolerance of 10^{-13} and an absolute tolerance of 10^{-25} on each component of the solution. All linear systems were solved using QR factorization. All random numbers were calculated using a Mersenne twister algorithm (MT19937). Calculations were implemented on a MacBook Pro 2011 model running Mac OS X 10.7.3 with a 2.7 GHz Intel Core i7 CPU and 8 GB of 1333 MHz DDR3 RAM. Source code implementations in MATLAB [133] and Python [209] are included for reproducibility in Appendix B.

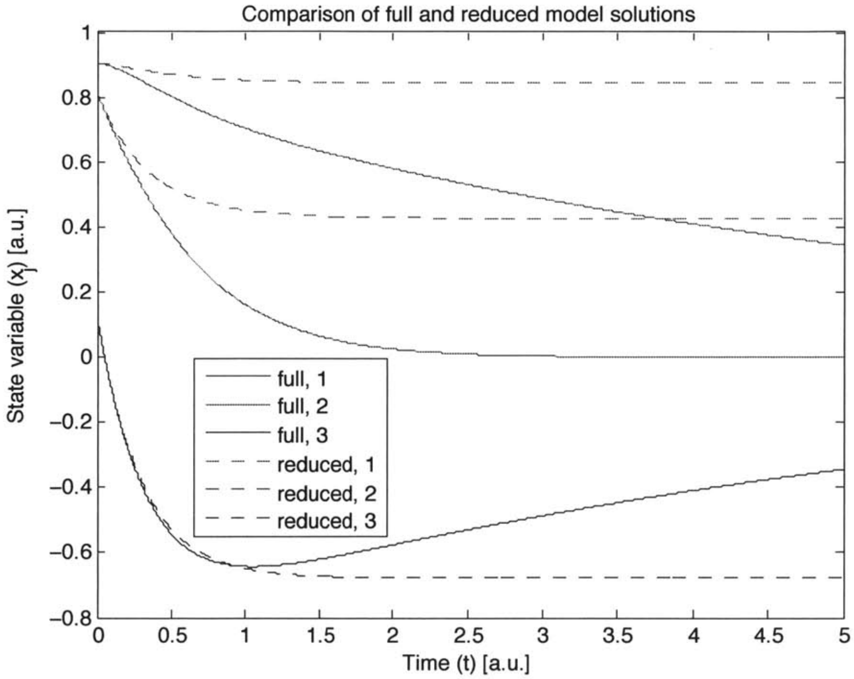

Figure 3-2: First three components of $\mathbf{x}(t)$ (dashed) and $\mathbf{y}(t)$ (solid) corresponding to the first choice of $\mathbf{A}$ as in (3.49), (3.50), (3.51), (3.52), and (3.53) and its corresponding projector.

For $\mathbf{A}$ as in (3.49), (3.50), (3.51), (3.52), and (3.53), if $\mathbf{V}$, $\mathbf{W}$, $\mathbf{V}_{\perp}$, and $\mathbf{W}_{\perp}$ are chosen such that $\mathbf{W}$ and $\mathbf{W}_{\perp}$ are orthonormal, then $\gamma = \|\mathbf{A}_{12}\|_2 = 2.4421$ and $\bar{\mu} = -2.1323$, based on Theorem 3.4.1. The initial condition $\mathbf{y}^*$ for (3.49) was randomly chosen, and $\mathbf{y}(t)$ and $\mathbf{x}(t)$ were computed on the interval $[0, 5]$, as inspired by [165]. The first three components of $\mathbf{y}(t)$ and $\mathbf{x}(t)$ are shown in Figure 3-2, and the second three components of $\mathbf{y}(t)$ and $\mathbf{x}(t)$ are shown in Figure 3-3. The solutions $\mathbf{y}$ and $\mathbf{x}$ each behave qualitatively similarly for all three values of $\mathbf{A}$ considered in this case

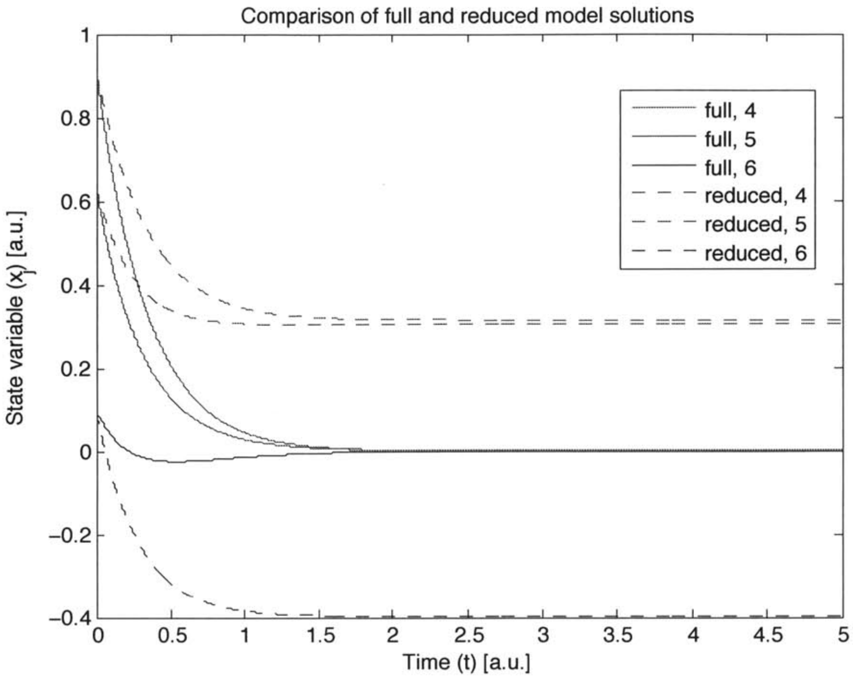

Figure 3-3: Second three components of $\mathbf{x}(t)$ (dashed) and $\mathbf{y}(t)$ (solid) corresponding to the first choice of $\mathbf{A}$ as in (3.49), (3.50), (3.51), (3.52), and (3.53) and its corresponding projector.

study, and will not be plotted for other values of $\mathbf{A}$. The first three components of $\mathbf{e}_i(t)$ and $\mathbf{e}(t)$ are shown in Figure 3-4; note that $e_j(t) = e_{ij}(t)$ for $j = 1, 2, 3$ and all t on $[0, 5]$. The second three components of $\mathbf{e}_i(t)$ and $\mathbf{e}(t)$ are shown in Figure 3-5. As with $\mathbf{y}(t)$ and $\mathbf{x}(t)$, the quantities $\mathbf{e}_i(t)$ and $\mathbf{e}(t)$ each behave qualitatively similarly for all three values of $\mathbf{A}$ considered; no additional plots of $\mathbf{e}_i(t)$ or $\mathbf{e}(t)$ will be presented. The 2-norm of the component of the error in $\mathcal{N}(\mathbf{P})$ was $\varepsilon = \|\mathbf{e}_c\|_2 = 2.3423$. The sup-norm and 2-norm of the component of the error in $\mathcal{R}(\mathbf{P})$ were $\|\mathbf{e}_i\|_\infty = 1.1659$ and $\|\mathbf{e}_i\|_2 = 2.3950$. The 2-norm of the total error was $\|\mathbf{e}\|_2 = 1.7180$. The bounds provided by Theorem 3.4.1 when $\mathbf{W}$ and $\mathbf{W}_\perp$ are orthonormal are $\|\mathbf{e}_i\|_\infty \leq 1.9937 \cdot 10^1$, $\|\mathbf{e}_i\|_2 \leq 4.3524 \cdot 10^1$, and $\|\mathbf{e}\|_2 \leq 4.5866 \cdot 10^1$. When $\mathbf{V}$ and $\mathbf{V}_\perp$ are orthonormal, $\gamma = 1.0727 \cdot 10^1$ and $\bar{\mu} = -2.4588$, illustrating that choice of $\mathbf{V}$, $\mathbf{W}$, $\mathbf{V}_\perp$, and $\mathbf{W}_\perp$ does affect the error bound provided by Theorem 3.4.1. Using Corollary 3.4.4, $\gamma' = 6.1365$ and $\bar{\mu}' = 3.0048$; the positive value of $\bar{\mu}'$ indicates that the error bounds provided by Corollary 3.4.4 would be much worse than the error bounds

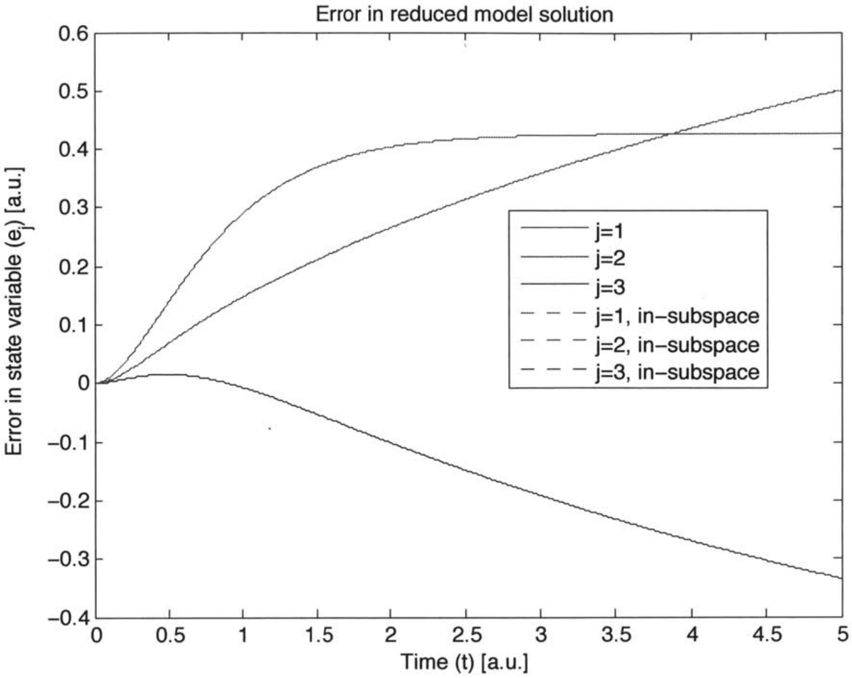

Figure 3-4: First three components of $\mathbf{e}_i(t)$ (dashed) and $\mathbf{e}(t)$ (solid) corresponding to the first choice of $\mathbf{A}$ as in (3.49), (3.50), (3.51), (3.52), and (3.53) and its corresponding projector. Note that for this value of $\mathbf{A}$, the first three components of $\mathbf{e}_i(t)$ and $\mathbf{e}(t)$ are virtually equal.

provided by Theorem 3.4.1 using either choice of the matrices $\mathbf{V}$, $\mathbf{W}$, $\mathbf{V}_\perp$, and $\mathbf{W}_\perp$ above. The projector for this choice of $\mathbf{A}$ had 2-norm $\|\mathbf{P}\|_2 = 2.6829$, indicating that numerical errors due to projector-vector multiplication will be small.

The second choice of $\mathbf{A}$ was to scale the value of $\mathbf{A}_{12}$ in (3.52) by one-half, keeping $\mathbf{A}_1$ and $\mathbf{A}_2$ the same (as in (3.51) and (3.53)). As a result, γ decreases, but $\bar{\mu}$ stays the same and ε increases. The initial condition and time interval of integration were kept the same. Recall that $\mathcal{R}(\mathbf{P})$ changes in response to changes in $\mathbf{A}$, but $\mathcal{N}(\mathbf{P})$ stays the same. If $\mathbf{V}$, $\mathbf{W}$, $\mathbf{V}_\perp$, and $\mathbf{W}_\perp$ are chosen such that $\mathbf{W}$ and $\mathbf{W}_\perp$ are orthonormal, then $\gamma = \|\mathbf{A}_{12}\| = 1.2210$ and $\bar{\mu} = -2.1323$, based on Theorem 3.4.1. The 2-norm of the component of the error in $\mathcal{N}(\mathbf{P})$ was $\varepsilon = \|\mathbf{e}_c\|_2 = 5.3596$. The sup-norm and 2-nrom of the component of the error in $\mathcal{R}(\mathbf{P})$ were $\|\mathbf{e}_i\|_\infty = 1.5754$ and $\|\mathbf{e}_i\|_2 = 2.8112$. The 2-norm of the total error was $\|\mathbf{e}\|_2 = 3.5945$. The error bounds provided by Theorem 3.4.1 when $\mathbf{W}$ and $\mathbf{W}_\perp$ are orthonormal were $\|\mathbf{e}_i\|_\infty \leq 8.1735 \cdot 10^1$, $\|\mathbf{e}_i\|_2 \leq 1.7843 \cdot 10^2$, and $\|\mathbf{e}\|_2 \leq 1.8379 \cdot 10^2$. When $\mathbf{V}$ and

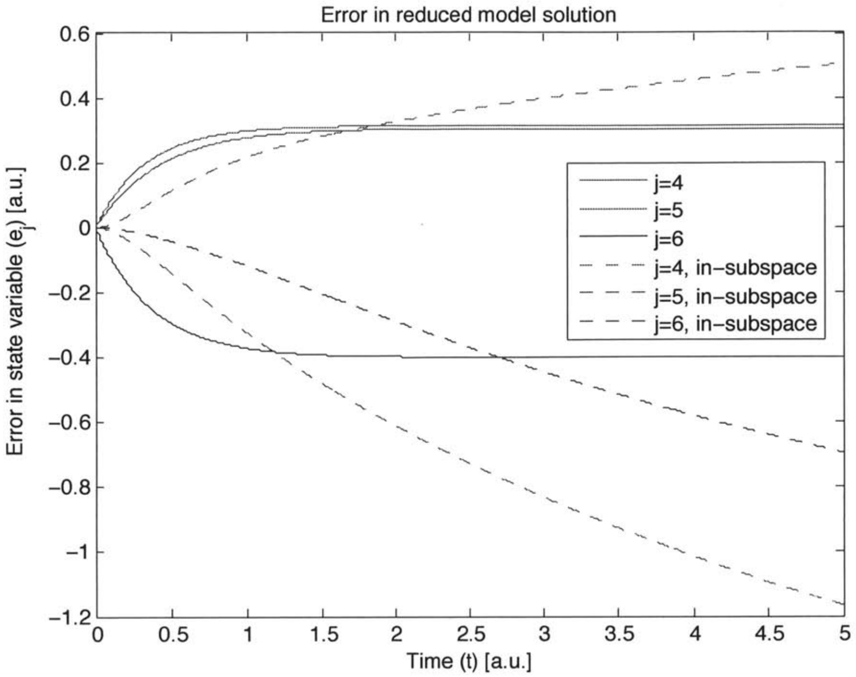

Figure 3-5: Second three components of $\mathbf{e}_i(t)$ (dashed) and $\mathbf{e}(t)$ (solid) corresponding to the first choice of $\mathbf{A}$ as in (3.49), (3.50), (3.51), (3.52), and (3.53) and its corresponding projector.

$\mathbf{V}_\perp$ are orthonormal, $\gamma = 1.7260 \cdot 10^1$ and $\bar{\mu} = -2.5041$, according to Theorem 3.4.1. Using Corollary 3.4.4, $\gamma' = 5.7798$ and $\bar{\mu}' = 5.8023$. This projector for this choice of $\mathbf{A}$ had 2-norm $\|\mathbf{P}\|_2 = 5.0786$, indicating again that numerical errors due to projector-vector multiplication will be small.

The third and final choice of $\mathbf{A}$ was to keep $\mathbf{A}_1$ and $\mathbf{A}_{12}$ as in (3.51) and (3.52), and scale the value of $\mathbf{A}_2$ in (3.53) by .715, so that γ stays the same and both $\bar{\mu}$ and ε increase, relative to the first choice of $\mathbf{A}$. If $\mathbf{V}$, $\mathbf{W}$, $\mathbf{V}_\perp$, and $\mathbf{W}_\perp$ are chosen such that $\mathbf{W}$ and $\mathbf{W}_\perp$ are orthonormal, then $\gamma = \|\mathbf{A}_{12}\|_2 = 2.4421$ and $\bar{\mu} = 4.8194 \cdot 10^{-1}$, based on Theorem 3.4.1. The 2-norm of the component of the error in $\mathcal{N}(\mathbf{P})$ was $\varepsilon = \|\mathbf{e}_c\|_2 = 3.4998$. The sup-norm and 2-norm of the component of the error in $\mathcal{R}(\mathbf{P})$ were $\|\mathbf{e}_i\|_\infty = 1.6444$ and $\|\mathbf{e}_i\|_2 = 3.7803$; the 2-norm of the total error is $\|\mathbf{e}\|_2 = 5.5926$. The error bounds provided by Theorem 3.4.1 when $\mathbf{W}$ and $\mathbf{W}_\perp$ are orthonormal are $\|\mathbf{e}_i\|_\infty \leq 9.9526 \cdot 10^2$, $\|\mathbf{e}_i\|_2 \leq 9.9366 \cdot 10^2$, and $\|\mathbf{e}\|_2 \leq 9.9716 \cdot 10^2$. According to Theorem 3.4.1, when $\mathbf{V}$ and $\mathbf{V}_\perp$ are orthonormal, $\gamma = 9.3634$ and $\bar{\mu} =$

-1.2220. Using Corollary 3.4.4, $\gamma' = 5.0389$ and $\bar{\mu} = 5.9639$; here, $||\mathbf{P}||_2 = 3.2114$.

3.6 Discussion

From the error results in Theorem 3.4.1 and Corollary 3.4.4, inferences can be made about how choices of $\mathbf{P}$, $\mathbf{V}$, $\mathbf{W}$, $\mathbf{V}_\perp$, and $\mathbf{W}_\perp$ affect the error bounds, as well as the error, to a first approximation. Discussion will focus on Theorem 3.4.1, though similar observations also apply to Corollary 3.4.4.

The three primary factors influencing bounds on the error due to model reduction are the parameters ε, γ, and $\bar{\mu}$. As noted earlier, ε is controlled by a model reduction method in the ideal case; if the reduced model is chosen well, using tight error tolerances, ε will be small.

To interpret γ and $\bar{\mu}$, it will be useful to introduce some additional mathematical background. Let $\mathbf{q} : S \subset X \to X$ be a nonlinear map, where $X \subset \mathbb{R}^n$. The least upper bound (lub) Lipschitz constant of $\mathbf{q}$, $L(\mathbf{q})$, is defined in [192] as

$$L(\mathbf{q}) = \sup_{\mathbf{u},\mathbf{v} \in S, \mathbf{u} \neq \mathbf{v}} \frac{||\mathbf{q}(\mathbf{u}) - \mathbf{q}(\mathbf{v})||}{||\mathbf{u} - \mathbf{v}||}. \tag{3.54}$$

It has the property that

$$||\mathbf{q}(\mathbf{u}) - \mathbf{q}(\mathbf{v})|| \leq L(\mathbf{q}) \cdot ||\mathbf{u} - \mathbf{v}||, \forall \mathbf{u}, \mathbf{v} \in S, \tag{3.55}$$

and if S is convex and $\mathbf{q}$ is differentiable, then

$$L(\mathbf{q}) = \sup_{\mathbf{u} \in S} ||\mathrm{D}\mathbf{q}(\mathbf{u})||. \tag{3.56}$$

In the hypotheses of Theorem 3.4.1, define the function $\mathbf{g} : \mathbb{R}^{n-k} \times \mathbb{R}^n \to \mathbb{R}^k$ by

$$\mathbf{g}(\mathbf{v}, \mathbf{z}) = \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{z} + \mathbf{W}_{\perp}\mathbf{v}). \tag{3.57}$$

If the set A is convex, then the best value of the Lipschitz constant γ is $\sup\{L(\mathbf{g}(\cdot, \mathbf{z})) : \mathbf{z} \in \widehat{\mathbf{y}}([0, T])\}$.

The extension of the logarithmic norm to nonlinear maps will also be useful for discussion. The least upper bound logarithmic Lipschitz constant of $\mathbf{q}$, $M(\mathbf{q})$, is defined in [192] by

$$M(\mathbf{q}) = \lim_{h \to 0^{+}} \frac{L(\mathbf{I} + h\mathrm{D}\mathbf{q}) - 1}{h}. \tag{3.58}$$

The lub logarithmic Lipschitz constant generalizes the logarithmic norm; for a square matrix $\mathbf{A}$, $M(\mathbf{A}) = \mu(\mathbf{A})$. Furthermore, if $\mathbf{q}$ is differentiable, and S is convex,

$$M(\mathbf{q}) = \sup_{\mathbf{u} \in S} \mu(\mathrm{D}\mathbf{q}(\mathbf{u})). \tag{3.59}$$

In the hypotheses of Theorem 3.4.1, if the set V is convex, then the best value of the bound $\bar{\mu}$ on the logarithmic norm is $M(\mathbf{h})$, where $\mathbf{h} : \mathbb{R}^{k} \to \mathbb{R}^{k}$ is defined by

$$\mathbf{h}(\widetilde{\mathbf{y}}) = \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_{0} + \mathbf{V}\widetilde{\mathbf{y}}). \tag{3.60}$$

The concepts of lub Lipschitz constant and lub logarithmic Lipschitz constant can be used to a prove a result like Theorem 3.4.1; such an approach was taken in [29] to prove error bounds on POD-DEIM reduced models, and would yield bounds that are essentially the same as those in Theorem 3.4.1. Their utility here is in interpreting the meaning of the bounds stated in Theorem 3.4.1.

Let $e^{t\mathbf{h}} : \widetilde{\mathbf{y}}(0) \mapsto \widetilde{\mathbf{y}}(t)$ be the flow corresponding to the differential equation

$$\dot{\widetilde{\mathbf{y}}}(t) = \mathbf{h}(\widetilde{\mathbf{y}}(t)) = \mathbf{W}^{\mathrm{T}}\mathbf{f}(\mathbf{y}_0 + \mathbf{V}\widetilde{\mathbf{y}}(t)), \tag{3.61}$$

as in (3.10). Then it can be shown [192] that

$$L(e^{t\mathbf{h}}) \leq e^{tM(\mathbf{h})} \leq e^{t\bar{\mu}}. \tag{3.62}$$

In other words, $\bar{\mu}$ corresponds to the maximum rate of change of the solution to (3.10) and (3.61). In the ideal case, if a reduced model is chosen well, k (*i.e.*, the rank of the projection matrix, or the number of variables in a Petrov-Galerkin representation of a reduced model [156]) will be small, and the solution $\widetilde{\mathbf{y}} : [0, T] \to \mathbb{R}^k$ will change slowly with time, enabling cheap numerical integration of (3.10). Consequently, in the ideal case, $\bar{\mu}$ will be small.

The constant $\gamma \geq \sup\{L(\mathbf{g}(\cdot, \mathbf{v})) : \mathbf{v} \in \widehat{\mathbf{y}}([0, T])\}$ corresponds to the maximum rate of change of the right-hand side of the lumped system (3.10) in the directions corresponding to $\mathcal{N}(\mathbf{P}) = \mathcal{R}(\mathbf{W})^{\perp} = \mathcal{R}(\mathbf{W}_{\perp})$. In other words, it is the maximum rate of change of the right-hand side of (3.10) in directions neglected by the reduced model. When (3.1) is a stiff system, certain directions in state space are associated (locally, in the case of nonlinear systems) with fast rates of change in the solution $\mathbf{y} : [0, T] \to \mathbb{R}^n$ and also fast rates of change in the time derivative of $\mathbf{y}$. These directions associated with fast changes are usually candidates for inclusion in $\mathcal{N}(\mathbf{P})$ so that solution of the resulting lumped system changes slowly; rapid transients are usually neglected. Consequently, in stiff systems, if $\bar{\mu}$ is small because a good reduced model is chosen by cleverly choosing $\mathbf{V}$ and $\mathbf{W}$, γ is likely to be large. One would expect some tradeoff between $\bar{\mu}$ and γ in stiff systems, depending on the choice of reduced model (*i.e.*, choice of $\mathbf{V}$ and $\mathbf{W}$); if directions corresponding to fast changes are included in $\mathcal{R}(\mathbf{P})$, $\bar{\mu}$ is likely to be large, and γ is likely to decrease. As a result, the bounds in Theorem 3.4.1 are expected to overestimate the error, especially for large values of T; in all likelihood, for stiff systems, they will drastically overestimate the error due to model reduction, but do provide some insight into factors affecting error. For researchers designing reduced models, the key insight is to choose $\mathbf{V}$ and $\mathbf{W}$ to minimize $\bar{\mu}$; γ is a secondary consideration. Both the overestimation of error and insight into factors affecting error are in keeping with the use of the logarithmic norm to bound the error due to numerical integration of ODEs; the logarithmic norm yields very large bounds for stiff systems, but these bounds are in terms of the step size h. Typically, h can be chosen sufficiently small so that stability and error criteria are satisfied, yielding useful bounds for short times only.

The remaining parameters in the error bounds in Theorem 3.4.1 have less influence on the error bounds. There is some freedom in choosing the end time, depending on the needs of the user, but the exponential dependence of the bounds on time suggests that the result will over estimate the error for long times if $\bar{\mu} > 0$, which includes many applications of interest. (It is worth noting that $\bar{\mu} > 0$ does not imply that an ODE is unstable, unless the right-hand side of an ODE has a symmetric Jacobian matrix. (Linear) Stability of an ODE system is dictated by the eigenvalues of the Jacobian matrix of its right-hand side.) Since full rank decompositions of projection matrices are not unique, there is freedom in choosing $\mathbf{V}$ and $\mathbf{V}_{\perp}$; for numerical calculations with the Petrov-Galerkin (lumped) representation, the product $||\mathbf{V}||\,||\mathbf{W}||$ should be as close to $||\mathbf{P}||$ as possible [195]. For a thorough discussion of the numerical analysis associated with calculating oblique projection matrices, as well as oblique projector-vector products, consult [195]. Note that $||\mathbf{P}|| = 1$ if and only if $\mathbf{P}$ is an orthogonal projector. If $\mathbf{P}$ is an oblique projector, $||\mathbf{P}|| > 1$. Choices of $\mathbf{V}$, $\mathbf{V}_{\perp}$, $\mathbf{W}$, and $\mathbf{W}_{\perp}$ will also affect γ and $\bar{\mu}$ indirectly.

3.7 Conclusions and Future Work

In this work, state space error bounds for projection-based model reduction methods were derived in the nonlinear ODE setting in terms of the function norm of the projection error. These are the first such bounds for oblique projection of nonlinear ODE right-hand sides. When the function norm of the projection error is not known precisely, the bounding result yields an estimate of the state space error. The analysis also yields insight into what factors influence the error in the reduced model solution, and indicates that one benefit of using an orthogonal projector is that stronger error bounds may be derived using the previous work by [165]. Finally, these error bounds demonstrate that local error control implies global error control for projection-based model reduction.

However, it is difficult to calculate estimates of state space error bounds using this result, and the resulting bounds will not be strong in general. To facilitate calculation of stronger estimates of error bounds, the small sample statistical condition estimator (SCE) error estimation method developed by [86] may be extended from orthogonal projectors to oblique projectors using the analysis above. These error estimates are easier to calculate, should yield better results, and should provide users of reduced order models with additional information regarding the accuracy and validity of their reduced model approximations.

Chapter 4

State-Space Error Bounds For All Reduced Model ODEs

4.1 Introduction

Model reduction is used in a number of contexts, including fluid mechanics [14, 98, 128, 109, 129], control theory [101], atmospheric modeling [59, 48, 193], combustion modeling [103, 104, 124, 188, 206], circuit simulation [20, 169, 170], and others, both to reduce the computational requirements of computationally demanding simulations and to analyze and interpret physical models. In order to be used to generate quantitatively accurate approximations for mission-critical applications, accurate bounds on or estimates of the approximation error due to model reduction are necessary.

Currently, bounds on the approximation error exist only for projection-based methods [165, 155] and for the non-projection-based method POD-DEIM [30, 29]. These error bounds are based on logarithmic norms of the Jacobian matrix of the ODE right-hand side and have their theoretical roots in Gronwall's inequality [77] and the seminal work on bounding the norms of solutions of ODEs by Dahlquist [38]. Although *a priori* bounds of this type have not been strong in general for the numerical solution of ODEs [11, 83, 192] or model reduction of ODEs, they have been used to develop much more accurate *a posteriori* estimates of the error in both contexts (for work on errors in the numerical solutions of ODEs, see [218, 190, 57, 25]; for work on errors in model reduction, see [86, 178, 148, 217, 82, 80, 81, 79, 99]).

However, for non-projection-based methods like manifold learning methods (such as Isomap [202], locally linear embedding [177], diffusion maps [37, 143, 36], and others [181, 141, 100, 149, 106, 214, 49]), and methods that exploit application-specific problem structure (for instance, in combustion chemistry, reaction elimination [160, 5, 53, 18, 153] or simultaneous reaction and species elimination [137]), neither bounds of any kind nor estimates of the approximation error exist for explicit ODEs with nonlinear right-hand sides. The most closely related work is by Serban, *et al.* [185], which estimates approximation errors due to a *combination* of model reduction and perturbation of parameters. In this work, the approach in [165] and [155] is extended to include *all* model reduction methods. As in [155], although the bounds developed will not be strong, similar to the bounds on the norms of solutions of ODEs by Gronwall [77] and by Dahlquist [38], they can be used as inspiration for future work on *a posteriori* error estimation in model reduction.

4.2 Model Reduction

Here, model reduction will be discussed in the ODE setting. Consider the initial value problem

$$\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t)), \quad \mathbf{y}(0) = \mathbf{y}^*, \tag{4.1}$$

where $\mathbf{y}(t) \in \mathbb{R}^n$ represents system state variables, $\mathbf{y}^* \in \mathbb{R}^n$, and $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ with $\mathbf{f} \in \mathcal{C}^1$. From (4.1), a model reduction method constructs a reduced model

$$\dot{\mathbf{x}}(t) = \widehat{\mathbf{f}}(\mathbf{x}(t)), \quad \mathbf{x}(0) = \mathbf{x}^*, \tag{4.2}$$

where $\widehat{\mathbf{f}} : \mathbb{R}^n \to \mathbb{R}^n$ approximates $\mathbf{f}$, $\mathbf{x}(t) \in \mathbb{R}^n$ represents the state variables of the reduced model, and $\mathbf{x}^* \in \mathbb{R}^n$.

One popular class of methods for constructing $\widehat{\mathbf{f}}$ from $\mathbf{f}$ uses projection, of which there are several representatives [165, 30, 8, 7, 14, 32, 29, 34, 74, 75, 87, 103, 104, 112, 114, 113, 211, 219]. Existing theory can be used to calculate a priori state bounds on reduced models constructed with these methods [165, 155]. Here, the focus is on methods that are *not* projection-based.

A common approach is to neglect small terms in $\mathbf{f}$. One such example is reaction elimination [18, 160, 53, 153, 5], used in combustion applications. Deleting terms from $\mathbf{f}$ avoids the need to estimate parameters associated with those terms. In chemical kinetics, estimating reaction rate parameters from experiment or quantum mechanics calculations often takes more effort than solving the ODEs that use those parameters. In the isothermal, isobaric batch reactor case, chosen for simplicity, $\mathbf{f}$ takes the form

$$\mathbf{f}(\mathbf{y}) = \frac{\mathbf{MNr}(\mathbf{y})}{\rho(\mathbf{y})}, \tag{4.3}$$

where in (4.3), $\mathbf{M} \in \mathbb{R}_{+}^{n \times n}$ is a diagonal matrix of species molecular weights, $\mathbf{N} \in \mathbb{R}^{n \times m}$ is the stoichiometry matrix for the reaction mechanism, $\mathbf{r} : \mathbb{R}^n \to \mathbb{R}^m$ is a function that returns the rates of all m reactions in the reaction mechanism at the system temperature and pressure, $\rho : \mathbb{R}^n \to \mathbb{R}$ is a function that returns the mass density of the system at the system temperature and pressure, and $\mathbf{y} \in \mathbb{R}^n$ is a vector of species mass fractions. Reaction elimination constructs $\widehat{\mathbf{f}}$ from $\mathbf{f}$ in (4.3) by calculating a diagonal matrix $\mathbf{Z} \in \{0, 1\}^{m \times m}$ satisfying certain error constraints. Then

$$\widehat{\mathbf{f}}(\mathbf{y}) = \frac{\mathbf{MNZr}(\mathbf{y})}{\rho(\mathbf{y})}; \tag{4.4}$$

the reactions corresponding to zeros on the diagonal of $\mathbf{Z}$ need not be computed. It can be seen that there is no way to represent this transformation using projection. Other examples are simultaneous reaction and species elimination [137] and skeletal mechanism generation ([121] is one representative example). Again, using (4.3) as a starting point, $\widehat{\mathbf{f}}$ is constructed from $\mathbf{f}$ by calculating two diagonal matrices, $\mathbf{Z} \in \{0,1\}^{m \times m}$ (that eliminates reactions) and $\mathbf{W} \in \{0,1\}^{n \times n}$ (that renders species nonreactive), together satisfying error constraints imposed by the method. Then

$$\widehat{\mathbf{f}}(\mathbf{y}) = \frac{\mathbf{WMNZr}(\mathbf{y})}{\rho(\mathbf{y})}; \tag{4.5}$$

again, there is no way to carry out this transformation using projection only.

In order to calculate *a priori* state bounds on reduced models constructed using these methods (and other non-projection-based model reduction methods), the existing state bounding theory for projection-based reduced models [165, 155] must be extended.

4.3 Mathematical Preliminaries

To bound the state space error in non-projection-based model reduction, the approach of this paper will be to bound the norm of a solution to a nonlinear ODE. Following the presentations of [165] and [155], consider the linear system

$$\dot{\mathbf{y}}(t) = \mathbf{Ay}(t) + \mathbf{r}(t), \quad \mathbf{y}(0) = \mathbf{y}^*, \tag{4.6}$$

for the purpose of illustration, where $\mathbf{A} \in \mathbb{R}^{n \times n}$. The solution of (4.6) takes the form

$$\mathbf{y}(t) = e^{\mathbf{A}t}\mathbf{y}^* + \int_0^t e^{\mathbf{A}(t-\tau)}\mathbf{r}(\tau)\, d\tau. \tag{4.7}$$

From (4.7), bounds on the norm of $\mathbf{y}(t)$ may be derived using Gronwall's lemma [77] or Dahlquist-like inequalities involving the logarithmic norm of $\mathbf{A}$ [83, 192]. Following the approach of [165], bounds on the norm of the function $\mathbf{y} : [0, T] \to \mathbb{R}^n$ are derived instead, where $T > 0$. In this paper, for any function $\mathbf{g} : [0, T] \to \mathbb{R}^n$, $||\mathbf{g}(t)||$ is the norm of the point $\mathbf{g}(t) \in \mathbb{R}^n$, assumed to be the 2-norm unless otherwise stated. The function norm will be denoted $||\mathbf{g}||$ and will also be the 2-norm unless otherwise stated. Keeping function norms in mind, (4.7) may be written as

$$\mathbf{y} = \mathbf{F}(T, \mathbf{A})\mathbf{r} + \mathbf{G}(T, \mathbf{A})\mathbf{y}^*,$$

where $\mathbf{F}(T, \mathbf{A}) : L^2([0, T], \mathbb{R}^n) \to L^2([0, T], \mathbb{R}^n)$ and $\mathbf{G}(T, \mathbf{A}) : \mathbb{R}^n \to L^2([0, T], \mathbb{R}^n)$ are linear operators. The desired bound on $||\mathbf{y}||$ then takes the form

$$||\mathbf{y}|| \leq ||\mathbf{F}(T, \mathbf{A})||\,||\mathbf{r}|| + ||\mathbf{G}(T, \mathbf{A})||\,||\mathbf{y}_0||. \tag{4.8}$$

Sharp estimates for the operator norms of $\mathbf{F}(T, \mathbf{A})$ and $\mathbf{G}(T, \mathbf{A})$ are difficult to obtain. As can be seen from the form of (4.7), these estimates reduce to estimating the norm of the matrix exponential. The classical approach to this problem [192] yields

$$||e^{t\mathbf{A}}|| \leq e^{t\mu(\mathbf{A})}, \quad t \geq 0,$$

where $\mu(\mathbf{A})$ is the logarithmic norm related to the 2-norm of the square matrix $\mathbf{A}$:

$$\mu(\mathbf{A}) = \lim_{h \to 0^+} \frac{\|\mathbf{I} + h\mathbf{A}\| - 1}{h}.$$

The logarithmic norm may be negative, and has the property

$$\max_i \operatorname{Re} \lambda_i \leq \mu(\mathbf{A}),$$

where $\{\lambda_i\}$ are the eigenvalues of $\mathbf{A}$. Bounding the norm of the solution of a nonlinear ODE follows similar reasoning; for a more detailed explanation of the nonlinear case, see [83, 192].

4.4 Error Analysis for Model Reduction

The development of error bounds in this section parallels the presentations in [165] and [155]. Consider approximating the solution $\mathbf{y} : [0, T] \to \mathbb{R}^n$ of (4.1) by the solution $\mathbf{x} : [0, T] \to \mathbb{R}^n$ of (4.2), constructed using a model reduction method, where $T > 0$. A bound on the error $\mathbf{e}(t) = \mathbf{x}(t) - \mathbf{y}(t)$ will be derived. Unlike the case of projection, however, $\mathbb{R}^n$ cannot be decomposed into complementary subspaces. Rather, the error $\mathbf{e}(t)$ will be linearly decomposed into two separate contributions, neither of which can be restricted to a proper subspace of $\mathbb{R}^n$: $\mathbf{e}_t(t)$, the error due to "truncating" the right-hand side of (4.1) in mapping it to (4.2), and $\mathbf{e}_p(t)$, the error due to propagating the truncation error over time. Truncation error corresponds to out-of-subspace error in the projection case and is similar to the idea of local truncation error in the numerical solution of ODEs [83]. Propagation error corresponds to in-subspace error in the projection case (see [165] and [155]).

Let

$$\mathbf{e}_t(t) = \mathbf{x}^* + \int_0^t \widehat{\mathbf{f}}(\mathbf{y}(u))\,du - \mathbf{y}(t), \tag{4.9}$$

$$\mathbf{e}_p(t) = \mathbf{x}(t) - \left(\int_0^t \widehat{\mathbf{f}}(\mathbf{y}(u))\,du + \mathbf{x}^*\right), \tag{4.10}$$

so that $\mathbf{e}(t) = \mathbf{e}_t(t) + \mathbf{e}_p(t)$.

Typically, no attempt is made to bound $\mathbf{e}_p(t)$ explicitly. However, an *a priori* error estimate for $\mathbf{e}_p(t)$ can be derived in terms of $\mathbf{e}_t(t)$. Differentiating (4.10) and substituting (4.1) and (4.2) for the resulting time derivatives yields

$$\dot{\mathbf{e}}_p(t) = \widehat{\mathbf{f}}(\mathbf{y}(t) + \mathbf{e}_p(t) + \mathbf{e}_t(t)) - \widehat{\mathbf{f}}(\mathbf{y}(t)), \quad \mathbf{e}_p(0) = \mathbf{0}, \tag{4.11}$$

where the initial condition follows from the definition of $\mathbf{e}_p$ in (4.10). In (4.11), $\mathbf{e}_t(t)$ and $\mathbf{y}(t)$ will be treated as forcing terms.

Before presenting error bounding results for the nonlinear ODE case, it is instructive to consider error bounding results for the linear case. Suppose that (4.1) takes the form $\dot{\mathbf{y}}(t) = \mathbf{A}\mathbf{y}(t)$ with $\mathbf{A} \in \mathbb{R}^{n\times n}$, and suppose (4.2) takes the form $\dot{\mathbf{x}}(t) = \widehat{\mathbf{A}}\mathbf{x}(t)$, where $\widehat{\mathbf{A}} \in \mathbb{R}^{n\times n}$. Then (4.11) becomes

$$\dot{\mathbf{e}}_p(t) = \widehat{\mathbf{A}}\mathbf{e}_p(t) + \widehat{\mathbf{A}}\mathbf{e}_t(t), \quad \mathbf{e}_p(0) = \mathbf{0}.$$

Using the result in (4.8) yields the bound

$$||\mathbf{e}_p|| \leq ||F(T, \widehat{\mathbf{A}})||\,||\widehat{\mathbf{A}}||\,||\mathbf{e}_t||,$$

so the total error is bounded by

$$||\mathbf{e}|| \leq (||F(T, \widehat{\mathbf{A}})||||\widehat{\mathbf{A}}|| + 1)||\mathbf{e}_t||.$$

The nonlinear case proceeds in analogous fashion. Write the solution $\mathbf{x} : [0, T] \to \mathbb{R}^n$ of (4.2) in terms of the solution $\mathbf{y} : [0, T] \to \mathbb{R}^n$ of (4.1):

$$\mathbf{x}(t) = \mathbf{y}(t) + \mathbf{e}_t(t) + \mathbf{e}_p(t), \tag{4.12}$$

where $T > 0$. Write a hypothetical solution $\widehat{\mathbf{y}} : [0, T] \to \mathbb{R}^n$ with truncation error *only* as:

$$\widehat{\mathbf{y}}(t) = \mathbf{y}(t) + \mathbf{e}_t(t) = \mathbf{x}^* + \int_0^t \widehat{\mathbf{f}}(\mathbf{y}(u))\, \mathrm{d}u. \tag{4.13}$$

Then the linear case can be generalized in the following theorem:

Theorem 4.4.1. *Let $\gamma \geq 0$ be the Lipschitz constant of $\widehat{\mathbf{f}}$ in a region containing $\mathbf{y}([0, T])$ and $\widehat{\mathbf{y}}([0, T])$. To be precise, suppose*

$$||\widehat{\mathbf{f}}(\widehat{\mathbf{y}}(t) + \mathbf{v}) - \widehat{\mathbf{f}}(\widehat{\mathbf{y}}(t))|| \leq \gamma ||\mathbf{v}|| \tag{4.14}$$

for all $(\mathbf{v}, t) \in A \subset \mathbb{R}^n \times [0, T]$, where the region A is such that the associated region $\bar{A} = \{(\widehat{\mathbf{y}}(t) + \mathbf{v}, t) : (\mathbf{v}, t) \in A\}$ contains $(\widehat{\mathbf{y}}(t), t)$ and $(\mathbf{y}(t), t)$ for all t in $[0, T]$. Let $\mu(\mathrm{D}\widehat{\mathbf{f}}(\mathbf{z})) \leq \bar{\mu}$ for $\mathbf{z} \in V \subset \mathbb{R}^n$, where V contains the set $\{\lambda\widehat{\mathbf{y}}(t) + (1 - \lambda)\mathbf{x}(t) : t \in [0, T], \lambda \in [0, 1]\}$, and $\mu(\cdot)$ denotes the logarithmic norm related to the 2-norm.

The function $\mathbf{e}_p$ satisfies

$$\inf\{C \geq 0 : |\mathbf{e}_p(t)| \leq C \text{ a.e. on } [0, T]\} = ||\mathbf{e}_p||_\infty \leq \begin{cases} \varepsilon\gamma\left(\frac{e^{2\bar{\mu}T} - 1}{2\bar{\mu}}\right)^{1/2}, & \bar{\mu} \neq 0, \\ \varepsilon\gamma T^{1/2}, & \bar{\mu} = 0, \end{cases} \tag{4.15}$$

and the 2-norm of the function e *satisfies*

$$\left(\int_0^T \|\mathrm{e}(t)\|^2 \,\mathrm{d}t\right)^{1/2} = \|\mathbf{e}\| \leq \begin{cases} \varepsilon\left(1+\gamma\left(\frac{e^{2\bar{\mu}T}-1-2\bar{\mu}T}{4\bar{\mu}^2}\right)^{1/2}\right), & \bar{\mu} \neq 0, \\ \varepsilon(1+2^{-1/2}\gamma T), & \bar{\mu} = 0, \end{cases} \tag{4.16}$$

where

$$\varepsilon = \|\mathbf{e}_t\| = \left(\int_0^T \|\mathrm{e}_t(t)\|^2 \,\mathrm{d}t\right)^{1/2}. \tag{4.17}$$

Proof. The proof follows the development of Proposition 4.2 in [165] and Theorem 4.1 in [155]. Applying a Taylor expansion for $h > 0$, $\mathbf{e}_p(t+h) = \mathbf{e}_p(t) + h\dot{\mathbf{e}}_p(t) + O(h^2)$, which satisfies

$$\begin{aligned} \|\mathbf{e}_p(t+h)\| &= \|\mathbf{e}_p(t) + h\dot{\mathbf{e}}_p(t) + O(h^2)\|, \\ &= \|\mathbf{e}_p(t) + h\widehat{\mathbf{f}}(\mathbf{y}(t) + \mathbf{e}_t(t) + \mathbf{e}_p(t)) - h\widehat{\mathbf{f}}(\mathbf{y}(t))\| + O(h^2). \end{aligned} \tag{4.18}$$

Using the triangle inequality on the previous equation (4.18) yields

$$\begin{aligned} \|\mathbf{e}_p(t+h)\| \leq\ & \|\mathbf{e}_p(t) + h\widehat{\mathbf{f}}(\mathbf{y}(t) + \mathbf{e}_t(t) + \mathbf{e}_p(t)) - h\widehat{\mathbf{f}}(\mathbf{y}(t) + \mathbf{e}_t(t))\| \\ & + h\|\widehat{\mathbf{f}}(\mathbf{y}(t) + \mathbf{e}_t(t)) - \widehat{\mathbf{f}}(\mathbf{y}(t))\| + O(h^2). \end{aligned} \tag{4.19}$$

Let $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$ be the function

$$\mathbf{g}(\boldsymbol{\eta}) = \boldsymbol{\eta} + h\widehat{\mathbf{f}}(\boldsymbol{\eta}). \tag{4.20}$$

Then

$$\|\mathbf{e}_p(t) + h\widehat{\mathbf{f}}(\mathbf{y}(t) + \mathbf{e}_p(t) + \mathbf{e}_t(t)) - h\widehat{\mathbf{f}}(\mathbf{y}(t) + \mathbf{e}_t(t))\| = \|\mathbf{g}(\widehat{\mathbf{y}}(t) + \mathbf{e}_p(t)) - \mathbf{g}(\widehat{\mathbf{y}}(t))\|. \tag{4.21}$$

Applying a multivariate mean value theorem (Exercise 2.5 from [52]) to g yields

$$||\mathbf{g}(\widehat{\mathbf{y}}(t) + \mathbf{e}_p(t)) - \mathbf{g}(\widehat{\mathbf{y}}(t)|| \leq \kappa ||\mathbf{e}_p(t)|| \tag{4.22}$$

for any $\kappa \in \mathbb{R}$ such that

$$\kappa \geq \sup_{\boldsymbol{\eta} \in [\widehat{\mathbf{y}}(t), \widehat{\mathbf{y}}(t) + \mathbf{e}_p(t)]} ||\mathrm{D}\mathbf{g}(\boldsymbol{\eta})|| = \sup_{\boldsymbol{\eta} \in [\widehat{\mathbf{y}}(t), \widehat{\mathbf{y}}(t) + \mathbf{e}_p(t)]} ||\mathbf{I} + h\mathrm{D}\widehat{\mathbf{f}}(\boldsymbol{\eta})||. \tag{4.23}$$

Here, for any two vectors $\boldsymbol{\eta}_1, \boldsymbol{\eta}_2 \in \mathbb{R}^n$, $[\boldsymbol{\eta}_1, \boldsymbol{\eta}_2]$ denotes the line segment joining the two. (Traditionally, this bracket notation refers to intervals; however, the convention used by Rathinam and Petzold [165] is followed here.) Since the line $[\widehat{\mathbf{y}}(t), \widehat{\mathbf{y}}(t) + \mathbf{e}_p(t)]$ is a compact subset of $\mathbb{R}^n$,

$$\sup_{\boldsymbol{\eta} \in [\widehat{\mathbf{y}}(t), \widehat{\mathbf{y}}(t) + \mathbf{e}_p(t)]} ||\mathbf{I} + h\mathrm{D}\widehat{\mathbf{f}}(\boldsymbol{\eta})|| = \max_{\boldsymbol{\eta} \in [\widehat{\mathbf{y}}(t), \widehat{\mathbf{y}}(t) + \mathbf{e}_p(t)]} ||\mathbf{I} + h\mathrm{D}\widehat{\mathbf{f}}(\boldsymbol{\eta})||. \tag{4.24}$$

It follows from (4.19), (4.21), (4.22), and (4.24), that

$$\begin{aligned} ||\mathbf{e}_p(t+h)|| - ||\mathbf{e}_p(t)|| &\leq \left(\max_{\boldsymbol{\eta} \in [\widehat{\mathbf{y}}(t), \widehat{\mathbf{y}}(t) + \mathbf{e}_p(t)]} ||\mathbf{I} + h\mathrm{D}\widehat{\mathbf{f}}(\boldsymbol{\eta})|| \right) ||\mathbf{e}_p(t)|| + h||\widehat{\mathbf{f}}(\mathbf{y}(t) + \mathbf{e}_t(t)) - \widehat{\mathbf{f}}(\mathbf{y}(t))||, \\ &\leq \left(\max_{\boldsymbol{\eta} \in [\widehat{\mathbf{y}}(t), \widehat{\mathbf{y}}(t) + \mathbf{e}_p(t)]} ||\mathbf{I} + h\mathrm{D}\widehat{\mathbf{f}}(\boldsymbol{\eta})|| \right) ||\mathbf{e}_p(t)|| + h\gamma||\mathbf{e}_t(t)|| + O(h^2), \end{aligned} \tag{4.25}$$

which implies that

$$\frac{||\mathbf{e}_p(t+h)|| - ||\mathbf{e}_p(t)||}{h} \leq \bar{\mu}||\mathbf{e}_p(t)|| + \gamma||\mathbf{e}_t(t)|| + O(h), \tag{4.26}$$

where the $O(h)$ term may be uniformly bounded independent of $||\mathbf{e}_p(t)||$ (using theory from Taylor series, see [83], Equations 10.17 and 10.18). Then it follows from Theorem 10.6 of [83] that

$$||\mathbf{e}_p(t)|| \leq \gamma \int_0^t e^{\bar{\mu}(t-\tau)} ||\mathbf{e}_t(\tau)|| \, \mathrm{d}\tau. \tag{4.27}$$

After applying the Cauchy-Schwarz inequality on the right-hand side, then

$$\|\mathbf{e}_p(t)\| \leq \begin{cases} \gamma\Big(\frac{e^{2\bar{\mu}t}-1}{2\bar{\mu}}\Big)^{1/2}\Big(\int_0^t \|\mathbf{e}_t(\tau)\|^2\,\mathrm{d}\tau\Big)^{1/2}, & \bar{\mu} \neq 0, \\ \gamma t^{1/2}\Big(\int_0^t \|\mathbf{e}_t(\tau)\|^2\,\mathrm{d}\tau\Big)^{1/2}, & \bar{\mu} = 0, \end{cases} \tag{4.28}$$

from which it follows that

$$\|\mathbf{e}_p\|_\infty \leq \begin{cases} \varepsilon\gamma\Big(\frac{e^{2\bar{\mu}T}-1}{2\bar{\mu}}\Big)^{1/2}, & \bar{\mu} \neq 0, \\ \varepsilon\gamma T^{1/2}, & \bar{\mu} = 0. \end{cases} \tag{4.29}$$

Substituting (4.17), then squaring (4.28), integrating, and taking the square root to pass to the L^2-norm yields the bound

$$\|\mathbf{e}_p\| \leq \begin{cases} \varepsilon\gamma\Big(\frac{e^{2\bar{\mu}T}-1-2\bar{\mu}T}{4\bar{\mu}^2}\Big)^{1/2}, & \bar{\mu} \neq 0, \\ 2^{-1/2}\varepsilon\gamma T^{1/2}, & \bar{\mu} = 0. \end{cases} \tag{4.30}$$

Applying the triangle inequality yields

$$\|\mathbf{e}\| \leq \|\mathbf{e}_p\| + \|\mathbf{e}_t\| \leq \begin{cases} \varepsilon\Bigg(1+\gamma\Big(\frac{e^{2\bar{\mu}T}-1-2\bar{\mu}T}{4\bar{\mu}^2}\Big)^{1/2}\Bigg), & \bar{\mu} \neq 0, \\ \varepsilon(1+2^{-1/2}\gamma T), & \bar{\mu} = 0. \end{cases} \tag{4.31}$$

□

Remark 4.4.2. As in [155], when $\bar{\mu} < 0$, uniform bounds, independent of T, can be obtained from Theorem 4.4.1 from the inequality

$$\frac{e^{2\bar{\mu}t}-1}{\bar{\mu}} \leq |\bar{\mu}|^{-1},$$

in which case

$$\|\mathbf{e}_i\|_\infty \leq \varepsilon\gamma|2\bar{\mu}|^{-1/2}, \tag{4.32}$$

$$\|\mathbf{e}\| \leq \varepsilon\Big(1 + \frac{\gamma}{2|\bar{\mu}|}\Big). \tag{4.33}$$

Remark 4.4.3. If Theorem 4.4.1 is applied to projection-based model reduction, then the truncation error $\mathbf{e}_t$ is the complementary subspace error $\mathbf{e}_c$ in [155], and the propagating error $\mathbf{e}_p$ is the in-subspace error in [155]. Then Theorem 4.4.1 is equivalent to Corollary 4.4 of [155], aside from hypotheses on the Lipschitz constant γ; stronger bounds may be obtained by leveraging the structure of projection-based model reduction, either using Theorem 4.1 of [155] or Corollary 4.4 of [155]. For projection-based model reduction, Theorem 4.1 of [155] gives stronger bounds than Corollary 4.4 in that paper.

Remark 4.4.4. By considering truncation ($\mathbf{e}_t(t)$) and propagation ($\mathbf{e}_p(t)$) errors separately, this analysis yields a bound on the norm of the total error function $\mathbf{e}$ in terms of ε. The value of ε depends on the solution $\mathbf{y} : [0, T] \to \mathbb{R}^n$ of (4.1), the function $\widehat{\mathbf{f}}$, and the initial condition $\mathbf{x}^*$. Generally, $\|\mathbf{e}_t\|$ is not known exactly unless the solution of (4.1) is also calculated. Bounds on $\|\mathbf{e}_t\|$ may be estimated by using any error control results provided by a model reduction method, or by substituting a known solution of (4.1) with different initial conditions that approximates $\mathbf{y}$ into (4.9) and taking the norm. Using an estimate of ε in Theorem 4.4.1 only yields estimates of bounds on the function norm of the total error at best; such bounding estimates may be inaccurate if the function used to approximate $\mathbf{y}$ in (4.9) is a bad approximation. Consequently, estimates of ε must be used with caution.

4.5 Case Study

To illustrate the factors affecting $\mathbf{e}_p$ and $\mathbf{e}$ given $\mathbf{e}_t$ (or bounds on $\|\mathbf{e}_t\|_2$), consider the linear time invariant ODE

$$\dot{\mathbf{y}}(t) = \mathbf{A}\mathbf{y}(t), \quad \mathbf{y}(0) = \mathbf{y}^*, \tag{4.34}$$

where **A** takes the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{A}_2 \end{bmatrix}, \tag{4.35}$$

with the blocks of **A** first taking the values

$$\mathbf{A}_1 = \begin{bmatrix} -0.2 & 0 & 0 \\ 0 & -0.3464 & 4 \\ 0 & -4 & -0.3464 \end{bmatrix}, \tag{4.36}$$

$$\mathbf{A}_{12} = \begin{bmatrix} 0.3893 & 0.5179 & -1.543 \\ 1.39 & 1.3 & 0.8841 \\ 0.06293 & -0.9078 & -1.184 \end{bmatrix}, \tag{4.37}$$

$$\mathbf{A}_2 = \begin{bmatrix} -5 & 0 & 0 \\ 0 & -6.13 & -3.54 \\ 0 & 3.54 & -6.13 \end{bmatrix}, \tag{4.38}$$

so that $n = 6$. This example is related to an example in [165]. Three values of **A** were considered; modifications to (4.36), (4.37), and (4.38) will be discussed later in this section. The reduced model ODE for each value of **A** will be

$$\dot{\mathbf{x}}(t) = \widehat{\mathbf{A}}\mathbf{x}(t), \quad \mathbf{x}(0) = \mathbf{y}^*, \tag{4.39}$$

where

$$\widehat{\mathbf{A}} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_2 \end{bmatrix}. \tag{4.40}$$

Such a reduced model might occur when neglecting coupling between two sets of state variables.

The parameters $\bar{\mu}$ and γ were changed independently by scaling $\mathbf{A}_1$ or $\mathbf{A}_2$. The parameter ε was kept constant over all three choices of $\mathbf{A}$ by scaling $\mathbf{A}_{12}$ to compensate for changes in $\mathbf{A}_1$ and $\mathbf{A}_2$ so that the effects of changing γ and $\bar{\mu}$ could be studied independently.

Note that $\mathbf{A}_1$ and $\mathbf{A}_2$ are normal, and their eigenvalues have negative real parts. The spectrum of $\mathbf{A}$ is the union of the spectra of $\mathbf{A}_1$ and $\mathbf{A}_2$, and for the values of $\mathbf{A}$ in (4.35), (4.36), (4.37), and (4.38), the eigenvalues of $\mathbf{A}_2$ have large negative real parts compared to the eigenvalues of $\mathbf{A}_1$.

Given (4.34) and the corresponding reduced ODE (4.39), the values of γ and $\bar{\mu}$ in Theorem 4.4.1 were calculated as $\gamma = \|\widehat{\mathbf{A}}\|_2$ and $\bar{\mu} = \mu(\widehat{\mathbf{A}})$. The ODEs (4.34) and (4.39) were integrated using an explicit Runge-Kutta (4,5) Dormand-Prince pair using a relative tolerance of 10^{-13} and an absolute tolerance of 10^{-25} on each solution component. All random numbers were calculated using a Mersenne twister algorithm (MT19937). For (4.34) and (4.39), using the definition of $\mathbf{e}_t$ in (4.9) yields

$$\mathbf{e}_t(t) = \int_0^t (\widehat{\mathbf{A}} - \mathbf{A})\mathbf{y}(u)\,\mathrm{d}u. \tag{4.41}$$

The truncation error $\mathbf{e}_t$ is always calculated by using the trapezoidal rule, given $\mathbf{y}$. The total error is calculated as

$$\mathbf{e}(t) = \mathbf{x}(t) - \mathbf{y}(t), \tag{4.42}$$

so that the propagating error can be calculated as

$$\mathbf{e}_p(t) = \mathbf{e}(t) - \mathbf{e}_t(t). \tag{4.43}$$

The 2-norms of $\mathbf{e}_t$, $\mathbf{e}_p$, and $\mathbf{e}$ are also calculated using the trapezoidal rule. Calculations were implemented on a MacBook Pro 2011 model running Mac OS X 10.7.3 with a 2.7 GHz Intel Core i7 and 8 GB of 1333 MHz DDR3 RAM. Source code for implementations in MATLAB [133] and Python [209] are included for reproducibility in Appendix C.

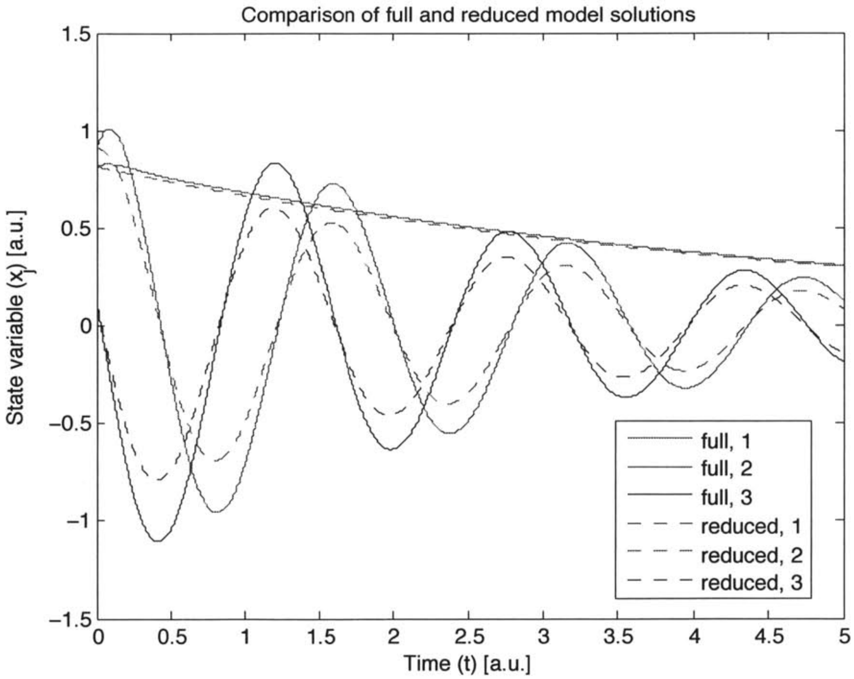

Figure 4-1: First three components of $\mathbf{x}(t)$ (dashed) and $\mathbf{y}(t)$ (solid) corresponding to the first choice of $\mathbf{A}$ as in (4.34), (4.35), (4.36), (4.37), and (4.38) and its corresponding reduced model. The last three components of $\mathbf{x}(t)$ and $\mathbf{y}(t)$ are identical, and are not plotted.

For the first choice of $\mathbf{A}$, taking values given by (4.35), (4.36), (4.37), and (4.38), it follows from Theorem 4.4.1 that $\bar{\mu} = -0.2$ and $\gamma = 7.0787$. The initial condition $\mathbf{y}^*$ was chosen randomly and $\mathbf{y}(t)$ and $\mathbf{x}(t)$ were computed over the interval $[0, 5]$. Note that the last three components of $\mathbf{y}(t)$ and $\mathbf{x}(t)$ are equal on $[0, 5]$, from which

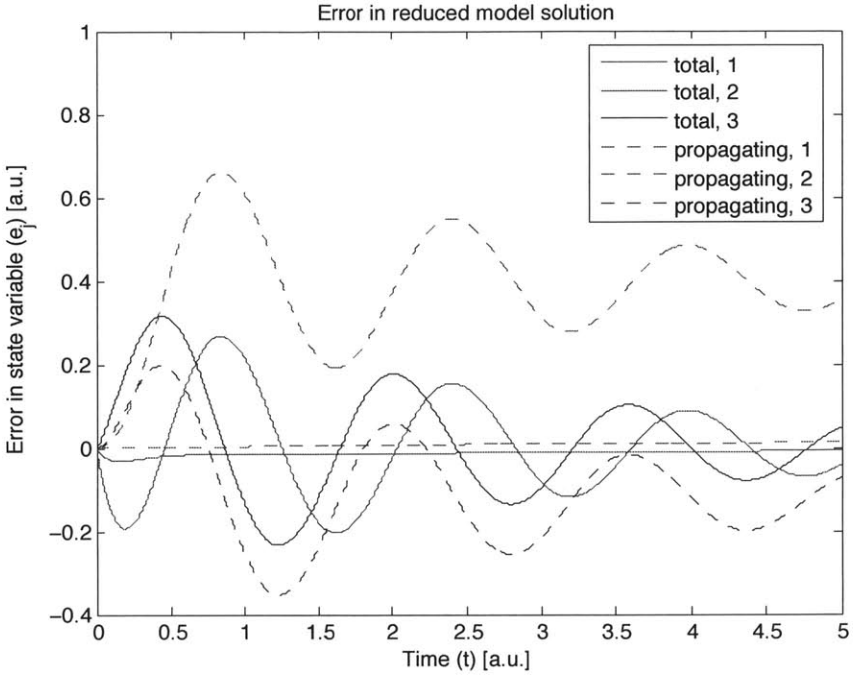

Figure 4-2: First three components of $\mathbf{e}_p(t)$ (dashed) and $\mathbf{e}(t)$ (solid) corresponding to the first choice of $\mathbf{A}$ as in (4.34), (4.35), (4.36), (4.37), and (4.38) and its corresponding reduced model. The last three components of $\mathbf{e}_p(t)$ and $\mathbf{e}(t)$ are zero, and are not plotted.

it follows (using (4.41)) that the last three components of $\mathbf{e}_t(t)$, $\mathbf{e}_p(t)$, and $\mathbf{e}(t)$ are all zero on $[0, 5]$. Consequently, only the first three components of $\mathbf{y}(t)$ and $\mathbf{x}(t)$ are plotted in Figure 4-1; this plot will be the only plot of $\mathbf{y}(t)$ and $\mathbf{x}(t)$, because the solutions $\mathbf{y}(t)$ and $\mathbf{x}(t)$ each behave similarly for all three values of $\mathbf{A}$ considered in this section. The first thre components of $\mathbf{e}_p(t)$ and $\mathbf{e}(t)$ are plotted in Figure 4-2; as with $\mathbf{y}(t)$ and $\mathbf{x}(t)$, the quantities $\mathbf{e}_p(t)$ and $\mathbf{e}(t)$ each behave similarly for all values of $\mathbf{A}$ considered in this section, and no further plots will be presented. The model reduction truncation error was $\varepsilon = \|\mathbf{e}_t\|_2 = 9.0197 \cdot 10^{-1}$; this value of ε will be the same for all three values of $\mathbf{A}$ considered. The sup-norm and 2-norm of the propagating error were $\|\mathbf{e}_p\|_\infty = 6.6042 \cdot 10^{-1}$ and $\|\mathbf{e}_p\|_2 = 4.0231 \cdot 10^{-1}$. The error bounds provided by Theorem 4.4.1 were one or two orders of magnitude larger: $\|\mathbf{e}_p\|_\infty \leq 9.3873$, $\|\mathbf{e}_p\|_2 \leq 1.7008 \cdot 10^{1}$, and $\|\mathbf{e}\|_2 \leq 1.7910 \cdot 10^{1}$.

The second choice of $\mathbf{A}$ considered was to keep $\mathbf{A}_1$ as in (4.36), scale the value of $\mathbf{A}_2$ in (4.38) by a factor of 2 so that γ increased by a factor of 2, and scale the value of $\mathbf{A}_{12}$ in (4.37) by a factor of 1.9745 to keep ε at the same value. The value of $\bar{\mu}$ remained the same, so that $\bar{\mu} = -0.2$, $\gamma = 1.4157 \cdot 10^1$, and $\varepsilon = \|\mathbf{e}_t\|_2 = 9.0197 \cdot 10^{-1}$. The sup-norm and 2-norm of the propagating error were $\|\mathbf{e}_p\|_\infty = 6.8930 \cdot 10^{-1}$ and $\|\mathbf{e}_p\|_2 = 1.0029$; the total error was $\|\mathbf{e}\|_2 = 4.4784 \cdot 10^{-1}$. The error bounds provided by Theorem 4.4.1 were $\|\mathbf{e}_p\| \leq 1.8775 \cdot 10^1$, $\|\mathbf{e}_p\|_2 \leq 3.5016 \cdot 10^1$, and $\|\mathbf{e}\|_2 \leq 3.4918 \cdot 10^1$.

The third and final choice of $\mathbf{A}$ was to keep $\mathbf{A}_{12}$ as in (4.37) and $\mathbf{A}_2$ as in (4.38), and scale the value of $\mathbf{A}_1$ in (4.36) by a factor of one-half. This choice of $\mathbf{A}$ kept γ and ε at the same values calculated using the first choice of $\mathbf{A}$, but decreased $\bar{\mu}$ by a factor of one-half. Consequently, for this choice of $\mathbf{A}$, $\bar{\mu} = -0.1$, $\gamma = 7.0787$, and $\varepsilon = \|\mathbf{e}_t\|_2 = 9.0197 \cdot 10^{-1}$. The sup-norm and 2-norm of the propagating error were $\|\mathbf{e}_p\|_\infty = 6.9820 \cdot 10^{-1}$ and $\|\mathbf{e}_p\|_2 = 1.0838$; the 2-norm of the total error in the reduced model solution was $\|\mathbf{e}\|_2 = 5.8730 \cdot 10^{-1}$. The error bounds provided by Theorem 4.4.1 were $\|\mathbf{e}_p\|_\infty \leq 1.1351 \cdot 10^1$, $\|\mathbf{e}_p\|_2 \leq 1.9363 \cdot 10^1$, and $\|\mathbf{e}\|_2 \leq 2.0265 \cdot 10^1$.

4.6 Discussion

Similar to the results in [155], inferences can be made about how the choice of $\widehat{\mathbf{f}}$ affects the error bounds. The three primary factors influencing bounds are the parameters ε, γ, and $\bar{\mu}$. As noted in the previous section, ε is controlled by a model reduction method in the ideal case; if the reduced model is chosen well, ε will be small. An interpretation of γ and $\bar{\mu}$ can be made using arguments similar to those in [155]; such arguments can also be extended to prove an alternate version of Theorem 4.4.1, similar to the work of [29]. The results of such arguments are that $\bar{\mu}$ corresponds to the maximum rate of change of the solution to (4.2), and that γ corresponds to the maximum rate of change of the right-hand side of (4.2). If $\widehat{\mathbf{f}}$ is chosen well, $\bar{\mu}$ and γ will be small, but if it is chosen such that ε is small, $\widehat{\mathbf{f}}$ must also faithfully represent the dynamics of (4.1). A common application of model reduction is to stiff systems. If (4.1) is a stiff system, (4.2) is also likely to be stiff for small ε, even though one aim of model reduction is to make such stiff systems less stiff. As a result, barring fortuitous properties of a specific choice of $\widehat{\mathbf{f}}$, $\bar{\mu}$ and γ are still likely to be large, yielding a bound that overestimates the approximation error due to model reduction in most cases of stiff systems. This finding is consistent with related work by Gronwall and Dahlquist on bounding the norm of solutions of ODEs.

4.7 Conclusions and Future Work

In this work, state space error bounds for model reduction methods were derived in the nonlinear ODE setting in terms of the function norm of the truncation error due to model reduction. These are the first such bounds that are method-agnostic and do not rely on a projection-based structure. When the function norm of the truncation error is not known precisely, the bounding result yields an estimate of the function norm of the total state space error. The analysis also yields insight into what factors affect the error in the reduced model solution, and shows that assuming additional structure (such as assuming that the model reduction method is projection-based) may yield stronger bounds, as in [155].

However, as in the projection-based case, it is difficult to calculate even estimates of state space bounds, and such bounds are likely to be loose. To enable calculation of stronger error estimates, methods from sensitivity analysis and global error estimates for ODEs should be employed, using ideas from the analysis above. These error estimates are easier to calculate, should yield better results, and should provide users of non-projection-based reduced order models with better information regarding the accuracy and validity of their reduced model approximations.

Chapter 5

Contributions and Future Work

The chapter will be divided into two parts. In the first section, the main contributions of this thesis will be summarized. In the second section, future work will be suggested, focusing on opportunities to develop new model reduction methods, as well as extending the error bounding work presented in Chapters 3 and 4.

5.1 Contributions

To recap, the contributions of this thesis are as follows.

First, the formalism of projection-based model reduction, common in fields other than combustion, is introduced to show that several model reduction methods developed for combustion applications are projection-based. This formalism enables analysis of projection-based model reduction methods as a whole, rather than analysis of each individual model reduction method in isolation.

This formalism motivates the *a priori* bounding of the global error in projection-based reduced order models. These bounds are derived using the same theory as *a priori* bounds on the global error in the numerical solution of ODEs [38], and extends a previous similar result by Rathinam and Petzold [165] that bounds the global error in orthogonal projection-based reduced order model. These bounds are the first to apply to oblique projection-based model reduction methods; many model reduction methods used in combustion are oblique projection-based. The bounds derived are tight, but often drastically overestimate the global error; their primary use is to demonstrate that in projection-based model reduction, local error control implies global error control. Similar conclusions were drawn when these bounds were derived for the numerical solution of ODEs [38].

This previous result is then extended to *all* model reduction methods. Many model reduction methods in combustion, such as reaction elimination [17, 153] and simultaneous reaction and species elimination [137], are *not* projection-based, and require separate theory. Although the global error bounds are typically weak, they again demonstrate that local error control implies global error control.

Finally, all of the source code used to generate the numerical results in this thesis is included in appendices. The inclusion of this source code more completely documents the algorithms used in this work, and also ensures that the results of this thesis are reproducible. Furthermore, inclusion of the source code prevents unnecessary duplication of effort, and enables future researchers to more easily build upon the work in this thesis.

5.2 Future Work

The original proposal for this these was to extend the reaction elimination work of Bhattacharjee, *et al.* [17] and Oluwole, *et al.* [153], and the simultaneous reaction and species elimination work of Mitsos, *et al.* [137], to interval-constrained simultaneous reaction and species elimination. In addition, the previous work on reaction and species elimination was to be extended to point-constrained (similar to Bhattacharjee, *et al.* [17], and Mitsos, *et al.* [137]) and range-constrained (similar to Oluwole, *et al.* [153]) formulations. Time permitting, the projection-based approaches and reaction and species elimination-based approaches were to be combined and used in large-scale case studies (2-D and 3-D simulations of flames requiring parallel computing resources and adaptive model reduction). The mixed-integer linear programming (MILP, also called mixed-integer programming MIP) formulations for these approaches are simple to present. For clarity and posterity, these approaches will be discussed briefly, along with pertinent background. After that, extensions to the error bounding work presented in Chapters 3 and 4 will be discussed.

5.2.1 Opportunities to Develop New Model Reduction Methods

The basic premise of interval-constrained simultaneous reaction and species elimination is to modify the point-constrained simultaneous reaction and species elimination formulation by Mitsos, *et al.* [137] in the same fashion as the point-constrained reaction elimination formulation by Bhattacharjee, *et al.* [17] was modified to yield the interval-constrained reaction elimination formulation of Oluwole, *et al.* [153]. The purpose of such a modification is to ensure that any error control placed on the time derivatives of the state variables in a reduced model ODE is enforced over a union of hyperrectangles in the host set (*i.e.*, domain) of the state variables of the reduced model ODE, as noted by Oluwole, *et al.* [153]. A formulation for interval-constrained simultaneous reaction and species elimination will be presented in two steps. For posterity, an unpublished reformulation of point-constrained simultaneous reaction and species elimination by Mitsos [136] will be reproduced so that it may be documented publicly:

$$\min_{\mathbf{w},\mathbf{z}} \sum_{j=1}^{N_S} \alpha_j w_j + \sum_{i=1}^{N_R} \beta_i z_i, \tag{5.1a}$$

$$\text{s.t.} \quad \left| \frac{\sum_{j=1}^{N_S} h_j(T_\ell) M_j \sum_{i=1}^{N_R} \nu_{ji} r_i(\mathbf{x}_\ell, T_\ell) z_i}{\rho(\mathbf{x}_\ell, T_\ell) C_P(\mathbf{x}_\ell, T_\ell)} - \Gamma_0(\mathbf{x}_\ell, T_\ell) \right| \leq$$

$$atol_0 + rtol_0 |\Gamma_0(\mathbf{x}_\ell, T_\ell)|, \quad \ell = 1, \ldots, N_t, \tag{5.1b}$$

$$\left| \frac{M_j \sum_{i=1}^{N_R} \nu_{ji} r_i(\mathbf{x}_\ell, T_\ell) z_i}{\rho(\mathbf{x}_\ell, T_\ell)} - \Gamma_j(\mathbf{x}_\ell, T_\ell) \right| \leq$$

$$atol_j + rtol_j |\Gamma_j(\mathbf{x}_\ell, T_\ell)|, \quad j = 1, \ldots, N_S; \quad \ell = 1, \ldots, N_t, \tag{5.1c}$$

$$w_j \geq z_i, \quad j = 1, \ldots, N_S, \forall i : \nu_{ji} \neq 0, \tag{5.1d}$$

$$\mathbf{w} \in [0, 1]^{N_S}, \tag{5.1e}$$

$$\mathbf{z} \in \{0, 1\}^{N_R}, \tag{5.1f}$$

where the nomenclature for this formulation comes from Mitsos, *et al.* [137]:

- The model being reduced is an ODE governing an adiabatic-isobaric batch reactor for a given reaction mechanism:

$$\dot{\mathbf{y}}(t) = \mathbf{\Gamma}(\mathbf{y}(t)), \tag{5.2}$$

  where

$$\dot{y}_0(t) = \frac{\sum_{j=1}^{N_S} h_j(y_0(t)) M_j \sum_{i=1}^{N_R} \nu_{ji} r_i(\mathbf{x}(t), y_0(t))}{\rho(\mathbf{x}(t), y_0(t)) C_P(\mathbf{x}(t), y_0(t))}, \tag{5.3}$$

$$\dot{y}_j(t) = \frac{M_j \sum_{i=1}^{N_R} \nu_{ji} r_i(\mathbf{x}(t), y_0(t))}{\rho(\mathbf{x}(t), y_0(t))}, \quad j = 1, \ldots, N_S, \tag{5.4}$$

- N_S denotes the number of species in the reaction mechanism
- N_R denotes the number of reactions in the reaction mechanism

- N_t denotes the number of reference data points

- $y_0(t)$ is the system temperature

- $\mathbf{x}(t) = [\mathbf{y}_1(t), \dots, \mathbf{y}(t)]^{\mathrm{T}}$ are the system species mass fractions

- $j \in \{1, \dots, N_S\}$ is an index referring to species in the reaction mechanism

- $i \in \{1, \dots, N_R\}$ is an index referring to reactions in the reaction mechanism

- $\ell \in \{1, \dots, N_t\}$

- $\rho(\cdot, \cdot)$ is a function that returns the system mass density

- $r_i(\cdot, \cdot)$ is a function that returns the (volumetric) rate of reaction i

- $C_P(\cdot, \cdot)$ is a function that returns the (mixture) specific heat capacity of the system (at constant pressure)

- M_j is the molar mass of species j

- ν_{ji} is the net stoichiometric coefficient of species j in reaction i, using the usual convention that ν_{ji} is positive when species j is a net product of reaction i, negative when species j is a net reactant of reaction i, and zero otherwise.

- $h_j(\cdot)$ is the specific enthalpy of species j

- $z_i = 0$ if reaction i is excluded from the reduced mechanism and $z_i = 1$ if reaction i is included in the reduced mechanism

- $w_j = 0$ if species j is nonreactive in the reduced mechanism, and $w_j = 1$ if species j is reactive in the reduced mechanism

This nomenclature will be reused later in this section. In the case of the functions mentioned in the list above, the notation will be abused for the interval case in two specific ways. First, species mass fractions, $x_j(t)$, $j = 1, \dots, N_S$, will be replaced by species concentrations, $c_j(t)$, $j = 1, \ldots, N_S$, because species concentrations are used in [153] instead of species mass fractions. Second, interval extensions of the functions in the list above will be denoted by replacing their point arguments, denoted by lowercase Latin letters, with intervals, denoted by uppercase Latin letters. (Temperature, denoted by T, is the exception, and will always be a scalar when written explicitly.) A discussion of interval arithmetic (such as interval extensions) is outside the scope of this thesis; an interested reader should consult the brief introduction within the papers of Oluwole, *et al.* [153], as well as the books by Moore [140]; Moore, *et al.* [139]; and Neumaier [147].

Applying the transformations that take the point-constrained reaction elimination formulation of Bhattacharjee, *et al.* [17] to the interval-constrained formulation of Oluwole, *et al.* [153] to the formulation in 5.1 yields an interval-constrained simultaneous reaction and species elimination formulation.

$$
\begin{aligned}
&\min_{\mathbf{w},\mathbf{z}} \sum_{j=1}^{N_S} \alpha_j w_j + \sum_{i=1}^{N_R} \beta_i z_i, && (5.5a)\\
\text{s.t.}\quad &\sum_{i=1}^{N_R} (1 - z_i) I_{0i}^L(\mathbf{Y}_\ell) \geq -tol_0^U(\mathbf{Y}_\ell), \quad \ell = 1, \ldots, N_t, && (5.5b)\\
&\sum_{i=1}^{N_R} (1 - z_i) I_{0i}^U(\mathbf{Y}_\ell) \leq tol_0^U(\mathbf{Y}_\ell), \quad \ell = 1, \ldots, N_t, && (5.5c)\\
&\sum_{i=1}^{N_R} (1 - z_i) I_{ji}^L(\mathbf{Y}_\ell) \geq -tol_j^U(\mathbf{Y}_\ell), \quad j = 1, \ldots, N_S, \quad \ell = 1, \ldots, N_t, && (5.5d)\\
&\sum_{i=1}^{N_R} (1 - z_i) I_{ji}^U(\mathbf{Y}_\ell) \leq tol_j^U(\mathbf{Y}_\ell), \quad j = 1, \ldots, N_S, \quad \ell = 1, \ldots, N_t, && (5.5e)\\
&w_j \geq z_i, \quad j = 1, \ldots, N_S, \forall i : \nu_{ji} \neq 0, && (5.5f)\\
&\mathbf{w} \in [0, 1]^{N_S}, && (5.5g)\\
&\mathbf{z} \in \{0, 1\}^{N_R}, && (5.5h)
\end{aligned}
$$

where the nomenclature from this formulation is partially borrowed from Oluwole, *et al.* [153]:

- $\phi = [T(t), \mathbf{c}^{\mathrm{T}}(t)]^{\mathrm{T}}$ in Oluwole, *et al.* [153] is replaced with $\mathbf{y} = [T(t), \mathbf{c}^{\mathrm{T}}]^{\mathrm{T}}$, where $T(t) = y_0(t)$ denotes the system temperature and $\mathbf{c}(t)$ denotes the system species concentrations. As stated earlier, capital letters denote intervals, except temperature, which will always be capitalized, so $\mathbf{Y}_\ell$ is an interval for $\ell = 1, \ldots, N_t$.

- The index k in Oluwole, *et al.* [153] is replaced with the index i.

- The function I_{ji} is defined by:

$$I_{ji}(\mathbf{y}(t)) = \begin{cases} \frac{\sum_{J=1}^{N_S} M_J \nu_{Ji} h_J(y_0(t)) r_i(\mathbf{c}(t), y_0(t))}{\rho(\mathbf{c}(t), y_0(t)) C_P(\mathbf{c}(t), y_0(t))}, & j = 0; \quad i = 1, \ldots, N_R, \\ \frac{M_j \nu_{ji} r_i(\mathbf{c}(t), y_0(t))}{\rho(\mathbf{c}(t), y_0(t))}, & j = 1, \ldots, N_S, \quad i = 1, \ldots, N_R, \end{cases} \tag{5.6}$$

  as in Oluwole, *et al.* [153], with the argument $\phi = [T, \mathbf{c}]^{\mathrm{T}}$ replaced by $\mathbf{y} = [T, \mathbf{c}]^{\mathrm{T}}$.

- The superscripts L and U refer to the lower and upper bounds, respectively, of the interval extension of I_{ji}. This notation replaces the subscripts *lo* and *up* notation in Oluwole, *et al.* [153].

- The function tol_j is defined by

$$tol_j(\mathbf{y}) = atol_j + rtol_j |\Gamma_j(\mathbf{y})|, \quad j = 0, \ldots, N_S, \tag{5.7}$$

  as an abuse of notation. This function defines the error tolerances for model reduction as a function of the reference data used for model reduction. Again, the $\mathbf{y}$ in this chapter replaces the argument ϕ in Oluwole, *et al.* [153].

It is hoped that the comments above clarify the inconsistencies in notation among Bhattacharjee, *et al.* [17]; Oluwole, *et al.* [153]; and Mitsos, *et al.* [137].

Based on the presentation in Chapter 2, a natural extension of the point-constrained reaction elimination formulation of Bhattacharjee, *et al.* [17] and point-constrained simultaneous reaction and species elimination formulation of Mitsos, *et al.* [137] to projection-based model reduction is

$$\begin{aligned}
&\min_{\mathbf{P}} \sum_{j=1}^{N_S+1} p_{jj} = \operatorname{tr}(\mathbf{P}), && (5.8a)\\
\text{s.t.}\quad &|(\mathbf{I}-\mathbf{P})\mathbf{\Gamma}(\mathbf{y}_\ell)| \leq [tol_0(\mathbf{y}_\ell), \ldots, tol_{N_S}(\mathbf{y}_\ell)]^{\mathrm{T}}, \quad \ell = 1, \ldots, N_t, && (5.8b)\\
&\mathbf{P}^2 = \mathbf{P}, && (5.8c)\\
&\mathbf{P} \in \mathbb{R}^{(N_S+1)\times(N_S+1)}, && (5.8d)
\end{aligned}$$

where the absolute value and inequality in (5.8b) are both applied element-wise. The objective function (5.8a) is the number of variables in a Petrov-Galerkin (or lumped) representation of a projection-based reduced model, analogous to the objective functions in reaction elimination and simultaneous reaction and species elimination that each represent a metric for the "size" of the reduced model. Error control is accomplished via the constraints in (5.8b), analogous to the error control in both reaction elimination and simultaneous reaction and species elimination.

For the remainder of this section, it will be useful to denote the range and nullspace of a matrix $\mathbf{A}$ by $\mathcal{R}(\mathbf{A})$ and $\mathcal{N}(\mathbf{A})$, respectively.

Despite its intuitiveness, the formulation in (5.8) is problematic because it is large, nonlinear, and nonconvex. The problem is large because it has $(N_S+1)^2$ decision variables, and there exist combustion reaction mechanisms in use with more than $N_S = 10^3$ species, yielding instances of (5.8) with over 10^6 decision variables. Although linear programs with 10^6 are tractable, nonlinear programs with so many variables are not necessarily tractable. In addition to being nonlinear, the formulation in (5.8) is also nonconvex, due to the nonlinear equality constraint (5.8c). Consequently, formulation (5.8) is intractable, except possibly for sufficiently small test cases. Furthermore, (5.8) could admit undesirable pathological solution. For instance, if $N_t \leq N_S + 1$, then it is always possible to find a feasible solution to (5.8) with objective function value no greater than N_t by selecting $\mathbf{P}$ such that $\mathcal{R}(\mathbf{P}) = \operatorname{span}(\{\mathbf{\Gamma}(\mathbf{y}_\ell)\}_{\ell=1}^{N_t})$, and $\mathbf{P} = \mathbf{P}^{\mathrm{T}}$ (so that $\mathbf{P}$ is an orthogonal projector). This choice of projector will be exact at the N_t reference points selected, but will not necessarily be physically meaningful, since its range is the right-hand side of (5.2) evaluated at each of the reference points selected. A proof of this assertion is outside the scope of this thesis. An alternative formulation that is tractable and does not admit pathological solutions is preferable.

One possible reformulation of (5.8) is to let

$$\mathbf{P} = \mathbf{B}\operatorname{diag}(\mathbf{w})\mathbf{B}^{-1} \tag{5.9}$$

for a given invertible matrix $\mathbf{B} \in \mathbb{R}^{(N_S+1)\times(N_S+1)}$ by leveraging the similarity of projection matrices to binary diagonal matrices. If $\mathbf{b}_j$ is the j^{th} column of $\mathbf{B}$, then $\mathbf{b}_j \in \mathcal{R}(\mathbf{P}) = \mathcal{N}(\mathbf{I} - \mathbf{P})$ if $w_j = 1$ and $\mathbf{b}_j \in \mathcal{N}(\mathbf{P}) = \mathcal{R}(\mathbf{I} - \mathbf{P})$ if $w_j = 0$, for $j = 0, \ldots, N_S$ (to abuse notation and start indexing some vectors at zero, for consistency). For this reason, a natural name of $\mathbf{B}$ is "basis matrix".

It will be convenient to use the following expression:

$$\mathbf{B}\operatorname{diag}(\mathbf{w})\mathbf{B}^{-1} = \sum_{j=1}^{N_S+1} w_j \mathbf{b}_j \boldsymbol{\beta}_j^{\mathrm{T}}, \tag{5.10}$$

where $\boldsymbol{\beta}_j^{\mathrm{T}}$ is the j^{th} row of $\mathbf{B}^{-1}$ (distinct from the usage of $\boldsymbol{\beta}$ in (5.1)), and define the function $\mathcal{I}_j$ in the spirit of (5.6) by

$$\mathcal{I}_j(\mathbf{y}) = \mathbf{b}_j \boldsymbol{\beta}_j^{\mathrm{T}} \mathbf{\Gamma}(\mathbf{y}). \tag{5.11}$$

Using the expressions in (5.9), (5.10), and (5.14) yields the formulation

$$
\begin{aligned}
&\min_{\mathbf{w}} \sum_{j=1}^{N_S+1} w_j, && \text{(5.12a)}\\
\text{s.t.}\quad &\sum_{j=1}^{N_S+1} (1-w_j)\mathcal{I}_j(\mathbf{y}_\ell) \leq [tol_0(\mathbf{y}_\ell), \ldots, tol_{N_S}(\mathbf{y}_\ell)]^{\mathrm{T}}, \quad \ell = 1, \ldots, N_t, && \text{(5.12b)}\\
&\sum_{j=1}^{N_S+1} (1-w_j)\mathcal{I}_j(\mathbf{y}_\ell) \geq -[tol_0(\mathbf{y}_\ell), \ldots, tol_{N_S}(\mathbf{y}_\ell)]^{\mathrm{T}}, \quad \ell = 1, \ldots, N_t, && \text{(5.12c)}\\
&\mathbf{w} \in \{0,1\}^{N_S+1}. && \text{(5.12d)}
\end{aligned}
$$

Equation (5.12a) comes from noting that the trace of a matrix is invariant under change of basis:

$$
\operatorname{tr}(\mathbf{P}) = \operatorname{tr}(\mathbf{B}\operatorname{diag}(\mathbf{w})\mathbf{B}^{-1}) = \sum_{j=1}^{N_S+1} w_j. \tag{5.13}
$$

The error constraints in (5.12b) and (5.12c) are analogous to similar error constraints in point-constrained reaction elimination and point-constrained simultaneous reaction and species elimination. It can be shown that (5.12) is a restriction of (5.8); again, the proof is out of the scope of this thesis.

From (5.12), an interval-constrained formulation can be expressed easily:

$$
\begin{aligned}
&\min_{\mathbf{w}} \sum_{j=1}^{N_S+1} w_j, && \text{(5.14a)}\\
\text{s.t.}\quad &\sum_{j=1}^{N_S+1} (1-w_j)\mathcal{I}_j^U(\mathbf{Y}_\ell) \leq [tol_0^U(\mathbf{Y}_\ell), \ldots, tol_{N_S}^U(\mathbf{Y}_\ell)]^{\mathrm{T}}, \quad \ell = 1, \ldots, N_t, && \text{(5.14b)}\\
&\sum_{j=1}^{N_S+1} (1-w_j)\mathcal{I}_j^L(\mathbf{Y}_\ell) \geq -[tol_0^U(\mathbf{Y}_\ell), \ldots, tol_{N_S}^U(\mathbf{Y}_\ell)]^{\mathrm{T}}, \quad \ell = 1, \ldots, N_t, && \text{(5.14c)}\\
&\mathbf{w} \in \{0,1\}^{N_S+1}; && \text{(5.14d)}
\end{aligned}
$$

this formulation is analogous to interval-constrained reaction elimination and interval-constrained simultaneous reaction and species elimination.

A difficulty in using (5.12) and (5.14) at present is selection of the basis matrix B. Since B enters the formulation through a mathematical simplification with no physical explanation, and the simplification itself offers no guidance on the selection of B, external information must be used to select it. Determination of an "optimal" basis (in some sense) is an open question; based on discussion of pathological solution to (5.8), projector rank is not necessarily the best choice of objective. Most projection-based model reduction methods calculate a projection matrix P in such a way that a basis matrix B can be derived from an eigendecomposition. Such methods couple determination of the basis matrix and determination of the projector, and could be used to select a basis for (5.12) and (5.14). Since there are no known methods (to the author's knowledge) that control the error in the time derivative of state variables in a reduced model, evaluated at multiple reference points or intervals, (5.12) and (5.14) could be used to augment existing model reduction methods with those types of error control. Projection-based model reduction methods use physical considerations (such as the quasi-steady state approximation) or dynamical systems considerations (such as eigendecomposition of the Jacobian matrix of the right-hand side of an ODE) to calculate a projection matrix; independent of existing model reduction methods, these considerations may also be useful in determining a basis matrix. Finally, purely mathematical considerations (ease of solving the formulations (5.12) and (5.14), independent of dynamical systems or physical concerns) could be used to select a basis matrix, such as an orthonormal matrix, or an identity matrix. Regardless of the method used to select B, (5.12) and (5.14) can be used to evaluate the choice of B. Excepting pathological solutions, the rank of a projector associated with an optimal solution of (5.12) or (5.14) can be used as one metric for the quality of B at given reference state data points or intervals. Bad choices of B tend to correspond to optimal solutions of (5.12) and (5.14) associated with projectors that have large ranks; of course, the possibility exists that, for a given set of reference data and tolerances, no reduction is possible. Some model reduction algorithms and some linear algebra algorithms operate using iterative methods. Given an initial guess of B, it may be worth using (5.8) or other means to develop a method for calculating a "better" basis matrix (however one defines "better", since rank alone is an insufficient criterion, due to the pathological solutions to (5.8), and choice of additional criteria is not obvious). Assuming model reduction is possible for given reference state data and tolerances, the utilities of (5.8), (5.12), and (5.14) is limited by choice of B; there is an opportunity for significant advances in projection-based model reduction if effective choices of B can be found.

One potential interesting choice of B would be one in which the resulting **V** and **W** (from Chapter 2) could be used to evaluate the right-hand side of the lumped (Petrov-Galerkin) representation of the reduced model ODEs more quickly, reducing the CPU time required to solve the reduced model ODEs. An example of methods that choose **V** and **W** in this way are DEIM (discrete empirical interpolation method) and POD-DEIM [32, 31].

5.2.2 Opportunities to Develop Better Error Estimates and Bounds

As noted in Chapters 3 and 4, the *a priori* error bounds on solutions to reduced models tend to drastically overestimate the approximation error in the solution of the reduced model due to model reduction. A similar situation exists for the methods used to develop *a priori* error bounds on numerical solutions to ODEs [77, 38, 192]. Two approaches are used to compute more detailed information about the error in numerical solutions to ODEs.

One approach is to calculate *a posteriori* estimates of the error. This approach derives an ODE whose solution approximates the error (be it the error in the numerical solution to an ODE or the approximation error in the solution to a reduced order model) to leading order (asymptotically). *A posteriori* estimation has been used to estimate the error in orthogonal projection-based reduced models [86], error in reduced models due to perturbations in parameters [185], error due to operator decomposition methods for solving ODEs [54, 55] and PDEs [56, 58], error due to numerical methods used in solving reaction-diffusion PDEs [55], and global error control of numerical solutions of ODEs [218, 190, 57, 25]. Algorithmically, these methods are similar to those used in sensitivity analysis for ODEs [26, 46, 51, 131, 60, 24, 201], and this similarity can be used to develop multiple forward methods for *a posteriori* estimation of the approximation error in oblique projection-based model reduction methods and other, more general model reduction methods. Provided that the approximation error is sufficiently small, these methods provide accurate estimates of this error.

The other major approach is to calculate rigorous bounds on the error using interval bounding methods. Given the parametric ODE

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t), \mathbf{p}), \quad \mathbf{y}(0) = \mathbf{y}^* \in Y_0 \subset \mathbb{R}^n, \tag{5.15}$$

where Y_0 is a set of allowable initial conditions, n_p is the number of parameters, P is a set of allowable parameters, $\mathbf{p} \in P \subset \mathbb{R}^{n_p}$, the reachable set $\mathcal{S}(t)$ is defined by

$$\mathcal{S}(t) \equiv \{\mathbf{y}(t) : \mathbf{y}(t) \text{ satisfies (5.15) on } [0, t] \text{ for } (\mathbf{y}^*, \mathbf{p}) \in Y_0 \times P\}. \tag{5.16}$$

Interval bounding methods calculate an interval enclosure (*i.e.*, lower and upper bounds on each component) of the reachable set $\mathcal{S}(t)$.

To calculate rigorous bounds on the approximation error due to model reduction, an *exact* equation for the approximation error can be derived. Then, interval bounding methods can be applied to this equation. The two major approaches in interval bounding methods are Taylor methods and differential inequalities. Taylor methods use Taylor series expansions in concert with interval arithmetic (or more sophisticated arithmetics [16, 118]) to calculate validated enclosures of $\mathcal{S}(t)$, meaning that the interval enclosures hold even when computed in finite precision

[84, 163, 164, 187]. These methods are particularly useful for the computation of error in reduced order models because they provide the capability to bound rigorously both the error due to finite precision numerical methods and error due to model reduction approximations, even if $Y_0 \times P$ is a singleton (or a degenerate interval) [146]. Differential inequalities use interval arithmetic to derive a set of ODEs whose solution is an interval enclosure is $\mathcal{S}(t)$ [187]. The resulting enclosure, while rigorous, do not account for numerical errors, and thus do not yield guaranteed bounds in finite precision arithmetic, though this limitation can be remedied at the cost of additional sophistication [163]. Furthermore, for stable ODE systems, modern implementations of numerical methods for solving ODEs control the numerical error well; it is worth noting that for combustion systems, many problems of practical interest, such as ignition, or explosion, are not stable for all time. Both methods for interval bounding tend to calculate extremely conservative bounds on the reachable set of a parametric ODE, though these bounds can be improved by increasing the order of the Taylor series used in Taylor methods, or by leveraging external information to augment the calculated bounds [187]. Differential inequalities methods calculate bounds at a cost comparable to a single simulation; Taylor methods, in contrast, scale exponentially with the order, number of state variables, and number of parameters. Both approaches should be useful in bounding the error, and it is expected that once implementations for both methods are in place, performance concerns will dominate. In particular, when $Y_0 \times P$ is a singleton (*i.e.*, only one initial condition and one parameter are under consideration), rigorous bounds (or estimates, for that matter) must be computed more quickly than the exact error; otherwise, an exact error computation will be preferred.

Appendix A

Implementation of Examples for Chapter 2

Examples for Chapter 2 were implemented in MATLAB r2012a [133] and twice in Python 2.7.3 [209]. A Cantera input file for an ozone mechanism is required, and is also listed below in Cantera CTI format.

A.1 Cantera Ozone CTI file

```
#
# Generated from file ozone.inp
# by ck2cti on Tue Jul 19 14:32:00 2011
#
units(length = "cm", time = "s", quantity = "mol", act_energy = "kJ/mol")


ideal_gas(name = "ozone",
      elements = " O ",
      species = """ O  O2  O3 """,
      reactions = "all",
      initial_state = state(temperature = 300.0,
                        pressure = OneAtm)    )



#-------------------------------------------------------------------------------
#  Species data
#-------------------------------------------------------------------------------

species(name = "O",
    atoms = " O:1 ",
    thermo = (
       NASA( [  300.00,  1000.00], [  2.946429000E+00,  -1.638166000E-03,
                2.421032000E-06,  -1.602843000E-09,   3.890696000E-13,
                2.914764000E+04,   2.963995000E+00] ),
       NASA( [ 1000.00,  5000.00], [  2.542060000E+00,  -2.755062000E-05,
               -3.102803000E-09,   4.551067000E-12,  -4.368052000E-16,
                2.923080000E+04,   4.920308000E+00] )
             ),
    note = "120186"
       )

species(name = "O2",
    atoms = " O:2 ",
    thermo = (
       NASA( [  300.00,  1000.00], [  3.212936000E+00,   1.127486000E-03,
               -5.756150000E-07,   1.313877000E-09,  -8.768554000E-13,
               -1.005249000E+03,   6.034738000E+00] ),
       NASA( [ 1000.00,  5000.00], [  3.697578000E+00,   6.135197000E-04,
               -1.258842000E-07,   1.775281000E-11,  -1.136435000E-15,
               -1.233930000E+03,   3.189166000E+00] )
             ),
    note = "121386"
       )

species(name = "O3",
    atoms = " O:3 ",
    thermo = (
       NASA( [  300.00,  1000.00], [  2.462609000E+00,   9.582781000E-03,
               -7.087359000E-06,   1.363368000E-09,   2.969647000E-13,
                1.606152000E+04,   1.214187000E+01] ),
       NASA( [ 1000.00,  5000.00], [  5.429371000E+00,   1.820380000E-03,
               -7.705607000E-07,   1.499293000E-10,  -1.075563000E-14,
                1.523527000E+04,  -3.266387000E+00] )
             ),
    note = "121286"
       )



#-------------------------------------------------------------------------------
#  Reaction data
#-------------------------------------------------------------------------------

#  Reaction 1
reaction(  "O3 + O => O2 + O + O",  [6.76000E+06, 2.5, 101])

#  Reaction 2
reaction(  "O2 + O + O => O3 + O",  [1.18000E+02, 3.5, 0])

#  Reaction 3
reaction(  "O3 + O2 => O2 + O + O2",  [6.76000E+06, 2.5, 101])

#  Reaction 4
```

```
reaction(  "O2 + O + O2 => O3 + O2",  [1.18000E+02, 3.5, 0])

#  Reaction 5
reaction(  "O3 + O3 => O2 + O + O3",  [6.76000E+06, 2.5, 101])

#  Reaction 6
reaction(  "O2 + O + O3 => O3 + O3",  [1.18000E+02, 3.5, 0])

#  Reaction 7
reaction(  "O + O3 => 2 O2",  [4.58000E+06, 2.5, 25.1])

#  Reaction 8
reaction(  "2 O2 => O + O3",  [1.88000E+06, 2.5, 415])

#  Reaction 9
reaction(  "O2 + O => 2 O + O",  [5.71000E+06, 2.5, 491])

#  Reaction 10
reaction(  "2 O + O => O2 + O",  [2.47000E+02, 3.5, 0])

#  Reaction 11
reaction(  "O2 + O2 => 2 O + O2",  [5.71000E+06, 2.5, 491])

#  Reaction 12
reaction(  "2 O + O2 => O2 + O2",  [2.47000E+02, 3.5, 0])

#  Reaction 13
reaction(  "O2 + O3 => 2 O + O3",  [5.71000E+06, 2.5, 491])

#  Reaction 14
reaction(  "2 O + O3 => O2 + O3",  [2.47000E+02, 3.5, 0])
```

A.2 MATLAB Implementation

The MATLAB r2012a [133] implementation requires the installation of Cantera 2.0.0b3 (or later) [73], the Cantera MATLAB interface, Sundials 2.4 (or later), and SundialsTB [85].

```
function OzoneCaseStudy
% Purpose: Calculates the solution to an adiabatic-isobaric batch reactor
% problem using ozone mechanism.

% Close all open figures
close all

% Set up the problem parameters by setting the gas equal to GRIMech

problemData.gas = importPhase('ozone.cti');
initialTemperature = 1000;
```

```
set(problemData.gas, 'Temperature', initialTemperature, ...
    'Pressure', oneatm, 'MassFractions', 'O:0, O2:0.15, O3:0.85');
problemData.timePts = linspace(0, 2e-6, 10000);

% Set conditions to stoichiometric flame.
numSpecies = nSpecies(problemData.gas)
numVars = numSpecies + 1;
problemData.initCond = zeros(numVars,1);

problemData.initCond(1) = initialTemperature;
problemData.initCond(2:end) = massFractions(problemData.gas);
problemData.initCond;

% Tolerances
problemData.absTol = 1e-15 * ones(size(problemData.initCond));
problemData.relTol = 1e-12;

% problemData.absTol = 1e-6 * ones(size(problemData.initCond));
% problemData.relTol = 1e-6;

% Start timer
tic;

% Full solution
fullSolution = fullSoln(problemData);

% Stop timer
toc;

cutoffTemp = 2263; %in Kelvin
originDataPt = find(fullSolution(2,:) > cutoffTemp, 1, 'first') - 1;
rednData.origin = fullSolution(2:end, originDataPt);

firstRangeVector = rhsFn(fullSolution(1, originDataPt), ...
    rednData.origin, problemData);
firstRangeVector(1) = 0; %Zero out the temperature component only!
secondRangeVector = [1;0;0;0];
% thirdRangeVector = [0;1;0;0];
% basis = [firstRangeVector, secondRangeVector, thirdRangeVector];
basis = [firstRangeVector, secondRangeVector];

[orthoBasis, ~] = qr(basis(:,1:2));

% V = orthoBasis(:,1:3);
V = orthoBasis(:,1:2);
V(:,1) = -V(:,1);

projector = V*V';

% projector = zeros(4,4);
% projector(1,1) = 1;
% projector(2,2) = 1;
% projector(4,4) = 1;
```

```
redProblemData = problemData;
redProblemData.initCond = rednData.origin;
redProblemData.absTol = 1e-15 * ones(size(redProblemData.initCond));
redProblemData.relTol = 1e-12;

rednData.P = projector;

tic;
redSolution = reducedSoln(redProblemData, rednData);
toc;

% Establish proper dimensions of lumped model solution.
redLumpedSolution = zeros(size(V,2) + 1, size(redSolution,2));
% Then recover lumped solution from reduced model solution, since the two
% are equivalent.
redLumpedSolution(1,:) = redSolution(1,:);
redLumpedSolution(2:end, :) = V' * (redSolution(2:end, :) - ...
    repmat(rednData.origin, [1, size(redSolution,2)]));

% Establish proper dimensions of lumped original model solutions.
origLumpedSoln = zeros(size(redLumpedSolution));
% Then lump the original model solution; equivalent to projecting the
% original model solution (after integration!).
origLumpedSoln(1,:) = fullSolution(1,:);
origLumpedSoln(2:end,:) = V' * (fullSolution(2:end, :) - ...
    repmat(rednData.origin, [1, size(redSolution,2)]));


npts = 20;

% Make a vector w/ points from 0 to 1
x = linspace(0, 1, npts);
%Make 2-D grid of points
[X, Y] = meshgrid(x);

% Plot cleanup carries out efficiently this loop, which restricts
% the plotted surface to the positive orthant:
% for i = 1:npts
%     for j = 1:npts
%         if (Y(i,j) > 1 - X(i,j))
%             Y(i,j) = 1 - X(i,j);
%         end
%     end
% end
Y = Y .* (Y <= 1 - X) + (1 - X) .* (Y > 1 - X);

% Forces plane to be blue or orange; for other colors, search
% "ColorSpec" in Matlab help
orange = [1 .5 0];
blue = [0 0 1];
purple = [.5 0 .5];
green = [0 1 0];

%X + Y + Z = 1 (sum of mass fractions equals 1);
```

```
Z = 1 - X - Y;
%only show positive orthant

x2 = linspace(0, 1, npts);
y2 = linspace(0, .5, npts);
[X2, Y2] = meshgrid(x2, y2);
% Permutation of solution coordinates to plotting coordinates.
permuteSolnToPlotCoord = [3, 4, 2];
% Point at which model is reduced.
planeOrigin = rednData.origin(permuteSolnToPlotCoord);
%Normal vector for plane
% normal = orthoBasis(permuteSolnToPlotCoord,1);
normal2 = [0; -orthoBasis(4,1)/orthoBasis(2,1); 0; 1];
normal = normal2(permuteSolnToPlotCoord);
normal = normal/norm(normal);

%Z intercept
Z2 = planeOrigin(3) - (normal(1)*(X2 - planeOrigin(1))...
    + normal(2)*(Y2 - planeOrigin(2))) / normal(3);

alpha = .4;
alphaData = alpha * ones(size(X));

figure
hold on
hidden on
surf(X2, Y2, Z2, 'EdgeColor', 'none', 'FaceColor', orange, ...
     'AlphaData', .4 * alphaData, 'AlphaDataMapping', 'none', ...
     'FaceAlpha', 'interp');
plot3(fullSolution(4,:), fullSolution(5,:), ...
    fullSolution(3,:), 'b--', 'LineWidth', 1.5);
plot3(redSolution(4,:), redSolution(5,:), redSolution(3,:), ...
    'r-', 'LineWidth', 1.5);
quiver3([planeOrigin(1)], [planeOrigin(2)], ...
    [planeOrigin(3)], [.05*normal(1)], [.05*normal(2)], ...
    [.05*normal(3)], 0.2);
legend('Original model', 'Reduced model', ...
    'Reduction plane', 'Normal vector', 'Location', 'Best');
axis([0 1 0 1 0 .025]);
title('Projector Representation: Ozone')
xlabel('Mass Frac O_2')
ylabel('Mass Frac O_3')
zlabel('Mass Frac O')
view([-69, 42])
grid on

figure
hold on
% plot3(lumpedOriginalModel(:,2), ozone.data(refPointIndex:end, 1), ...
%     ozone.data(refPointIndex:end,3), 'b--', 'LineWidth', 1.15);
% plot3(lumpedReducedModel(:,2), ozone.data(refPointIndex, 1) + ...
%     redOzone.data(:, 1), redOzone.data(:,3), 'r-', 'LineWidth', 1.75);
plot3(origLumpedSoln(2,originDataPt:end), ...
    origLumpedSoln(1,originDataPt:end) ...
    -origLumpedSoln(1,originDataPt),...%Subtract reduced model time offset
    fullSolution(3,originDataPt:end),...
    'b--', 'LineWidth', 1.15);
plot3(redLumpedSolution(2,:), ...
    redLumpedSolution(1,:), ...
    redSolution(3,:), ...
    'r-', 'LineWidth', 1.75);
xlabel(...
  '\alpha * Mass Frac O + \beta * Mass Frac O_2 + \gamma * Mass Frac O_3');
% ylabel('Mass Frac O');
ylabel('Time [s]');
zlabel('Mass Frac O');
view([-160 22]);
grid on
% xlabel('Temperature [K]');
legend('Original Model', 'Reduced Model', 'Location', 'Best');
title('Lumped Representation: Ozone');


figure
hold on
plot3(fullSolution(5,originDataPt:end), ...
    fullSolution(1,originDataPt:end) ...
    - fullSolution(1,originDataPt), ... %Subtract reduced model time offset
    fullSolution(3,originDataPt:end), 'b--', 'LineWidth', 1.5);
plot3(redSolution(5,:), redSolution(1,:), redSolution(3,:), ...
    'r-', 'LineWidth', 1.5);
xlabel('Mass Frac O_3');
ylabel('Time [s]');
zlabel('Mass Frac O');
grid on;
% view([11 10]);
legend('Original Model', 'Reduced Model', 'Location', 'Best');
title('Invariant Representation: Ozone');

% 2-D plots that are time traces of dynamics

figure
plot(fullSolution(1,:), fullSolution(2,:), 'b-');
hold on
plot(fullSolution(1, originDataPt) + redSolution(1,:), ...
    redSolution(2,:), 'r--');
xlabel('Time [s]');
ylabel('Temperature [K]')
title('Cantera simulation: Temperature profile');
legend('Original Model', 'Reduced Model', 'Location', 'Best');

figure
plot(fullSolution(1,:), fullSolution(3,:), 'b-');
hold on
plot(fullSolution(1, originDataPt) + redSolution(1,:), ...
    redSolution(3,:), 'r--');
xlabel('Time [s]');
ylabel('Mass Fraction O [a.u.]')
```

```
title('Cantera simulation: Mass Fraction O profile');
legend('Original Model', 'Reduced Model', 'Location', 'Best');

figure
plot(fullSolution(1,:), fullSolution(4,:), 'b-');
hold on
plot(fullSolution(1, originDataPt) + redSolution(1,:), ...
    redSolution(4,:), 'r--');
xlabel('Time [s]');
ylabel('Mass Fraction O2 [a.u.]')
title('Cantera simulation: Mass Fraction O2 profile');
legend('Original Model', 'Reduced Model', 'Location', 'Best');

figure
plot(fullSolution(1,:), fullSolution(5,:), 'b-');
hold on
plot(fullSolution(1, originDataPt) + redSolution(1,:), ...
    redSolution(5,:), 'r--');
xlabel('Time [s]');
ylabel('Mass Fraction O3 [a.u.]')
title('Cantera simulation: Mass Fraction O3 profile');
legend('Original Model', 'Reduced Model', 'Location', 'Best');

end

function solution = fullSoln(problemData)
% Purpose: Solves adiabatic-isobaric batch reactor problem.
% Inputs: problemData = struct containing problem data.
%      problemData.gas = Cantera "Solution" object containing gas phase
%      thermodynamic state data.
%      problemData.initCond = initial conditions of ODEs in problem;
%                             column vector (problemData.npts by 1)
%      problemData.timePts = times at which the solution should be
%      calculated; first time point corresponds to initial conditions!
%      problemData.absTol = vector of absolute tolerances for CVODE
%      problemData.relTol = scalar relative tolerance for CVODE
% Outputs: solution = solution of Fitzhugh-Nagumo problem.

% Relative and absolute tolerances for integration
% numVars = length(problemData.initCond);
relTol = problemData.relTol;
absTol = problemData.absTol;

% Options for integration using CVODE in sundialsTB
% Hand-coded Jacobian function options
% integrationOptions = CVodeSetOptions('UserData',problemData,...
%                            'RelTol',relTol,...
%                            'AbsTol',absTol,...
%                            'LinearSolver','Dense',...
%                            'JacobianFn',@jacFn,...
%                             'InitialStep', 1e-9, ...
%                             'MaxNumSteps', 1e7, ...
%                             'ErrorMessages', true, ...
%                             'MinStep', eps, ...
%                            'StopTime', problemData.timePts(end));

%                          'MaxNumSteps', 1e7, ...

% Difference quotient (DQ; finite difference) Jacobian options
integrationOptions = CVodeSetOptions('UserData',problemData,...
                          'RelTol',relTol,...
                          'AbsTol',absTol,...
                          'LinearSolver','Dense',...
                          'InitialStep', 1e-9, ...
                           'MaxNumSteps', 1e7, ...
                          'ErrorMessages', true, ...
                          'StopTime', problemData.timePts(end));
%                           'MaxStep', 1e-6, ...
%                           'MinStep', eps, ...

% Initialize integrator using BDF method and Newton solver
CVodeInit(@rhsFn, 'BDF', 'Newton', ...
    problemData.timePts(1), problemData.initCond, integrationOptions);

% Solution time points are columns; time is the first row, and then
% solution(2:end, :) = y(1:end) for each time point
solution = [problemData.timePts(1); problemData.initCond];

% Integration loop for remaining steps
for i=2:length(problemData.timePts)

%     Take a time step
    [status, t, y] = CVode(problemData.timePts(i), 'Normal');
    solution = horzcat(solution, [problemData.timePts(i); y]);

    if status ~= 0
        fprintf(1, 'Status = %d', status);
        break;
    end

end

stats = CVodeGetStats

% Free memory
CVodeFree;

end

function ydot = rhs(t, y, data)
% Purpose: Calculates the right-hand side adiabatic-isobaric batch reactor.
% Carries out the calculation for full system, such that it only returns
% the time derivatives.
% Inputs: t = time
%         y = vector of state variables (column vector); temperature [K]
%         first, followed by species mass fractions
%      data = struct containing problem data
%      data.gas = Cantera "Solution" object containing gas phase
%      thermodynamic state data.
% Outputs: ydot = vector of time derivatives of state variables; must have
%                 same dimensions as y
%            flag = Used to return error flags
%      new_data = Used to update data (otherwise, return empty array if
%                 there are no changes to avoid recopying)

setTemperature(data.gas, y(1));
setPressure(data.gas, oneatm);
% numSpecies = nSpecies(data.gas);
% I = eye(numSpecies);
% massEnthalpies = zeros(numSpecies,1);
% for i = 1:numSpecies
%     setMoleFractions(data.gas, I(:,i));
%     massEnthalpies(i) = enthalpy_mass(data.gas);
% end

% Update the thermodynamic state of the gas to be consistent with the
% current thermodynamic state of the system as described by the state
% vector y. Use the 'nonorm' option to ensure that mass fractions are NOT
% normalized (for calculation of derivatives).

setMassFractions(data.gas, y(2:end), 'nonorm');

% Ideal gas constant in J/kmole-K
% R = 8314;

% Calculate pure species molar enthalpies
% moleEnthalpies = (enthalpies_RT(data.gas) * R * y(1)).';

% Set up the time derivative vector
ydot = zeros(size(y));

% Set up the time derivatives of each state variable for an
% adiabatic-isobaric batch reactor, using mass fractions and temperatures
% as the thermodynamic state variables.
one_over_rho = 1.0/density(data.gas);
wdot = netProdRates(data.gas);
ydot(1) = - y(1) * gasconstant * enthalpies_RT(data.gas)' * ...
    wdot * one_over_rho / cp_mass(data.gas);
ydot(2:end) = wdot .* molarMasses(data.gas) * one_over_rho;
% mw = molarMasses(data.gas);
% nsp = nSpecies(data.gas);
% for i = 1:nsp
%     ydot(i+1) = one_over_rho * mw(i) * wdot(i);
% end

end

function [ydot, flag, new_data] = rhsFn(t, y, data)
% Purpose: Calculates the right-hand side adiabatic-isobaric batch reactor.
% Carries out the calculation for full system.
% Inputs: t = time
%          y = vector of state variables (column vector); temperature [K]
%          first, followed by species mass fractions
%       data = struct containing problem data
%       data.gas = Cantera "Solution" object containing gas phase
%       thermodynamic state data.
% Outputs: ydot = vector of time derivatives of state variables; must have
%                 same dimensions as y
%           flag = Used to return error flags
%       new_data = Used to update data (otherwise, return empty array if
%                 there are no changes to avoid recopying)

ydot = rhs(t, y, data);

% Update the thermodynamic state of the gas to be consistent with the
% current thermodynamic state of the system as described by the state
% vector y.
% set(data.gas, 'Temperature', y(1), 'MassFractions', y(2:end), ...
%       'Pressure', oneatm);

flag = 0;
% new_data = data;
new_data = [];

end

function ydot= rhsForJac(t, y, data)
% Purpose: Calculates the right-hand side adiabatic-isobaric batch reactor.
% Carries out the calculation for full system, such that it only returns
% the time derivatives.
% Inputs: t = time
%          y = vector of state variables (column vector); temperature [K]
%          first, followed by species mass fractions
%       data = struct containing problem data
%       data.gas = Cantera "Solution" object containing gas phase
%       thermodynamic state data.
% Outputs: ydot = vector of time derivatives of state variables; must have
%                 same dimensions as y
%           flag = Used to return error flags
%       new_data = Used to update data (otherwise, return empty array if
%                 there are no changes to avoid recopying)

setTemperature(data.gas, y(1));
numSpecies = nSpecies(data.gas);
I = eye(numSpecies);
massEnthalpies = zeros(numSpecies,1);
for i = 1:numSpecies
    setMoleFractions(data.gas, I(:,i));
    massEnthalpies(i) = enthalpy_mass(data.gas);
end

% Update the thermodynamic state of the gas to be consistent with the
% current thermodynamic state of the system as described by the state
% vector y. Use the 'nonorm' option to ensure that mass fractions are NOT
% normalized (for calculation of derivatives).
```

```
setMassFractions(data.gas, y(2:end), 'nonorm');

% Ideal gas constant in J/kmole-K
% R = 8314;

% Calculate pure species molar enthalpies
% moleEnthalpies = (enthalpies_RT(data.gas) * R * y(1)).';

% Set up the time derivative vector
ydot = zeros(size(y));

% Set up the time derivatives of each state variable for an
% adiabatic-isobaric batch reactor, using mass fractions and temperatures
% as the thermodynamic state variables.
ydot(1) = sum(molarMasses(data.gas) .* massEnthalpies .* ...
    netProdRates(data.gas), 1) / (cp_mass(data.gas) * density(data.gas));
ydot(2:end) = netProdRates(data.gas) .* molarMasses(data.gas) / ...
    density(data.gas);

end

function [J, flag, new_data] = jacFn(t, y, fy, data)
% Purpose: Calculates the Jacobian for the Homescu et al. 20 species
% example for CVODE. Simple nonlinear ODE example.
% Carries out calculation for full system.
% Inputs: t = time
%         y = vector of state variables (column vector); temperature [K]
%         first, followed by species mass fractions
%         ydot = vector of derivatives of state variables wrt time
%      data = struct containing problem data
%      data.gas = Cantera "Solution" object containing gas phase
%      thermodynamic state data.
% Outputs: J = Jacobian matrix; must have dimensions conformal to
%              premultiplying y
%        flag = Used to return error flags
%   new_data = Used to update data (otherwise, return empty array if there
%              are no changes to avoid recopying)

% Make the workspaces for the numerical Jacobian matrix global so that the
% workspaces are persistent between calls
% global fac
% atol = 1e-15 * ones(size(y)); %1e-15 is default atol for Cantera
% rtol = 1e-10; %1e-9 is default rtol for Cantera

atol = data.absTol;
rtol = data.relTol;

J = CVodeNumJac(@rhsFn, t, y, fy, data, atol, rtol);

flag = 0;
new_data = [];

end
```

```
function redSoln = reducedSoln(problemData, rednData)
% Purpose: Solves the original (not reduced) Fitzhugh-Nagumo problem and
% returns the solution data (including time points!).
% Inputs: problemData = struct containing problem data.
%       problemData.gas = Cantera "Solution" object containing gas phase
%       thermodynamic state data.
%       problemData.initCond = initial conditions of ODEs in problem;
%                              column vector (problemData.npts by 1)
%       problemData.timePts = times at which the solution should be; first
%       time point corresponds to initial conditions!
%       reported; column vector
%       problemData.absTol = vector of absolute tolerances for CVODE
%       problemData.relTol = scalar relative tolerance for CVODE
%       rednData.origin = origin of reduced model (column vector same size
%       as problemData.initCond; problemData.nPts by 1)
%       rednData.P = projection matrix
% Outputs: solution = solution of Fitzhugh-Nagumo problem.

% Calculate the projector
problemData.P = rednData.P;

% Relative and absolute tolerances for integration
relTol = problemData.relTol;
absTol = problemData.absTol;

% Options for integration using CVODE in sundialsTB
% integrationOptions = CVodeSetOptions('UserData',problemData,...
%                           'RelTol',relTol,...
%                           'AbsTol',absTol,...
%                           'LinearSolver','Dense',...
%                           'JacobianFn',@reducedJacFn,...
%                           'StopTime', problemData.timePts(end));

% Difference quotient (DQ; finite difference) Jacobian options
integrationOptions = CVodeSetOptions('UserData',problemData,...
                          'RelTol',relTol,...
                          'AbsTol',absTol,...
                          'LinearSolver','Dense',...
                          'InitialStep', 1e-9, ...
                           'MaxNumSteps', 1e7, ...
                          'ErrorMessages', true, ...
                          'StopTime', problemData.timePts(end));
%                           'MaxStep', 1e-6, ...
%                           'MinStep', eps, ...


% Reduced model initial conditions must be calculated for original model
% initial conditions
reducedInitCond = problemData.P * ...
    (problemData.initCond - rednData.origin) + ...
    rednData.origin;

% Initialize integrator using BDF method and Newton solver
CVodeInit(@reducedRhsFn, 'BDF', 'Newton', ...
    problemData.timePts(1), reducedInitCond, integrationOptions);

% Solution time points are columns; time is the first row, and then
% v(1:end) corresponds to redSoln(2:2:end, :), and w(1:end) corresponds to
% redSoln(3:2:end, :).
redSoln = [problemData.timePts(1); reducedInitCond];

% Integration loop for remaining steps
for i=2:length(problemData.timePts)

%     Take a time step
    [status, t, y] = CVode(problemData.timePts(i), 'Normal');
    redSoln = horzcat(redSoln, [problemData.timePts(i); y]);

    if status ~= 0
        fprintf(1, 'Status = %d', status);
        break;
    end

end

stats = CVodeGetStats

% Free memory
CVodeFree;

end

function [ydot, flag, new_data] = reducedRhsFn(t, y, data)
% Purpose: Calculates the right-hand side for the reduced Fitzhugh-Nagumo
% system for CVODE. This second-order PDE is discretized using central
% differences for the spatial derivatives, and second-order finite
% differences for the two Neumann boundary conditions.
% Carries out the calculation for reduced systems.
% Inputs: t = time
%         y = vector of state variables (column vector)
% Remember! y(1:2:end) = v(1:data.nPts); y(2:2:end) = w(1:data.nPts).
%      data = struct containing all data
%      data.epsilon = epsilon parameter of Fitzhugh-Nagumo equation
%      data.L = length of spatial domain
%      data.gamma = gamma parameter of Fitzhugh-Nagumo equation
%      data.b = b parameter of Fitzhugh-Nagumo equation
%      data.c = c parameter of Fitzhugh-Nagumo equation
%      data.nPts = number of points in spatial discretization
%      data.P = projection matrix; must be a data.nPts by data.nPts matrix
% Outputs: ydot = vector of time derivatives of state variables; must have
%                 same dimensions as y
%           flag = Used to return error flags
%      new_data = Used to update data (otherwise, return empty array if
%                 there are no changes to avoid recopying)

[ydot, flag, new_data] = rhsFn(t, y, data);
ydot = data.P * ydot;
```

```
end

function [J, flag, new_data] = reducedJacFn(t, y, fy, data)
% Purpose: Calculates the Jacobian for the reduced Fitzhugh-Nagumo system
% for CVODE.
% Carries out the calculation for both full and reduced systems
% Inputs: t = time
%         y = vector of state variables (column vector)
% Remember! y(1:2:end) = v(1:data.nPts); y(2:2:end) = w(1:data.nPts).
%        fy = vector of time derivative of state variables
%      data = struct containing all data
%      data.epsilon = epsilon parameter of Fitzhugh-Nagumo equation
%      data.L = length of spatial domain
%      data.gamma = gamma parameter of Fitzhugh-Nagumo equation
%      data.b = b parameter of Fitzhugh-Nagumo equation
%      data.c = c parameter of Fitzhugh-Nagumo equation
%      data.nPts = number of points in spatial discretization
%      data.P = projection matrix; must be a data.nPts by data.nPts matrix
% Outputs: J = Jacobian matrix; must have dimensions conformal to
%              premultiplying y
%       flag = Used to return error flags
%   new_data = Used to update data (otherwise, return empty array if there
%              are no changes to avoid recopying)

[J, flag, new_data] = jacFn(t, y, fy, data);
J = data.P * J;

end
```

A.3 Python Implementation

The first Python 2.7.3 [209] implementation requires the installation of Cantera 2.0.0b3 (or later) [73], the Cantera Python interface, NumPy 1.6.2 (or later) [152], SciPy 0.10.1 (or later) [93], and Matplotlib 1.0.0 (or later) [90]. An attempt was made to keep the number of dependencies to a minimum. It is likely that the Python code below will work with Python 2.6 (or later).

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-

# Dependencies:
# numpy (used version 1.7.0-dev)
# scipy (used version 0.11-dev)
# Cantera (used version 2.0b4)
# matplotlib (used version 1.1.0)

import numpy
import scipy.linalg
import scipy.integrate
import Cantera
import matplotlib.pyplot
import mpl_toolkits.mplot3d

def adiabaticIsobaricBatch(t, y, data):
    """
    Purpose:
    Calculates the right-hand side of ODEs governing an adiabatic-
    isobaric batch reactor. Carries out the calculation for full system, such
    that it only returns the time derivatives.

    Arguments:
    t (float): time [s]
    y (numpy.ndarray, 1-D; or list, 1-D): (row) vector of state variables;
        temperature first, followed by species mass fractions
    data (dict): emulates C-style (or MATLAB-style) struct with following fields:
        data[’gas’] (Cantera.Solution): object containing chemistry and gas
            physical properties

    Returns:
    ydot (numpy.ndarray, 1-D): (row) vector of time derivatives of state
        variables; must have same shape as y

    """

    # Set gas thermodynamic properties; mass fractions must NOT be normalized
    # so that finite-difference Jacobian matrix calculated accurately
    data[’gas’].setTemperature(y[0])
    data[’gas’].setPressure(Cantera.OneAtm)
    data[’gas’].setMassFractions(y[1:], norm=0)

    # Preallocate time derivative vector
    ydot = numpy.zeros(numpy.asarray(y).shape)

    # Precalculate reciprocal density and net molar production rates for reuse
    one_over_rho = 1.0 / data[’gas’].density()
    wdot = data[’gas’].netProductionRates()

    # Calculate time derivatives
    ydot[0] = - y[0] * Cantera.GasConstant * \
        numpy.dot(data[’gas’].enthalpies_RT(),wdot) * one_over_rho / \
        data[’gas’].cp_mass()
    # multiplication of numpy arrays = elementwise multiply of the two arrays, like
    ydot[1:] = wdot * data[’gas’].molarMasses() * one_over_rho
    return ydot

def redAdiabaticIsobaricBatch(t, y, data):
    """
    Purpose:
    Calculates the right-hand side of ODEs governing an adiabatic-
    isobaric batch reactor. Carries out the calculation for full system, such
    that it only returns the time derivatives.
    Arguments:
    t (float): time [s]
    y (numpy.ndarray, 1-D; or list, 1-D): (row) vector of state variables;
        temperature first, followed by species mass fractions
    data (dict): emulates C-style (or MATLAB-style) struct with following fields:
        data['gas'] (Cantera.Solution): object containing chemistry and gas
            physical properties
        data['P'] (numpy.ndarray, 2-D, square; or numpy.mat, square;
                list, 2-D): projection matrix; must be
            conformal for premultiplying numpy.mat(y).transpose()

    Returns:
    ydot (numpy.ndarray, 1-D): (row) vector of time derivatives of state
        variables; must have same shape as y

    """
    ydot = adiabaticIsobaricBatch(t, y, data)

    # ydot is a row vector, so instead of calculating (P * ydot^{T})^{T}, we
    # calculate ydot * P^{T}. numpy ndarrays are more efficient than matrices,
    # and the latter formulation uses fewer method calls than the former
    return numpy.asarray(numpy.dot(ydot, data['P'].transpose()))

def fullSoln(problemData):
    """
    Purpose:
    Solves adiabatic-isobaric batch reactor problem using the scipy.integrate
    interface to DVODE.

    Arguments:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
                as problemData['initCond']; or list, 1-D, same shape as
                problemData['initCond']): vector of absolute tolerances
            for DVODE.
        problemData['relTol'] (float): scalar relative tolerance for
            for DVODE.

    Returns:
    solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
            len(problemData['timePts']), numpy.ndarray.shape[1] ==
            (len(problemData['initCond']) + 1)): Solution of problem;
        each time point is a row, each state variable is a column.

    """
    # Set up the integrator
    dvode = scipy.integrate.ode(adiabaticIsobaricBatch)
    dvode.set_integrator('vode',
                         method='bdf',
                         with_jacobian=True,
                         atol=problemData['absTol'],
                         rtol=problemData['relTol'],
                         first_step=1e-9,
                         nsteps=1e7)
    dvode.set_initial_value(problemData['initCond'], 0)
    dvode.set_f_params(problemData)

    # Carry out the main integration loop
    solution = numpy.hstack((0, numpy.asarray(problemData['initCond'],
                                              )))
    for t in problemData['timePts'][1:]:
        if not dvode.successful():
            raise ArithmeticError('DVODE step unsuccessful!')
        dvode.integrate(t)
        solution = numpy.vstack((solution, numpy.hstack((dvode.t, dvode.y))))

    return solution

def redSoln(problemData, rednData):
    """
    Purpose:
    Solves the reduced adiabatic-isobaric batch reactor problem using the
    scipy.integrate interface to DVODE.

    Arguments:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
                as problemData['initCond']; or list, 1-D, same shape as
                problemData['initCond']): vector of absolute tolerances
            for DVODE.
        problemData['relTol'] (float): scalar relative tolerance for
            for DVODE.
    rednData (dict): emulates C-style (or MATLAB-style) struct with following
            fields:
        rednData['P'] (numpy.ndarray, 2-D; or numpy.mat, 2-D; or list, 2-D):
            projection matrix used for na\"{i}ve projection-based model reduction
        rednData['origin'] (numpy.ndarray, 1-D, len(rednData['origin']) ==
                len(problemData['initCond']); or list, 1-D,
                len(rednData['origin']) == len(problemData['initCond'])):
            origin of reduced model
    Returns:
    solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
            len(problemData['timePts']), numpy.ndarray.shape[1] ==
            (len(problemData['initCond']) + 1)): Solution of problem;
        each time point is a row, each state variable is a column.

    """

    # Rebind object passed through rednData to problemData; this
    # object isn't modified within this function call scope (and below),
    # but if the script ever changes such that this statement is no longer
    # true, expect errors.
    problemData['P'] = rednData['P']

    # Calculate initial conditions for reduced model based on initial
    # conditions for full model. Again, y is a row vector, so instead
    # of calculating (P * y^{T})^{T}, we calculate y * P^{T}. numpy ndarrays
    # are more efficient than matrices, and the latter formulation uses fewer
    # method calls than the former.
    redInitCond = numpy.asarray(
        numpy.dot(numpy.asarray(problemData['initCond']) -
                  numpy.asarray(rednData['origin']),
                  problemData['P'].transpose())) + \
                  numpy.asarray(rednData['origin'])

    # Set up the integrator
    dvode = scipy.integrate.ode(redAdiabaticIsobaricBatch)
    dvode.set_integrator('vode',
                         method='bdf',
                         with_jacobian=True,
                         atol=problemData['absTol'],
                         rtol=problemData['relTol'],
                         first_step=1e-9,
                         nsteps=1e7)
    dvode.set_initial_value(redInitCond, 0)
    dvode.set_f_params(problemData)

    # Carry out the main integration loop

    solution = numpy.hstack((0, redInitCond))
    for t in problemData['timePts'][1:]:
        if not dvode.successful():
            raise ArithmeticError('DVODE step unsuccessful!')
        dvode.integrate(t)
        solution = numpy.vstack((solution, numpy.hstack((dvode.t, dvode.y))))

    return solution

def calcRedModelParams(fullSolution, problemData):
    """
    Purpose:
    Calculate projector based on data from full model solution.
```

```
Arguments:
fullSolution (numpy.ndarray, 2-D; or list, 2-D): solution of full model
problemData (dict): emulates C-style (or MATLAB-style) struct with
       following fields:
    problemData['gas'] (Cantera.Solution): object containing chemistry
        and gas physical properties

Returns:
origin_index (int): time point index corresponding to origin data point
origin (numpy.ndarray, 1-D, where origin.shape[0] ==
        fullSolution.shape[1]):
    origin of projection-based reduced model (transposed, for convenience)
projector (numpy.ndarray, 2-D, where projector.shape[0] ==
        projector.shape[1] == fullSolution.shape[1]): projection matrix
    for projection-based model reduction.
W (numpy.ndarray, 2-D, where W.shape[0] == (fullSolution.shape[1] - 1)):
    so-called "lumping matrix"; projection nullspace is perpendicular to
    span of this matrix
orthoBasis (numpy.ndarray, 2-D, where orthoBasis.shape[0] ==
        orthoBasis.shape[1] == (fullSolution.shape[1] - 1)):
    orthonormal basis such that its first two columns correspond to the
    range of the projector, and its last two columns correspond to the
    nullspace of the projector.

"""

# The philosophy here was to find a nice point in the full model solution
# to serve as the origin. The two basis vectors that span the range space
# were the unit vector [[1,0,0,0]].transpose(), which corresponds to the
# "temperature direction", and the tangent vector of the full model
# solution (i.e., the right-hand side of the full model ODE). These basis
# vectors are used to construct an orthogonal projector.
# IF YOU WANT TO MODIFY THE PROJECTOR, YOU MUST MODIFY THE INTERNALS OF
# THIS FUNCTION!

# The origin will be the point in the solution set calculated immediately
# before the first point in the solution set calculated that exceeds a
# cutoff temperature.
cutoff_temp = 2263 # Kelvin
origin_index = numpy.flatnonzero(fullSolution[:, 1] > cutoff_temp)[0] - 1
origin = fullSolution[origin_index, 1:]

# Reminder:
# In the MATLAB script, the first column of V corresponds to the "lump",
# and the second column corresponds to temperature. In this Python script,
# the first column corresponds to temperature, and the second corresponds
# to the "lump". In order to obtain the proper V matrix in Python, the 2
# input basis vectors must be specified in the reverse order of the 2
# input basis vectors specified in MATLAB.

# Having determined the origin, a basis must be constructed in order to
# calculate a projector. The first basis vector is going to be the
# right-hand side of the full model, evaluated at the origin. The second
#  basis vector is going to be [[1,0,0,0]].transpose(). The basis matrix
```

```
    # must consist of column vectors in order to carry out the necessary
    # linear algebra.
    first_range_vec = numpy.asarray([1, 0, 0, 0])
    second_range_vec = adiabaticIsobaricBatch(0, origin, problemData)
    basis = numpy.vstack((first_range_vec, second_range_vec)).transpose()

    # An orthogonal projector is constructed from this basis by
    # orthonormalizing it.
    [orthoBasis, _] = scipy.linalg.qr(basis)
    V = orthoBasis[:, 0:2]
    # Sign reversal here carried out here so that more entries in V are
    # positive than negative; doesn't affect results.
    #V[:, 0] = -V[:, 0]
    W = V
    projector = numpy.dot(V, W.transpose())

    return origin_index, origin, projector, W, orthoBasis

def lump_soln(soln, W, origin, origin_index):
    """
    Purpose:
    From a solution in the original state variables, calculate a "lumped"
    or "Petrov-Galerkin projected" solution.

    Arguments:
    soln (numpy.ndarray, 2-D): Solution of adiabatic-isobaric batch reactor
        problem; each time point is a row, each state variable is a column.
    W (numpy.ndarray, 2-D, where W.shape[0] == (soln.shape[1] - 1)):
        so-called "lumping matrix"; projection nullspace is perpendicular to
        span of this matrix
    origin (numpy.ndarray, 1-D, where origin.shape[0] ==
            fullSolution.shape[1]):
        origin of projection-based reduced model (transposed, for convenience)
    origin_index (int): time point index corresponding to origin data point

    Returns:
    lumped_soln (numpy.ndarray, 2-D, where lumped_soln.shape[0] ==
            soln.shape[0] and lumped_soln.shape[1] == (W.shape[1] + 1)):
        lumped version of soln

    """

    # Calculate lumped model solution
    #Copy time data points
    # Since each data point is a row, instead of calculating
    # W^{T} * (y - y_{0}), calculate (y - y_{0})^{T} * W.
    lumped_soln = numpy.hstack((numpy.asarray([soln[:, 0]]).transpose(),
        numpy.dot((soln[:, 1:] - numpy.tile(origin, (soln.shape[0], 1))), W)))

    return lumped_soln

def setProblemData():
    """
    Purpose:
```

```
    Sets problem parameters.

    Arguments:
    None

    Returns:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
                as problemData['initCond']; or list, 1-D, same shape as
                problemData['initCond']): vector of absolute tolerances
            for DVODE.
        problemData['relTol'] (float): scalar relative tolerance for
            for DVODE.

    """

    # Set up problem parameters.
    # IF YOU WANT TO CHANGE THE FULL MODEL SOLUTION (AND ALL THE OTHERS),
    # CHANGE THE PARAMETERS HERE!
    problemData = {}
    problemData['gas'] = Cantera.IdealGasMix('./ozone.cti')
    initialTemperature = 1000
    initialMoleFracString = 'O:0, O2:0.15, O3:0.85'
    problemData['gas'].set(T=initialTemperature,
                           P=Cantera.OneAtm,
                           Y=initialMoleFracString)
    problemData['timePts'] = numpy.linspace(0, 2e-5, 10000)

    # From the problem parameters, repackage the data so that it can be
    # passed to ODE solvers.
    problemData['initCond'] = numpy.hstack((
        numpy.asarray(initialTemperature),
        numpy.asarray(problemData['gas'].massFractions())))
    problemData['absTol'] = 1e-15
    problemData['relTol'] = 1e-12

    return problemData

def CalculateFullRedAndLumpedSolns():
    """
    Purpose:
    Calculate three different solutions for an ozone flame:
    - Full model solution
    - Reduced model solution, reduced using projection-based model reduction
    - Lumped model solution (or Petrov-Galerkin projection), derived from
      reduced model solution.
```

```
The basic idea is to decouple the calculation of solutions from the
plotting of figures so that the functions in this file are of a
manageable size.

Arguments:
None.

Returns:
fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame
origLumpedSoln (numpy.ndarray, 2-D): lumped version of full model solution
redLumpedSoln (numpy.ndarray, 2-D): lumped model solution for ozone flame
rednData (dict): emulates C-style (or MATLAB-style) struct with following
        fields:
    rednData['P'] (numpy.ndarray, 2-D; or numpy.mat, 2-D; or list, 2-D):
        projection matrix used for na\"{i}ve projection-based model reduction
    rednData['origin'] (numpy.ndarray, 1-D, len(rednData['origin']) ==
            len(problemData['initCond']); or list, 1-D,
            len(rednData['origin']) == len(problemData['initCond'])):
        origin of reduced model
origin_index (int): time point index corresponding to origin data point
orthoBasis (numpy.ndarray, 2-D, where orthoBasis.shape[0] ==
        orthoBasis.shape[1] == (fullSolution.shape[1] - 1)):
    orthonormal basis such that its first two columns correspond to the
    range of the projector, and its last two columns correspond to the
    nullspace of the projector.
"""

#Set up problem data
problemData = setProblemData()

# Calculate full model solution
fullSolution = fullSoln(problemData)

# From the full model solution, calculate a projector.
# IF YOU WANT TO CHANGE THE LUMPED AND REDUCED MODEL SOLUTIONS,
# CHANGE THE INTERNALS OF calculateProjector
(origin_index,
 origin,
 projector,
 W, orthoBasis) = calcRedModelParams(fullSolution, problemData)

# Calculate reduced model solution
# Rows are system states at a given time
# Columns are single state variables (or time)
redProblemData = setProblemData()
redProblemData['initCond'] = origin
rednData = {'origin': origin, 'P': projector}
redSolution = redSoln(redProblemData, rednData)

# Calculate "lumping" (or Petrov-Galerkin projection) of original and
# reduced models
origLumpedSoln = lump_soln(fullSolution, W, origin, origin_index)
```

```
    redLumpedSoln = lump_soln(redSolution, W, origin, origin_index)

    # Correct for the time discrepancy of the full and reduced models
    redSolution[:,0] += fullSolution[origin_index, 0]
    redLumpedSoln[:,0] += fullSolution[origin_index, 0]

    return (fullSolution, redSolution, origLumpedSoln, redLumpedSoln,
         rednData, origin_index, orthoBasis)

def plot_temp(fullSolution, redSolution):
    """
    Purpose:
    Make temperature versus time plots that compare the full and
    reduced model solutions.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame

    Returns:
    temp_fig (matplotlib.figure.Figure): temperature versus time plot

    """

    temp_fig = matplotlib.pyplot.figure()
    matplotlib.pyplot.plot(fullSolution[:,0], fullSolution[:,1], 'b-')
    matplotlib.pyplot.plot(redSolution[:, 0], redSolution[:, 1], 'r--')
    matplotlib.pyplot.ticklabel_format(axis='both', scilimits=(-2,3))
    matplotlib.pyplot.xlabel('Time [s]')
    matplotlib.pyplot.ylabel('Temperature [K]')
    matplotlib.pyplot.title('Cantera simulation: Temperature profile')
    matplotlib.pyplot.legend( ('Original model', 'Reduced model'), loc='best')

    return temp_fig

def plot_o(fullSolution, redSolution):
    """
    Purpose:
    Make mass fraction oxygen atoms versus time plots that compare the full and
    reduced model solutions.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame

    Returns:
    o_fig (matplotlib.figure.Figure): mass fraction O atoms versus time plot

    """
    o_fig = matplotlib.pyplot.figure()
    matplotlib.pyplot.plot(fullSolution[:, 0], fullSolution[:, 2], 'b-')
    matplotlib.pyplot.plot(redSolution[:, 0], redSolution[:, 2], 'r--')
    matplotlib.pyplot.ticklabel_format(axis='both', scilimits=(-2,3))
    matplotlib.pyplot.xlabel('Time [s]')
    matplotlib.pyplot.ylabel('Mass Fraction O [a.u.]')
    matplotlib.pyplot.title('Cantera simulation: Mass Fraction O profile')
    matplotlib.pyplot.legend( ('Original model', 'Reduced model'), loc='best')

    return o_fig

def plot_o2(fullSolution, redSolution):
    """
    Purpose:
    Make mass fraction O2 versus time plots that compare the full and
    reduced model solutions.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame

    Returns:
    o2_fig (matplotlib.figure.Figure): mass fraction O2 versus time plot

    """

    o2_fig = matplotlib.pyplot.figure()
    matplotlib.pyplot.plot(fullSolution[:, 0], fullSolution[:, 3], 'b-')
    matplotlib.pyplot.plot(redSolution[:, 0], redSolution[:, 3], 'r--')
    matplotlib.pyplot.ticklabel_format(axis='both', scilimits=(-2,3))
    matplotlib.pyplot.xlabel('Time [s]')
    matplotlib.pyplot.ylabel('Mass Fraction O2 [a.u.]')
    matplotlib.pyplot.title('Cantera simulation: Mass Fraction O2 profile')
    matplotlib.pyplot.legend( ('Original model', 'Reduced model'), loc='best')

    return o2_fig

def plot_o3(fullSolution, redSolution):
    """
    Purpose:
    Make mass fraction O3 versus time plots that compare the full and
    reduced model solutions.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame

    Returns:
    o3_fig (matplotlib.figure.Figure): mass fraction O3 versus time plot

    """

    o3_fig = matplotlib.pyplot.figure()
    matplotlib.pyplot.plot(fullSolution[:, 0], fullSolution[:, 4], 'b-')
    matplotlib.pyplot.plot(redSolution[:, 0], redSolution[:, 4], 'r--')
    matplotlib.pyplot.ticklabel_format(axis='both', scilimits=(-2,3))
    matplotlib.pyplot.xlabel('Time [s]')
    matplotlib.pyplot.ylabel('Mass Fraction O3 [a.u.]')
    matplotlib.pyplot.title('Cantera simulation: Mass Fraction O3 profile')
    matplotlib.pyplot.legend( ('Original model', 'Reduced model'), loc='best')

    return o3_fig

def plot_projector_rep(fullSolution, redSolution, orthoBasis, origin):
    """
    Purpose:
    Make phase plot that compares the full and reduced model solutions using
    the projector representation.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame
    orthoBasis (numpy.ndarray, 2-D, where orthoBasis.shape[0] ==
            orthoBasis.shape[1] == (fullSolution.shape[1] - 1)):
        orthonormal basis such that its first two columns correspond to the
        range of the projector, and its last two columns correspond to the
        nullspace of the projector
    origin (numpy.ndarray, 1-D, where origin.shape[0] ==
            fullSolution.shape[1]):
        origin of projection-based reduced model (transposed, for convenience)

    Returns:
    proj_rep_fig (matplotlib.figure.Figure): phase plot (O_2, O_3, O)
        comparing solutions of full model and projector representation of
        reduced model

    """
    # Phase plot of the solutions of the full and reduced models
    proj_rep_fig = matplotlib.pyplot.figure()
    axes = proj_rep_fig.gca(projection='3d')
    axes.plot(fullSolution[:,3], fullSolution[:,4], fullSolution[:,2], 'b--')
    axes.plot(redSolution[:,3], redSolution[:,4], redSolution[:,2], 'r-')

    # Set up the grid of (x,y) points for a plane to guide the eye
    n_pts = 20
    x = numpy.linspace(0, 1, n_pts)
    y = numpy.linspace(0, .5, n_pts)
    X, Y = numpy.meshgrid(x, y)

    # Set up the color of the plane
    plane_color = 'orange'
    plane_face_colors = numpy.empty(X.shape, dtype='|S'+str(len(plane_color)))
    plane_face_colors.fill(plane_color)

    # Since the plots permute the order of the solution matrix entries,
    # the basis entries and origin entries must also be permuted in a
    # consistent manner
    axis_permutation = [2, 3, 1]
    plane_origin = origin[axis_permutation]

    # The basis_index column of orthoBasis corresponds to the important
    # "lumping" direction. This column is used to determine the normal
    # vector of the plane in this figure that guides the eye.
    basis_index = 1
    normal = numpy.asarray([0,
        -orthoBasis[3, basis_index]/orthoBasis[1, basis_index],
        0,
        1])
    normal = normal[axis_permutation]
    normal = normal / numpy.linalg.norm(normal, 2)

    # Once the origin of the plane and the normal of the plane are determined,
    # the z coordinates of the plane are determined using analytic geometry.
    Z = plane_origin [2] - (normal[0] * (X - plane_origin[0]) +
        normal[1] * (Y - plane_origin[1])) / normal[2]

    # Plot the (x,y,z) coordinates of the plane that guides the eye
    plane = axes.plot_surface(X, Y, Z, facecolors=plane_face_colors,
                              shade=0, alpha=.4)
    plane.set_edgecolors(’none’)

    # Add legend, axis labels, title, etc.
    axes.set_title(’Projector Representation: Ozone’)
    axes.legend( (’Original model’, ’Reduced model’), loc=’best’)
    axes.set_xlabel(r’Mass Frac O$_2$’)
    axes.set_xlim(0, 1)
    axes.set_ylabel(r’Mass Frac O$_3$’)
    axes.set_ylim(0, 1)
    axes.set_zlabel(r’Mass Frac O’)
    axes.set_zlim(0, .025)
    #axes.view_init(elev=-69, azim=42)
    axes.grid()

    return proj_rep_fig

def plot_lumped_rep(fullSolution, redSolution, origLumpedSoln,
                    redLumpedSoln, origin_index):
    """
    Purpose:
    Make phase plot that compares the full and reduced model solutions using
    the lumped representation.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame
    origLumpedSoln (numpy.ndarray, 2-D): lumped full model solution for ozone
        flame
    redLumpedSoln (numpy.ndarray, 2-D): lumped reduced model solution for ozone
        flame
    origin_index (float): value of first index of fullSolution[:,:]
        corresponding to the origin of the reduced model

    Returns:
    lumped_rep_fig (matplotlib.figure.Figure): phase plot comparing solutions
        of lumped full model and lumped representation of reduced model
    """
    lumped_rep_fig = matplotlib.pyplot.figure()
    axes = lumped_rep_fig.gca(projection='3d')

    # Reminder:
    # In the MATLAB script, the first column of V corresponds to the "lump",
    # and the second column corresponds to temperature. In this Python script,
    # the first column corresponds to temperature, and the second corresponds
    # to the "lump".

    # Note: Time zero now corresponds to origin for both solutions in this plot
    axes.plot(origLumpedSoln[origin_index:,2],
              origLumpedSoln[origin_index:,0] - origLumpedSoln[origin_index,0],
              fullSolution[origin_index:,2],
              'b--')
    axes.plot(redLumpedSoln[:,2],
              redLumpedSoln[:,0] - redLumpedSoln[0,0],
              redSolution[:,2],
              'r-')
    axes.set_title('Lumped Representation: Ozone')
    axes.legend( ('Original Model', 'Reduced Model'), loc='best')
    axes.set_xlabel(r'$\alpha \cdot$ Mass Frac O ' +
                    r'$+ \beta \cdot$ Mass Frac O$_2$ '+
                    r'$+ \gamma \cdot$ Mass Frac O$_3$')
    axes.set_ylabel('Time [s]')
    axes.set_zlabel('Mass Frac O')
    #axes.view_init(elev=-160, azim=22)
    axes.grid()

    return lumped_rep_fig

def plot_invariant_rep(fullSolution, redSolution, origin_index):
    """
    Purpose:
    Make phase plot that compares the full and reduced model solutions using
    the invariant representation.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame
    origin_index (float): value of first index fullSolution[:,:]
        corresponding to the origin of the reduced model

    Returns:
    invariant_rep_fig (matplotlib.figure.Figure): phase plot comparing
        solutions of invariant representations of full and reduced models
    """
    invariant_rep_fig = matplotlib.pyplot.figure()
    axes = invariant_rep_fig.gca(projection='3d')
    # Note: Time zero now corresponds to origin for both solutions in this plot
    axes.plot(fullSolution[origin_index:,4],
              fullSolution[origin_index:,0] - fullSolution[origin_index, 0],
              fullSolution[origin_index:,2],
              'b--')
    axes.plot(redSolution[:,4],
              redSolution[:,0] - redSolution[0,0],
              redSolution[:,2],
              'r-')
    axes.set_title('Invariant Representation: Ozone')
    axes.legend( ('Original Model', 'Reduced Model'), loc='best')
    axes.set_xlabel(r'Mass Frac O$_3$')
    axes.set_ylabel(r'Time [s]')
    axes.set_zlabel(r'Mass Frac O')
    #axes.view_init(elev=11, azim=10)
    axes.grid()

    return invariant_rep_fig

# Main program:

(fullSolution,
 redSolution,
 origLumpedSoln,
 redLumpedSoln,
 rednData,
 origin_index, orthoBasis) = CalculateFullRedAndLumpedSolns()

temp_fig = plot_temp(fullSolution, redSolution)
o_fig = plot_o(fullSolution, redSolution)
o2_fig = plot_o2(fullSolution, redSolution)
o3_fig = plot_o3(fullSolution, redSolution)
proj_rep_fig = plot_projector_rep(fullSolution, redSolution, orthoBasis,
                                  rednData['origin'])
lumped_rep_fig = plot_lumped_rep(fullSolution, redSolution, origLumpedSoln,
                    redLumpedSoln, origin_index)
invariant_rep_fig = plot_invariant_rep(fullSolution, redSolution,
                                       origin_index)

matplotlib.pyplot.show()
```

The second Python 2.7.3 implementation requires, in addition to the dependencies of the first implementation, PyDASSL 0.0.1 [4], and Assimulo 2.2 [3]. This example implements multiple numerical integrators in order to validate the numerical results.

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-

# Dependencies:
# numpy (used version 1.7.0-dev)
# scipy (used version 0.11-dev)
# pydas (used version 0.1.0)
# Assimulo (used trunk version after version 2.1.1, version 2.1.2-dev?)
# Cantera (used version 2.0b4)
# matplotlib (used version 1.1.0)

import numpy
import scipy.linalg
import scipy.integrate
import pydas
import assimulo.problem
import assimulo.solvers
import Cantera
import matplotlib.pyplot
import mpl_toolkits.mplot3d
import copy

def adiabaticIsobaricBatch(t, y, data):
    """
    Purpose:
    Calculates the right-hand side of ODEs governing an adiabatic-
    isobaric batch reactor. Carries out the calculation for full system, such
    that it only returns the time derivatives.

    Arguments:
    t (float): time [s]
    y (numpy.ndarray, 1-D; or list, 1-D): (row) vector of state variables;
        temperature first, followed by species mass fractions
    data (dict): emulates C-style (or MATLAB-style) struct with following fields:
        data[’gas’] (Cantera.Solution): object containing chemistry and gas
            physical properties

    Returns:
    ydot (numpy.ndarray, 1-D): (row) vector of time derivatives of state
        variables; must have same shape as y

    """

    # Set gas thermodynamic properties; mass fractions must NOT be normalized
    # so that finite-difference Jacobian matrix calculated accurately
    data[’gas’].setTemperature(y[0])
    data[’gas’].setPressure(Cantera.OneAtm)
    data[’gas’].setMassFractions(y[1:], norm=0)

    # Preallocate time derivative vector
    ydot = numpy.zeros(numpy.asarray(y).shape)

    # Precalculate reciprocal density and net molar production rates for reuse
    one_over_rho = 1.0 / data[’gas’].density()
    wdot = data[’gas’].netProductionRates()

    # Calculate time derivatives
    ydot[0] = - y[0] * Cantera.GasConstant * \
        numpy.dot(data[’gas’].enthalpies_RT(),wdot) * one_over_rho / \
        data[’gas’].cp_mass()
    # multiplication of numpy arrays = elementwise multiply of the two arrays, like th
    ydot[1:] = wdot * data[’gas’].molarMasses() * one_over_rho
    return ydot

def redAdiabaticIsobaricBatch(t, y, data):
    """
    Purpose:
    Calculates the right-hand side of ODEs governing an adiabatic-
    isobaric batch reactor. Carries out the calculation for full system, such
    that it only returns the time derivatives.

    Arguments:
    t (float): time [s]
    y (numpy.ndarray, 1-D; or list, 1-D): (row) vector of state variables;
        temperature first, followed by species mass fractions
    data (dict): emulates C-style (or MATLAB-style) struct with following fields:
        data['gas'] (Cantera.Solution): object containing chemistry and gas
            physical properties
        data['P'] (numpy.ndarray, 2-D, square; or numpy.mat, square;
                list, 2-D): projection matrix; must be
            conformal for premultiplying numpy.mat(y).transpose()

    Returns:
    ydot (numpy.ndarray, 1-D): (row) vector of time derivatives of state
        variables; must have same shape as y

    """
    ydot = adiabaticIsobaricBatch(t, y, data)

    return numpy.dot(data['P'], ydot)

def fullSoln(problemData):
    return fullSoln_cvode(problemData)

def redSoln(problemData, rednData):
    return redSoln_cvode(problemData, rednData)

def fullSoln_dvode(problemData):
    """
    Purpose:
    Solves adiabatic-isobaric batch reactor problem using the scipy.integrate
    interface to DVODE. Note: DVODE is a variable-order, variable step-size
    BDF method.

    Arguments:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
                as problemData['initCond']; or list, 1-D, same shape as
                problemData['initCond']): vector of absolute tolerances
            for DVODE.
        problemData['relTol'] (float): scalar relative tolerance for
            for DVODE.
    Returns:
    solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
            len(problemData[’timePts’]), numpy.ndarray.shape[1] ==
            (len(problemData[’initCond’]) + 1)): Solution of problem;
        each time point is a row, each state variable is a column.

    """

    # Set up the integrator
    dvode = scipy.integrate.ode(adiabaticIsobaricBatch)
    dvode.set_integrator(’vode’,
                        method=’bdf’,
                        with_jacobian=True,
                        atol=problemData[’absTol’],
                        rtol=problemData[’relTol’],
                        first_step=1e-9,
                        nsteps=1e7)
    dvode.set_initial_value(problemData[’initCond’], 0)
    dvode.set_f_params(problemData)

    # Carry out the main integration loop
    solution = numpy.hstack((0, numpy.asarray(problemData[’initCond’],
                                              )))
    for t in problemData[’timePts’][1:]:
        if not dvode.successful():
            raise ArithmeticError(’DVODE step unsuccessful!’)
        dvode.integrate(t)
        solution = numpy.vstack((solution, numpy.hstack((dvode.t, dvode.y))))

    return solution

def redSoln_dvode(problemData, rednData):
    """
    Purpose:
    Solves the reduced adiabatic-isobaric batch reactor problem using the
    scipy.integrate interface to DVODE. Note: DVODE is a variable-order,
    variable step-size BDF method.

    Arguments:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData[’gas’] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData[’initCond’] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData[’timePts’] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData[’absTol’] (float; or numpy.ndarray, 1-D, same shape
                as problemData[’initCond’]; or list, 1-D, same shape as
                problemData[’initCond’]): vector of absolute tolerances
            for DVODE.
        problemData[’relTol’] (float): scalar relative tolerance for
        for DVODE.
rednData (dict): emulates C-style (or MATLAB-style) struct with following
        fields:
    rednData['P'] (numpy.ndarray, 2-D; or numpy.mat, 2-D; or list, 2-D):
        projection matrix used for na\"{i}ve projection-based model reduction
    rednData['origin'] (numpy.ndarray, 1-D, len(rednData['origin']) ==
            len(problemData['initCond']); or list, 1-D,
            len(rednData['origin']) == len(problemData['initCond'])):
        origin of reduced model

Returns:
solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
        len(problemData['timePts']), numpy.ndarray.shape[1] ==
        (len(problemData['initCond']) + 1)): Solution of problem;
    each time point is a row, each state variable is a column.

"""

# Rebind object passed through rednData to problemData; this
# object isn't modified within this function call scope (and below),
# but if the script ever changes such that this statement is no longer
# true, expect errors.
problemData['P'] = rednData['P']

# Calculate initial conditions for reduced model based on initial
# conditions for full model. Remember that 1-D numpy.ndarrays can be
# treated as row or column vectors (depending on context).
redInitCond = numpy.dot(problemData['P'],
              numpy.asarray(problemData['initCond']) -
              numpy.asarray(rednData['origin'])) + \
              numpy.asarray(rednData['origin'])

# Set up the integrator
dvode = scipy.integrate.ode(redAdiabaticIsobaricBatch)
dvode.set_integrator('vode',
                    method='bdf',
                    with_jacobian=True,
                    atol=problemData['absTol'],
                    rtol=problemData['relTol'],
                    first_step=1e-9,
                    nsteps=1e7)
dvode.set_initial_value(redInitCond, 0)
dvode.set_f_params(problemData)

# Carry out the main integration loop

solution = numpy.hstack((0, redInitCond))
for t in problemData['timePts'][1:]:
    if not dvode.successful():
        raise ArithmeticError('DVODE step unsuccessful!')
    dvode.integrate(t)
    solution = numpy.vstack((solution, numpy.hstack((dvode.t, dvode.y))))

return solution
```

```
def fullSoln_dassl(problemData):
    """
    Purpose:
    Solves adiabatic-isobaric batch reactor problem using the pydas
    interface to DASSL. Note: DASSL is a variable-order, variable step-size
    BDF method.

    Arguments:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
                as problemData['initCond']; or list, 1-D, same shape as
                problemData['initCond']): vector of absolute tolerances
            for DASSL.
        problemData['relTol'] (float): scalar relative tolerance for
            for DASSL.

    Returns:
    solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
            len(problemData['timePts']), numpy.ndarray.shape[1] ==
            (len(problemData['initCond']) + 1)): Solution of problem;
        each time point is a row, each state variable is a column.

    """

    # Define the residual and optional Jacobian matrix
    class Problem(pydas.DASSL):
        def residual(self, t, y, dydt):
            res = numpy.asarray(dydt) - \
                adiabaticIsobaricBatch(t,y,problemData)
            return res, 0

    # Set up the integrator
    dassl = Problem()
    dassl.initialize(0, problemData['initCond'],
        adiabaticIsobaricBatch(0, problemData['initCond'], problemData),
        atol=problemData['absTol'], rtol=problemData['relTol'])

    # Carry out the main integration loop
    solution = numpy.hstack((0, numpy.asarray(problemData['initCond'],
                                            )))
    t_max = problemData['timePts'][-1]
    #while dassl.t < t_max:
    #    dassl.step(t_max)
    #    solution = numpy.vstack((solution, numpy.hstack((dassl.t, dassl.y))))
    for t in problemData['timePts'][1:]:
        dassl.advance(t)
        solution = numpy.vstack((solution, numpy.hstack((dassl.t, dassl.y))))

    return solution

def redSoln_dassl(problemData, rednData):
    """
    Purpose:
    Solves the reduced adiabatic-isobaric batch reactor problem using the
    pydas interface to DASSL. Note: DASSL is a variable-order, variable
    step-size BDF method.

    Arguments:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData[’gas’] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData[’initCond’] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData[’timePts’] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData[’absTol’] (float; or numpy.ndarray, 1-D, same shape
                as problemData[’initCond’]; or list, 1-D, same shape as
                problemData[’initCond’]): vector of absolute tolerances
            for DASSL.
        problemData[’relTol’] (float): scalar relative tolerance for
            for DASSL.
    rednData (dict): emulates C-style (or MATLAB-style) struct with following
            fields:
        rednData[’P’] (numpy.ndarray, 2-D; or numpy.mat, 2-D; or list, 2-D):
            projection matrix used for na\"{i}ve projection-based model reduction
        rednData[’origin’] (numpy.ndarray, 1-D, len(rednData[’origin’]) ==
                len(problemData[’initCond’]); or list, 1-D,
                len(rednData[’origin’]) == len(problemData[’initCond’])):
            origin of reduced model

    Returns:
    solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
            len(problemData[’timePts’]), numpy.ndarray.shape[1] ==
            (len(problemData[’initCond’]) + 1)): Solution of problem;
        each time point is a row, each state variable is a column.

    """

    # Rebind object passed through rednData to problemData; this
    # object isn’t modified within this function call scope (and below),
    # but if the script ever changes such that this statement is no longer
    # true, expect errors.
    problemData[’P’] = rednData[’P’]

    # Calculate initial conditions for reduced model based on initial
    # conditions for full model. Remember that 1-D numpy.ndarrays can be
    # treated as row or column vectors (depending on context).
    redInitCond = numpy.dot(problemData['P'],
                  numpy.asarray(problemData['initCond']) -
                  numpy.asarray(rednData['origin'])) + \
                  numpy.asarray(rednData['origin'])

    # Define the residual and optional Jacobian matrix
    class Problem(pydas.DASSL):
        def residual(self, t, y, dydt):
            res = numpy.asarray(dydt) - \
                redAdiabaticIsobaricBatch(t,y,problemData)
            return res, 0

    # Set up the integrator
    dassl = Problem()
    dassl.initialize(0, redInitCond,
        redAdiabaticIsobaricBatch(0, redInitCond, problemData),
        atol=problemData['absTol'], rtol=problemData['relTol'])

    # Carry out the main integration loop
    solution = numpy.hstack((0, redInitCond))
    t_max = problemData['timePts'][-1]
    #while dassl.t < t_max:
    #    dassl.step(t_max)
    #    solution = numpy.vstack((solution, numpy.hstack((dassl.t, dassl.y))))
    for t in problemData['timePts'][1:]:
        dassl.advance(t)
        solution = numpy.vstack((solution, numpy.hstack((dassl.t, dassl.y))))

    return solution

def fullSoln_cvode(problemData):
    """
    Purpose:
    Solves adiabatic-isobaric batch reactor problem using the Assimulo
    interface to CVODE. Note: CVODE is a variable-order, variable
    step-size BDF method.

    Arguments:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
                as problemData['initCond']; or list, 1-D, same shape as
                problemData['initCond']): vector of absolute tolerances
            for CVODE.
        problemData['relTol'] (float): scalar relative tolerance for
            for CVODE.
    Returns:
    solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
            len(problemData[’timePts’]), numpy.ndarray.shape[1] ==
            (len(problemData[’initCond’]) + 1)): Solution of problem;
        each time point is a row, each state variable is a column.

    """

    # Define right-hand side that incorporates problemData because Assimulo
    # assumes parameters are floats (or numpy.ndarray of floats)
    def rhs(t, y):
        ydot = adiabaticIsobaricBatch(t,y,problemData)
        return ydot

    # Set up the integrator
    batchProblem = assimulo.problem.Explicit_Problem(rhs,
                                                     problemData[’initCond’],
                                                     0)
    cvode = assimulo.solvers.CVode(batchProblem)
    cvode.atol = problemData[’absTol’]
    cvode.rtol = problemData[’relTol’]
    cvode.maxsteps = 10000000
    cvode.inith = 1e-9
    cvode.discr = ’BDF’
    cvode.iter = ’Newton’

    # Carry out the main integration loop
    t_max = problemData[’timePts’][-1]
    n_pts = len(problemData[’timePts’])
    cvode_t, cvode_y = cvode.simulate(t_max, n_pts)
    solution = numpy.hstack((
        numpy.asarray([cvode_t]).transpose(),
        numpy.asarray(cvode_y)))

    return solution

def redSoln_cvode(problemData, rednData):
    """
    Purpose:
    Solves the reduced adiabatic-isobaric batch reactor problem using the
    Assimulo interface to CVODE. Note: CVODE is a variable-order, variable
    step-size BDF method.

    Arguments:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData[’gas’] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData[’initCond’] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData[’timePts’] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData[’absTol’] (float; or numpy.ndarray, 1-D, same shape
            as problemData['initCond']; or list, 1-D, same shape as
            problemData['initCond']): vector of absolute tolerances
        for CVODE.
    problemData['relTol'] (float): scalar relative tolerance for
        for CVODE.
rednData (dict): emulates C-style (or MATLAB-style) struct with following
        fields:
    rednData['P'] (numpy.ndarray, 2-D; or numpy.mat, 2-D; or list, 2-D):
        projection matrix used for na\"{i}ve projection-based model reduction
    rednData['origin'] (numpy.ndarray, 1-D, len(rednData['origin']) ==
            len(problemData['initCond']); or list, 1-D,
            len(rednData['origin']) == len(problemData['initCond'])):
        origin of reduced model

Returns:
solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
        len(problemData['timePts']), numpy.ndarray.shape[1] ==
        (len(problemData['initCond']) + 1)): Solution of problem;
    each time point is a row, each state variable is a column.

"""

# Rebind object passed through rednData to problemData; this
# object isn't modified within this function call scope (and below),
# but if the script ever changes such that this statement is no longer
# true, expect errors.
problemData['P'] = rednData['P']

# Calculate initial conditions for reduced model based on initial
# conditions for full model. Remember that 1-D numpy.ndarrays can be
# treated as row or column vectors (depending on context).
redInitCond = numpy.dot(problemData['P'],
              numpy.asarray(problemData['initCond']) -
              numpy.asarray(rednData['origin'])) + \
              numpy.asarray(rednData['origin'])

# Define right-hand side that incorporates problemData because Assimulo
# assumes parameters are floats (or numpy.ndarray of floats)
def rhs(t, y):
    ydot = redAdiabaticIsobaricBatch(t,y,problemData)
    return ydot

# Set up the integrator
batchProblem = assimulo.problem.Explicit_Problem(rhs,
                                                  problemData['initCond'],
                                                  0)
cvode = assimulo.solvers.CVode(batchProblem)
cvode.atol = problemData['absTol']
cvode.rtol = problemData['relTol']
cvode.maxsteps = 10000000
cvode.inith = 1e-9
cvode.discr = 'BDF'
cvode.iter = 'Newton'
```

```
    # Carry out the main integration loop
    t_max = problemData['timePts'][-1]
    n_pts = len(problemData['timePts'])
    cvode_t, cvode_y = cvode.simulate(t_max, n_pts)
    solution = numpy.hstack(
        (numpy.asarray([cvode_t]).transpose(),
        numpy.asarray(cvode_y)))

    return solution

def fullSoln_radau5(problemData):
    """
    Purpose:
    Solves adiabatic-isobaric batch reactor problem using the Assimulo
    interface to RADAU5. Note: RADAU5 is a fifth-order, three-stage
    implicit Runge-Kutta method based on Radau IIA quadrature, with
    variable step-size control.

    Arguments:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
                as problemData['initCond']; or list, 1-D, same shape as
                problemData['initCond']): vector of absolute tolerances
            for RADAU5.
        problemData['relTol'] (float): scalar relative tolerance for
            for RADAU5.

    Returns:
    solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
            len(problemData['timePts']), numpy.ndarray.shape[1] ==
            (len(problemData['initCond']) + 1)): Solution of problem;
        each time point is a row, each state variable is a column.

    """

    # Define right-hand side that incorporates problemData because Assimulo
    # assumes parameters are floats (or numpy.ndarray of floats)
    def rhs(t, y):
        ydot = adiabaticIsobaricBatch(t,y,problemData)
        return ydot

    # Set up the integrator
    batchProblem = assimulo.problem.Explicit_Problem(rhs,
                                                    problemData['initCond'],
                                                    0)
    radau5 = assimulo.solvers.Radau5ODE(batchProblem)
    radau5.atol = problemData['absTol']
    radau5.rtol = problemData['relTol']
    radau5.maxsteps = 10000000
    radau5.inith = 1e-9
    radau5.discr = 'BDF'
    radau5.iter = 'Newton'

    # Carry out the main integration loop
    t_max = problemData['timePts'][-1]
    n_pts = len(problemData['timePts'])
    radau5_t, radau5_y = radau5.simulate(t_max, n_pts)
    solution = numpy.hstack((
        numpy.asarray([radau5_t]).transpose(),
        numpy.asarray(radau5_y)))

    return solution

def redSoln_radau5(problemData, rednData):
    """
    Purpose:
    Solves the reduced adiabatic-isobaric batch reactor problem using the
    Assimulo interface to RADAU5. Note: RADAU5 is a fifth-order,
    three-stage implicit Runge-Kutta method based on Radau IIA quadrature,
    with variable step-size control.

    Arguments:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
                as problemData['initCond']; or list, 1-D, same shape as
                problemData['initCond']): vector of absolute tolerances
            for RADAU5.
        problemData['relTol'] (float): scalar relative tolerance for
            for RADAU5.
    rednData (dict): emulates C-style (or MATLAB-style) struct with following
            fields:
        rednData['P'] (numpy.ndarray, 2-D; or numpy.mat, 2-D; or list, 2-D):
            projection matrix used for na\"{i}ve projection-based model reduction
        rednData['origin'] (numpy.ndarray, 1-D, len(rednData['origin']) ==
                len(problemData['initCond']); or list, 1-D,
                len(rednData['origin']) == len(problemData['initCond'])):
            origin of reduced model

    Returns:
    solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
            len(problemData['timePts']), numpy.ndarray.shape[1] ==
            (len(problemData['initCond']) + 1)): Solution of problem;
        each time point is a row, each state variable is a column.

    """

    # Rebind object passed through rednData to problemData; this
    # object isn't modified within this function call scope (and below),
    # but if the script ever changes such that this statement is no longer
    # true, expect errors.
    problemData['P'] = rednData['P']

    # Calculate initial conditions for reduced model based on initial
    # conditions for full model. Remember that 1-D numpy.ndarrays can be
    # treated as row or column vectors (depending on context).
    redInitCond = numpy.dot(problemData['P'],
                  numpy.asarray(problemData['initCond']) -
                  numpy.asarray(rednData['origin'])) + \
                  numpy.asarray(rednData['origin'])

    # Define right-hand side that incorporates problemData because Assimulo
    # assumes parameters are floats (or numpy.ndarray of floats)
    def rhs(t, y):
        ydot = redAdiabaticIsobaricBatch(t,y,problemData)
        return ydot

    # Set up the integrator
    batchProblem = assimulo.problem.Explicit_Problem(rhs,
                                                     problemData['initCond'],
                                                     0)
    radau5 = assimulo.solvers.Radau5ODE(batchProblem)
    radau5.atol = problemData['absTol']
    radau5.rtol = problemData['relTol']
    radau5.maxsteps = 10000000
    radau5.inith = 1e-9
    radau5.discr = 'BDF'
    radau5.iter = 'Newton'

    # Carry out the main integration loop
    t_max = problemData['timePts'][-1]
    n_pts = len(problemData['timePts'])
    radau5_t, radau5_y = radau5.simulate(t_max, n_pts)
    solution = numpy.hstack(
        (numpy.asarray([radau5_t]).transpose(),
        numpy.asarray(radau5_y)))

    return solution

def fullSoln_rodas(problemData):
    """
    Purpose:
    Solves adiabatic-isobaric batch reactor problem using the Assimulo
    interface to RODAS. Note: RODAS is a third-order Rosenbrock method
    (diagonally implicit Runge-Kutta) with variable step-size control.

    Arguments:
```

```
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
                as problemData['initCond']; or list, 1-D, same shape as
                problemData['initCond']): vector of absolute tolerances
            for RODAS.
        problemData['relTol'] (float): scalar relative tolerance for
            for RODAS.

    Returns:
    solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
            len(problemData['timePts']), numpy.ndarray.shape[1] ==
            (len(problemData['initCond']) + 1)): Solution of problem;
        each time point is a row, each state variable is a column.

    """

    # Define right-hand side that incorporates problemData because Assimulo
    # assumes parameters are floats (or numpy.ndarray of floats)
    def rhs(t, y):
        ydot = adiabaticIsobaricBatch(t,y,problemData)
        return ydot

    # Set up the integrator
    batchProblem = assimulo.problem.Explicit_Problem(rhs,
                                                      problemData['initCond'],
                                                      0)
    rodas = assimulo.solvers.RodasODE(batchProblem)
    rodas.atol = problemData['absTol']
    rodas.rtol = problemData['relTol']
    rodas.maxsteps = 10000000
    rodas.inith = 1e-9
    rodas.discr = 'BDF'
    rodas.iter = 'Newton'

    # Carry out the main integration loop
    t_max = problemData['timePts'][-1]
    n_pts = len(problemData['timePts'])
    rodas_t, rodas_y = rodas.simulate(t_max, n_pts)
    solution = numpy.hstack((
        numpy.asarray([rodas_t]).transpose(),
        numpy.asarray(rodas_y)))

    return solution

def redSoln_rodas(problemData, rednData):
    """
```

```
Purpose:
Solves the reduced adiabatic-isobaric batch reactor problem using the
Assimulo interface to RODAS. Note: RODAS is a third-order Rosenbrock
method (diagonally implicit Runge-Kutta) with variable step-size control.

Arguments:
problemData (dict): emulates C-style (or MATLAB-style) struct with
        following fields:
    problemData['gas'] (Cantera.Solution): object containing chemistry
        and gas physical properties
    problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
        vector of state variables
    problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
        at which the solution should be calculated; first time point
        corresponds to initial conditions!
    problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
            as problemData['initCond']; or list, 1-D, same shape as
            problemData['initCond']): vector of absolute tolerances
        for RODAS.
    problemData['relTol'] (float): scalar relative tolerance for
        for RODAS.
rednData (dict): emulates C-style (or MATLAB-style) struct with following
        fields:
    rednData['P'] (numpy.ndarray, 2-D; or numpy.mat, 2-D; or list, 2-D):
        projection matrix used for na\"{i}ve projection-based model reduction
    rednData['origin'] (numpy.ndarray, 1-D, len(rednData['origin']) ==
            len(problemData['initCond']); or list, 1-D,
            len(rednData['origin']) == len(problemData['initCond'])):
        origin of reduced model

Returns:
solution (numpy.ndarray, 2-D, numpy.ndarray.shape[0] ==
        len(problemData['timePts']), numpy.ndarray.shape[1] ==
        (len(problemData['initCond']) + 1)): Solution of problem;
    each time point is a row, each state variable is a column.

"""

# Rebind object passed through rednData to problemData; this
# object isn't modified within this function call scope (and below),
# but if the script ever changes such that this statement is no longer
# true, expect errors.
problemData['P'] = rednData['P']

# Calculate initial conditions for reduced model based on initial
# conditions for full model. Remember that 1-D numpy.ndarrays can be
# treated as row or column vectors (depending on context).
redInitCond = numpy.dot(problemData['P'],
              numpy.asarray(problemData['initCond']) -
              numpy.asarray(rednData['origin'])) + \
              numpy.asarray(rednData['origin'])

# Define right-hand side that incorporates problemData because Assimulo
# assumes parameters are floats (or numpy.ndarray of floats)
```

```
    def rhs(t, y):
        ydot = redAdiabaticIsobaricBatch(t,y,problemData)
        return ydot

    # Set up the integrator
    batchProblem = assimulo.problem.Explicit_Problem(rhs,
                                                     problemData['initCond'],
                                                     0)
    rodas = assimulo.solvers.RodasODE(batchProblem)
    rodas.atol = problemData['absTol']
    rodas.rtol = problemData['relTol']
    rodas.maxsteps = 10000000
    rodas.inith = 1e-9
    rodas.discr = 'BDF'
    rodas.iter = 'Newton'

    # Carry out the main integration loop
    t_max = problemData['timePts'][-1]
    n_pts = len(problemData['timePts'])
    rodas_t, rodas_y = rodas.simulate(t_max, n_pts)
    solution = numpy.hstack(
        (numpy.asarray([rodas_t]).transpose(),
        numpy.asarray(rodas_y)))

    return solution

def calcRedModelParams(fullSolution, problemData):
    """
    Purpose:
    Calculate projector based on data from full model solution.

    Arguments:
    fullSolution (numpy.ndarray, 2-D; or list, 2-D): solution of full model
    problemData (dict): emulates C-style (or MATLAB-style) struct with
           following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties

    Returns:
    origin_index (int): time point index corresponding to origin data point
    origin (numpy.ndarray, 1-D, where origin.shape[0] ==
            fullSolution.shape[1]):
        origin of projection-based reduced model (transposed, for convenience)
    projector (numpy.ndarray, 2-D, where projector.shape[0] ==
            projector.shape[1] == fullSolution.shape[1]): projection matrix
        for projection-based model reduction.
    W (numpy.ndarray, 2-D, where W.shape[0] == (fullSolution.shape[1] - 1)):
        so-called "lumping matrix"; projection nullspace is perpendicular to
        span of this matrix
    orthoBasis (numpy.ndarray, 2-D, where orthoBasis.shape[0] ==
            orthoBasis.shape[1] == (fullSolution.shape[1] - 1)):
        orthonormal basis such that its first two columns correspond to the
        range of the projector, and its last two columns correspond to the
        nullspace of the projector.
    """

    # The philosophy here was to find a nice point in the full model solution
    # to serve as the origin. The two basis vectors that span the range space
    # were the unit vector [[1,0,0,0]].transpose(), which corresponds to the
    # "temperature direction", and the tangent vector of the full model
    # solution (i.e., the right-hand side of the full model ODE). These basis
    # vectors are used to construct an orthogonal projector.
    # IF YOU WANT TO MODIFY THE PROJECTOR, YOU MUST MODIFY THE INTERNALS OF
    # THIS FUNCTION!

    # The origin will be the point in the solution set calculated immediately
    # before the first point in the solution set calculated that exceeds a
    # cutoff temperature.
    cutoff_temp = 2263 # Kelvin
    origin_index = numpy.flatnonzero(fullSolution[:, 1] > cutoff_temp)[0] - 1
    origin = fullSolution[origin_index, 1:]

    # Reminder:
    # In the MATLAB script, the first column of V corresponds to the "lump",
    # and the second column corresponds to temperature. In this Python script,
    # the first column corresponds to temperature, and the second corresponds
    # to the "lump". In order to obtain the proper V matrix in Python, the 2
    # input basis vectors must be specified in the reverse order of the 2
    # input basis vectors specified in MATLAB.

    # Having determined the origin, a basis must be constructed in order to
    # calculate a projector. The first basis vector is going to be the
    # right-hand side of the full model, evaluated at the origin. The second
    #  basis vector is going to be [[1,0,0,0]].transpose(). The basis matrix
    # must consist of column vectors in order to carry out the necessary
    # linear algebra.
    first_range_vec = numpy.asarray([1, 0, 0, 0])
    second_range_vec = adiabaticIsobaricBatch(0, origin, problemData)
    basis = numpy.vstack((first_range_vec, second_range_vec)).transpose()

    # An orthogonal projector is constructed from this basis by
    # orthonormalizing it.
    [orthoBasis, _] = scipy.linalg.qr(basis)
    V = orthoBasis[:, 0:2]
    W = copy.copy(V)
    projector = numpy.dot(V, W.transpose())

    return origin_index, origin, projector, W, orthoBasis

def lump_soln(soln, W, origin, origin_index):
    """
    Purpose:
    From a solution in the original state variables, calculate a "lumped"
    or "Petrov-Galerkin projected" solution.

    Arguments:
    soln (numpy.ndarray, 2-D): Solution of adiabatic-isobaric batch reactor
```

```
        problem; each time point is a row, each state variable is a column.
    W (numpy.ndarray, 2-D, where W.shape[0] == (soln.shape[1] - 1)):
        so-called "lumping matrix"; projection nullspace is perpendicular to
        span of this matrix
    origin (numpy.ndarray, 1-D, where origin.shape[0] ==
            fullSolution.shape[1]):
        origin of projection-based reduced model (transposed, for convenience)
    origin_index (int): time point index corresponding to origin data point

    Returns:
    lumped_soln (numpy.ndarray, 2-D, where lumped_soln.shape[0] ==
            soln.shape[0] and lumped_soln.shape[1] == (W.shape[1] + 1)):
        lumped version of soln

    """

    # Calculate lumped model solution
    #Copy time data points
    # Since each data point is a row, instead of calculating
    # W^{T} * (y - y_{0}), calculate (y - y_{0})^{T} * W.
    lumped_soln = numpy.hstack((numpy.asarray([soln[:, 0]]).transpose(),
        numpy.dot((soln[:, 1:] - numpy.tile(origin, (soln.shape[0], 1))), W)))

    return lumped_soln

def setProblemData():
    """
    Purpose:
    Sets problem parameters.

    Arguments:
    None

    Returns:
    problemData (dict): emulates C-style (or MATLAB-style) struct with
            following fields:
        problemData['gas'] (Cantera.Solution): object containing chemistry
            and gas physical properties
        problemData['initCond'] (numpy.ndarray, 1-D; or list, 1-D): (row)
            vector of state variables
        problemData['timePts'] (numpy.ndarray, 1-D; or list, 1-D): times
            at which the solution should be calculated; first time point
            corresponds to initial conditions!
        problemData['absTol'] (float; or numpy.ndarray, 1-D, same shape
                as problemData['initCond']; or list, 1-D, same shape as
                problemData['initCond']): vector of absolute tolerances
            for DVODE.
        problemData['relTol'] (float): scalar relative tolerance for
            for DVODE.

    """

    # Set up problem parameters.
    # IF YOU WANT TO CHANGE THE FULL MODEL SOLUTION (AND ALL THE OTHERS),
    # CHANGE THE PARAMETERS HERE!
    problemData = {}
    problemData['gas'] = Cantera.IdealGasMix('ozone.cti')
    initialTemperature = 1000
    initialMoleFracString = 'O:0, O2:0.15, O3:0.85'
    problemData['gas'].set(T=initialTemperature,
                           P=Cantera.OneAtm,
                           Y=initialMoleFracString)
    problemData['timePts'] = numpy.linspace(0, 2e-5, 10000)

    # From the problem parameters, repackage the data so that it can be
    # passed to ODE solvers.
    problemData['initCond'] = numpy.hstack((
        numpy.asarray(initialTemperature),
        numpy.asarray(problemData['gas'].massFractions())))

    # Appropriate error tolerances for ODE solvers like DVODE, CVODE,
    # RADAU5, and RODAS
    #problemData['absTol'] = 1e-15
    #problemData['relTol'] = 1e-12

    # Appropriate error tolerances for DAE solvers like DASSL
    problemData['absTol'] = 1e-7
    problemData['relTol'] = 1e-7

    return problemData

def CalculateFullRedAndLumpedSolns():
    """
    Purpose:
    Calculate three different solutions for an ozone flame:
    - Full model solution
    - Reduced model solution, reduced using projection-based model reduction
    - Lumped model solution (or Petrov-Galerkin projection), derived from
      reduced model solution.

    The basic idea is to decouple the calculation of solutions from the
    plotting of figures so that the functions in this file are of a
    manageable size.

    Arguments:
    None.

    Returns:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame
    origLumpedSoln (numpy.ndarray, 2-D): lumped version of full model solution
    redLumpedSoln (numpy.ndarray, 2-D): lumped model solution for ozone flame
    rednData (dict): emulates C-style (or MATLAB-style) struct with following
            fields:
        rednData['P'] (numpy.ndarray, 2-D; or numpy.mat, 2-D; or list, 2-D):
            projection matrix used for na\"{i}ve projection-based model reduction
        rednData['origin'] (numpy.ndarray, 1-D, len(rednData['origin']) ==
                len(problemData['initCond']); or list, 1-D,
                len(rednData['origin']) == len(problemData['initCond'])):
            origin of reduced model
    origin_index (int): time point index corresponding to origin data point
    orthoBasis (numpy.ndarray, 2-D, where orthoBasis.shape[0] ==
            orthoBasis.shape[1] == (fullSolution.shape[1] - 1)):
        orthonormal basis such that its first two columns correspond to the
        range of the projector, and its last two columns correspond to the
        nullspace of the projector.
    """

    #Set up problem data
    problemData = setProblemData()

    # Calculate full model solution
    fullSolution = fullSoln(problemData)

    # From the full model solution, calculate a projector.
    # IF YOU WANT TO CHANGE THE LUMPED AND REDUCED MODEL SOLUTIONS,
    # CHANGE THE INTERNALS OF calculateProjector
    (origin_index,
     origin,
     projector,
     W, orthoBasis) = calcRedModelParams(fullSolution, problemData)

    # Calculate reduced model solution
    # Rows are system states at a given time
    # Columns are single state variables (or time)
    redProblemData = setProblemData()
    redProblemData['initCond'] = origin
    rednData = {'origin': origin, 'P': projector}
    redSolution = redSoln(redProblemData, rednData)

    # Calculate "lumping" (or Petrov-Galerkin projection) of original and
    # reduced models
    origLumpedSoln = lump_soln(fullSolution, W, origin, origin_index)
    redLumpedSoln = lump_soln(redSolution, W, origin, origin_index)

    # Correct for the time discrepancy of the full and reduced models
    redSolution[:,0] += fullSolution[origin_index, 0]
    redLumpedSoln[:,0] += fullSolution[origin_index, 0]

    return (fullSolution, redSolution, origLumpedSoln, redLumpedSoln,
         rednData, origin_index, orthoBasis)

def plot_temp(fullSolution, redSolution):
    """
    Purpose:
    Make temperature versus time plots that compare the full and
    reduced model solutions.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame
    Returns:
    temp_fig (matplotlib.figure.Figure): temperature versus time plot

    """

    temp_fig = matplotlib.pyplot.figure()
    matplotlib.pyplot.plot(fullSolution[:,0], fullSolution[:,1], ’b-’)
    matplotlib.pyplot.plot(redSolution[:, 0], redSolution[:, 1], ’r--’)
    matplotlib.pyplot.ticklabel_format(axis=’both’, scilimits=(-2,3))
    matplotlib.pyplot.xlabel(’Time [s]’)
    matplotlib.pyplot.ylabel(’Temperature [K]’)
    matplotlib.pyplot.title(’Cantera simulation: Temperature profile’)
    matplotlib.pyplot.legend( (’Original model’, ’Reduced model’), loc=’best’)

    return temp_fig

def plot_o(fullSolution, redSolution):
    """
    Purpose:
    Make mass fraction oxygen atoms versus time plots that compare the full and
    reduced model solutions.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame

    Returns:
    o_fig (matplotlib.figure.Figure): mass fraction O atoms versus time plot

    """
    o_fig = matplotlib.pyplot.figure()
    matplotlib.pyplot.plot(fullSolution[:, 0], fullSolution[:, 2], ’b-’)
    matplotlib.pyplot.plot(redSolution[:, 0], redSolution[:, 2], ’r--’)
    matplotlib.pyplot.ticklabel_format(axis=’both’, scilimits=(-2,3))
    matplotlib.pyplot.xlabel(’Time [s]’)
    matplotlib.pyplot.ylabel(’Mass Fraction O [a.u.]’)
    matplotlib.pyplot.title(’Cantera simulation: Mass Fraction O profile’)
    matplotlib.pyplot.legend( (’Original model’, ’Reduced model’), loc=’best’)

    return o_fig

def plot_o2(fullSolution, redSolution):
    """
    Purpose:
    Make mass fraction O2 versus time plots that compare the full and
    reduced model solutions.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame

    Returns:
    o2_fig (matplotlib.figure.Figure): mass fraction O2 versus time plot
    """

    o2_fig = matplotlib.pyplot.figure()
    matplotlib.pyplot.plot(fullSolution[:, 0], fullSolution[:, 3], 'b-')
    matplotlib.pyplot.plot(redSolution[:, 0], redSolution[:, 3], 'r--')
    matplotlib.pyplot.ticklabel_format(axis='both', scilimits=(-2,3))
    matplotlib.pyplot.xlabel('Time [s]')
    matplotlib.pyplot.ylabel('Mass Fraction O2 [a.u.]')
    matplotlib.pyplot.title('Cantera simulation: Mass Fraction O2 profile')
    matplotlib.pyplot.legend( ('Original model', 'Reduced model'), loc='best')

    return o2_fig

def plot_o3(fullSolution, redSolution):
    """
    Purpose:
    Make mass fraction O3 versus time plots that compare the full and
    reduced model solutions.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame

    Returns:
    o3_fig (matplotlib.figure.Figure): mass fraction O3 versus time plot

    """

    o3_fig = matplotlib.pyplot.figure()
    matplotlib.pyplot.plot(fullSolution[:, 0], fullSolution[:, 4], 'b-')
    matplotlib.pyplot.plot(redSolution[:, 0], redSolution[:, 4], 'r--')
    matplotlib.pyplot.ticklabel_format(axis='both', scilimits=(-2,3))
    matplotlib.pyplot.xlabel('Time [s]')
    matplotlib.pyplot.ylabel('Mass Fraction O3 [a.u.]')
    matplotlib.pyplot.title('Cantera simulation: Mass Fraction O3 profile')
    matplotlib.pyplot.legend( ('Original model', 'Reduced model'), loc='best')

    return o3_fig

def plot_projector_rep(fullSolution, redSolution, orthoBasis, origin):
    """
    Purpose:
    Make phase plot that compares the full and reduced model solutions using
    the projector representation.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame
    orthoBasis (numpy.ndarray, 2-D, where orthoBasis.shape[0] ==
            orthoBasis.shape[1] == (fullSolution.shape[1] - 1)):
        orthonormal basis such that its first two columns correspond to the
        range of the projector, and its last two columns correspond to the
        nullspace of the projector
    origin (numpy.ndarray, 1-D, where origin.shape[0] ==
        fullSolution.shape[1]):
    origin of projection-based reduced model (transposed, for convenience)

Returns:
proj_rep_fig (matplotlib.figure.Figure): phase plot (O_2, O_3, O)
    comparing solutions of full model and projector representation of
    reduced model

"""
# Phase plot of the solutions of the full and reduced models
proj_rep_fig = matplotlib.pyplot.figure()
axes = proj_rep_fig.gca(projection='3d')
axes.plot(fullSolution[:,3], fullSolution[:,4], fullSolution[:,2], 'b--')
axes.plot(redSolution[:,3], redSolution[:,4], redSolution[:,2], 'r-')

# Set up the grid of (x,y) points for a plane to guide the eye
n_pts = 20
x = numpy.linspace(0, 1, n_pts)
y = numpy.linspace(0, .5, n_pts)
X, Y = numpy.meshgrid(x, y)

# Set up the color of the plane
plane_color = 'orange'
plane_face_colors = numpy.empty(X.shape, dtype='|S'+str(len(plane_color)))
plane_face_colors.fill(plane_color)

# Since the plots permute the order of the solution matrix entries,
# the basis entries and origin entries must also be permuted in a
# consistent manner
axis_permutation = [2, 3, 1]
plane_origin = origin[axis_permutation]

# The basis_index column of orthoBasis corresponds to the important
# "lumping" direction. This column is used to determine the normal
# vector of the plane in this figure that guides the eye.
basis_index = 1
normal = numpy.asarray([0,
    -orthoBasis[3, basis_index]/orthoBasis[1, basis_index],
    0,
    1])
normal = normal[axis_permutation]
normal = normal / numpy.linalg.norm(normal, 2)

# Once the origin of the plane and the normal of the plane are determined,
# the z coordinates of the plane are determined using analytic geometry.
Z = plane_origin [2] - (normal[0] * (X - plane_origin[0]) +
    normal[1] * (Y - plane_origin[1])) / normal[2]

# Plot the (x,y,z) coordinates of the plane that guides the eye
plane = axes.plot_surface(X, Y, Z, facecolors=plane_face_colors,
                          shade=0, alpha=.4)
plane.set_edgecolors('none')

# Add legend, axis labels, title, etc.
```

```
    axes.set_title('Projector Representation: Ozone')
    axes.legend( ('Original model', 'Reduced model'), loc='best')
    axes.set_xlabel(r'Mass Frac O$_2$')
    axes.set_xlim(0, 1)
    axes.set_ylabel(r'Mass Frac O$_3$')
    axes.set_ylim(0, 1)
    axes.set_zlabel(r'Mass Frac O')
    axes.set_zlim(0, .025)
    #axes.view_init(elev=-69, azim=42)
    axes.grid()

    return proj_rep_fig

def plot_lumped_rep(fullSolution, redSolution, origLumpedSoln,
                    redLumpedSoln, origin_index):
    """
    Purpose:
    Make phase plot that compares the full and reduced model solutions using
    the lumped representation.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame
    origLumpedSoln (numpy.ndarray, 2-D): lumped full model solution for ozone
        flame
    redLumpedSoln (numpy.ndarray, 2-D): lumped reduced model solution for ozone
        flame
    origin_index (float): value of first index of fullSolution[:,:]
        corresponding to the origin of the reduced model

    Returns:
    lumped_rep_fig (matplotlib.figure.Figure): phase plot comparing solutions
        of lumped full model and lumped representation of reduced model
    """
    lumped_rep_fig = matplotlib.pyplot.figure()
    axes = lumped_rep_fig.gca(projection='3d')

    # Reminder:
    # In the MATLAB script, the first column of V corresponds to the "lump",
    # and the second column corresponds to temperature. In this Python script,
    # the first column corresponds to temperature, and the second corresponds
    # to the "lump".

    # Note: Time zero now corresponds to origin for both solutions in this plot
    axes.plot(origLumpedSoln[origin_index:,2],
              origLumpedSoln[origin_index:,0] - origLumpedSoln[origin_index,0],
              fullSolution[origin_index:,2],
              'b--')
    axes.plot(redLumpedSoln[:,2],
              redLumpedSoln[:,0] - redLumpedSoln[0,0],
              redSolution[:,2],
              'r-')
    axes.set_title('Lumped Representation: Ozone')
    axes.legend( ('Original Model', 'Reduced Model'), loc='best')
    axes.set_xlabel(r'$\alpha \cdot$ Mass Frac O ' +
                    r'$+ \beta \cdot$ Mass Frac O$_2$ '+
                    r'$+ \gamma \cdot$ Mass Frac O$_3$')
    axes.set_ylabel('Time [s]')
    axes.set_zlabel('Mass Frac O')
    #axes.view_init(elev=-160, azim=22)
    axes.grid()

    return lumped_rep_fig

def plot_invariant_rep(fullSolution, redSolution, origin_index):
    """
    Purpose:
    Make phase plot that compares the full and reduced model solutions using
    the invariant representation.

    Arguments:
    fullSolution (numpy.ndarray, 2-D): full model solution for ozone flame
    redSolution (numpy.ndarray, 2-D): reduced model solution for ozone flame
    origin_index (float): value of first index fullSolution[:,:]
        corresponding to the origin of the reduced model

    Returns:
    invariant_rep_fig (matplotlib.figure.Figure): phase plot comparing
        solutions of invariant representations of full and reduced models
    """
    invariant_rep_fig = matplotlib.pyplot.figure()
    axes = invariant_rep_fig.gca(projection='3d')
    # Note: Time zero now corresponds to origin for both solutions in this plot
    axes.plot(fullSolution[origin_index:,4],
              fullSolution[origin_index:,0] - fullSolution[origin_index, 0],
              fullSolution[origin_index:,2],
              'b--')
    axes.plot(redSolution[:,4],
              redSolution[:,0] - redSolution[0,0],
              redSolution[:,2],
              'r-')
    axes.set_title('Invariant Representation: Ozone')
    axes.legend( ('Original Model', 'Reduced Model'), loc='best')
    axes.set_xlabel(r'Mass Frac O$_3$')
    axes.set_ylabel(r'Time [s]')
    axes.set_zlabel(r'Mass Frac O')
    #axes.view_init(elev=11, azim=10)
    axes.grid()

    return invariant_rep_fig

def main_function():
    """
    Purpose:
    Main driver function.

    Arguments:
    None.
    Returns:
    None.

    """

    # Main program:

    (fullSolution,
     redSolution,
     origLumpedSoln,
     redLumpedSoln,
     rednData,
     origin_index, orthoBasis) = CalculateFullRedAndLumpedSolns()

    temp_fig = plot_temp(fullSolution, redSolution)
    o_fig = plot_o(fullSolution, redSolution)
    o2_fig = plot_o2(fullSolution, redSolution)
    o3_fig = plot_o3(fullSolution, redSolution)
    proj_rep_fig = plot_projector_rep(fullSolution, redSolution, orthoBasis,
                                      rednData['origin'])
    lumped_rep_fig = plot_lumped_rep(fullSolution, redSolution, origLumpedSoln,
                    redLumpedSoln, origin_index)
    invariant_rep_fig = plot_invariant_rep(fullSolution, redSolution,
                                           origin_index)

    matplotlib.pyplot.show()

    return

if __name__ == "__main__":
    main_function()
```

Appendix B

Implementation of Examples for Chapter 3

Examples for Chapter 3 were implemented in MATLAB r2012a [133] and in Python 2.7.3 [209].

B.1 MATLAB Implementation

The MATLAB r2012a [133] implementation requires the installation of Sundials 2.4 (or later), and SundialsTB [85].

```
function IllustrativeCaseStudy()
% Purpose: Case study that attempts to re-engineer Linda Petzold's example.
% Inputs: None.
% Outputs: None.
% Assertion checks: None.

% TODO(goxberry@gmail.com): Add unit tests.

% Close all plots
close all;

% Get the default random number generation stream in MATLAB, and reset it
% for reproducibility. See "Loren on the Art of MATLAB", November 5, 2008,
% http://blogs.mathworks.com/loren/2008/11/05/ (continued on next line)
% new-ways-with-random-numbers part i/
stream0 = RandStream('mt19937ar','Seed',0);
RandStream.setDefaultStream(stream0);

% "A" matrix from Rathinam and Petzold, "A New Look at Proper Orthogonal
% Decomposition", SINUM, Vol. 41, No. 5, pp. 1893-1925 (2004).
petzoldA1 = [-0.1, 0, 0;
              0, -0.1732, 2;
              0, -2.0, -0.1732;];

petzoldA2 = [-1.0, 0, 0;
             0, -1.226, -0.7080;
             0, 0.7080, -1.226;];

petzoldA12 = [0.3893, 0.5179, -1.543;
              1.390, 1.3, 0.8841;
              0.06293, -0.9078, -1.184;];

petzoldA = [petzoldA1, petzoldA12;
            zeros(3,3), petzoldA2];


% Use symmetrized Petzold matrix with spectrum shifted downward, so that
% spectrum is real, and its logarithmic 2-norm is negative. Set range basis
% to last three eigenvectors (which have nonzero entries in their last
% three components). Set nullspace equal to the first three standard
% Euclidean basis vectors.
symmA = petzoldA;
symmA(2,3) = -2;
symmA(6,5) = -0.7080;
symmA = symmA - 2 * eye(6,6);
[eigenVec, eigenVal] = eig(symmA);
rangeBasis = eigenVec(:,4:6);
nullBasis = [zeros(3,3); eye(3,3)];

% Generate random initial condition.
% initCond = rand(1,6);
initCond = ones(1,6);

% Run case study on this coefficient matrix and choice of bases
comparisonOfModels(symmA, rangeBasis, nullBasis, initCond);

% Scale the upper right block of the symmetrized, shifted Petzold matrix,
% which corresponds to halving gamma in our bounds.
lowerGammaA = symmA;
lowerGammaA(1:3,4:6) = .5 * symmA(1:3,4:6);
[eigenVec, eigenVal] = eig(lowerGammaA);
rangeBasis = eigenVec(:,4:6);
comparisonOfModels(lowerGammaA, rangeBasis, nullBasis, ...
    initCond);

% Scale the upper right block of the symmetrized, shifted Petzold matrix,
% which corresponds to halving gamma in our bounds.
higherMuA = symmA;
higherMuA(4:6,4:6) = .715 * symmA(4:6,4:6);
[eigenVec, eigenVal] = eig(higherMuA);
rangeBasis = eigenVec(:,4:6);
comparisonOfModels(higherMuA, rangeBasis, nullBasis, ...
    initCond);
```

```
end

function dy = linSys(t,y,A)
% Purpose: Function encoding linear ordinary differential equation system
% for numerical integration.
% Inputs: t = time
%         y = state vector, n by 1
%         A = coefficient matrix, n by n, for linear ordinary differential
%         equation \dot{y}(t) = A*y(t)
% Outputs: dy = right-hand side of differential equation \dot{y}(t) = ...
%               A*y(t)
% Assertion checks: None; check for conformality of A and y.
dy = A * y;
end

function comparisonOfModels(coeffMatrix, rangeBasis, nullBasis, initCond)
% Purpose: Integrates the linear system \dot{\mathbf{x}} = coeffMatrix *
% \mathbf{x}, and also integrates the projected system
% \dot{\hat{\mathbf{x}}} = \mathbf{P} * coeffMatrix * \hat{\mathbf{x}}.
% \mathbf{P} is a projection matrix that has range equal to the space
% spanned by the columns of rangeBasis, and null space equal to the space
% spanned by the columns of nullBasis.
% Input: coeffMatrix = n by n, coefficient matrix of linear ODE system; n
%                      should be consistent with rangeBasis
%        rangeBasis = n by k, columns determine range space of projection
%                     matrix; n and k are determined by size of matrix
%        nullBasis = n by (n-k), columns determine null space of projection
%                    matrix; n-k should be consistent with values
%                    determined by previous two arguments
%        initCond = 1 by n, initial condition for integration; row vector
%                   due to peculiarities of MATLAB syntax
% Output: None to scope outside of call. Will output information relevant
% to error bounds to terminal, either as text, or as plots.
% Assertion checks: None! Doesn't check for consistency, although error
% checks could be added later. NOTE: Many plots assume that n = 6; these
% can be generalized.

n = size(rangeBasis,1);
k = size(rangeBasis,2);

% Calculate Petrov-Galerkin matrices and projector. Use G. W. (Pete)
% Stewart's "On the Numerical Analysis of Oblique Projectors", SIMAX,
% Vol. 32, No. 1, pp. 309-348 (2011), to guide algorithms.
[X, Y, N, Xperp, Yperp, Nc] = CalcXQRYRep(rangeBasis, nullBasis);

% Stewart warns against calculating projection matrices directly, due
% to possible numerical error, but here, it is needed for some performance
% metrics (the norm of P).
P = CalcExplicitProjectors(X, Y, N);

% Calculate V, W, Vperp, Wperp where norm(W) = norm(Wperp) = 1.
[V, W, Vperp, Wperp] = CalcNorm1WRep(X, Y, N, Xperp, Yperp, Nc);
```

```
% Calculate V, W, Vperp, Wperp where norm(V) = norm(Vperp) = 1;
[Vprime, Wprime, VprimePerp, WprimePerp] =...
    CalcNorm1VRep(X, Y, N, Xperp, Yperp, Nc);

% Calculate constants in Theorem 4.1 and Corollary 4.4:
% Original choice of V, W, Vperp, Wperp
[gamma, muBar] = CalcThm4_1Consts(coeffMatrix, V, W, Wperp);

% Constants after change of basis for V, W, Vperp, and Wperp
[gammaPrime, muBarPrime] = CalcThm4_1Consts(coeffMatrix, Vprime, ...
    Wprime, WprimePerp);

% Cor. 4.4: Constants using projection matrix instead
[gammaProj, muBarProj] = CalcThm4_4Consts(coeffMatrix, X, Y, Wperp);

% Now, to simulate full and reduced system, emulating Rathinam and Petzold,
% "A New Look at Proper Orthogonal Decomposition", SINUM, Vol. 41, No. 5,
% pp. 1893-1925 (2004).
T = 5;
[epsilon, inSubSupNorm, inSub2Norm, totErr2Norm] = ...
    CalcErrors(T, initCond, coeffMatrix, X, Y, Xperp, Yperp);

% Calculate error bounds
[inSubSupNormBound, inSub2NormBound, totErr2NormBound] = ...
    CalculateErrorBoundsThm4_1(epsilon, gamma, muBar, T, V, Vperp);

% Results.
fprintf(1, '-----------------------------------------------------------\n')
fprintf(1, 'Size of matrix A (n by n), n = %e\n', n)
fprintf(1, 'Size of reduced order model, k = %e\n', k)
fprintf(1, 'gamma = %e\n', gamma);
fprintf(1, 'muBar = %e\n', muBar);
fprintf(1, '2-norm of P = %e\n', norm(P));
fprintf(1, '2-norm of V (should be 1) = %e\n', norm(V));
fprintf(1, '2-norm of W = %e\n', norm(W));
fprintf(1, 'gammaPrime (gamma under change of basis) = %e\n', gammaPrime);
fprintf(1, 'muBarPrime (muBar under change of basis) = %e\n', muBarPrime);
fprintf(1, '2-norm of Vprime = %e\n', norm(Vprime));
fprintf(1, '2-norm of Wprime = %e\n', norm(Wprime));
fprintf(1, '2-norm of VprimePerp = %e\n', norm(VprimePerp));
fprintf(1, 'gammaProj (gamma using P) = %e\n', gammaProj);
fprintf(1, 'muBarProj (muBar using P) = %e\n', muBarProj);
fprintf(1, 'condition number of N = %e\n', cond(N));
fprintf(1, 'Using gamma and muBar for orthonormal V, Vperp:\n')
fprintf(1, '2-norm of out-of-subspace error, epsilon = %e\n', epsilon);
fprintf(1, '2-norm of in-subspace error = %e\n', inSub2Norm);
fprintf(1, 'Bound on 2-norm of in-subspace error = %e\n', inSub2NormBound);
fprintf(1, 'Sup-norm of in-subspace error = %e\n', inSubSupNorm);
fprintf(1, 'Bound on Sup-norm of in-subspace error = %e\n', ...
    inSubSupNormBound);
fprintf(1, '2-norm of total error = %e\n', totErr2Norm);
fprintf(1, 'Bound on 2-norm of total error = %e\n', totErr2NormBound);
fprintf(1, '-----------------------------------------------------------\n')
```

```
end

function [inSubSupNormBound, inSub2NormBound, totErr2NormBound] = ...
    CalculateErrorBoundsThm4_1(epsilon, gamma, muBar, T, V, Vperp)
% Purpose: Calculate the error bounds given in Theorem 4.1.
% Inputs: epsilon = \varepsilon in Theorem 4.1, bound on function 2-norm of
%         out-of-subspace error, \|\mathbf{e}_{c}\|_{2}
%         gamma = \gamma in Theorem 4.1, bound on Lipschitz constant of
%         W'*A in directions corresponding to W_{\perp}, where \dot{y}(t)
%         = A*y(t)
%         muBar = \bar{\mu} in Theorem 4.1, bound on logarithmic norm of
%         W'*A*V, where \dot{y}(t) = A*y(t)
%         T = end time of integration
%         V = \mathbf{V} matrix in Theorem 4.1, basis for range of
%         projector
%         Vperp = \mathbf{V}_{\perp} matrix in Theorem 4.1, basis for
%         orthogonal complement of range of projector
% Outputs: inSubSupNormBound = bound on the function sup-norm of the
%          in-subspace error, \|\mathbf{e}_{i}\|_{\infty} in Theorem 4.1
%          inSub2NormBound = bound on the function 2-norm
%          totErr2NormBound =
% Assertion checks: None; need to make sure that V and Vperp have the same
% number of rows.

inSubSupNormBound = epsilon * gamma * ...
    sqrt( (exp(2 * muBar * T) - 1) / (2 * muBar) ) * ...
    norm(V) * norm(Vperp);

% Convenience variable used to hold intermediate result common to two later
% expressions
twoNormScalingFactor = gamma * norm(V) * norm(Vperp) *...
    sqrt( (exp(2 * muBar * T) - 1 - 2 * muBar * T) / (4 * muBar^2));

inSub2NormBound = epsilon * twoNormScalingFactor ;
totErr2NormBound = epsilon * (1 + twoNormScalingFactor );

end


function [X, Y, N, Xperp, Yperp, Nc] = ...
    CalcXQRYRep(rangeBasis, nullBasis)
% Purpose: Calculates the XQRY representation of a projection matrix and
% its complementary projection matrix, given a basis for the range and null
% space of the projection matrix. See G. W. (Pete)
% Stewart's "On the Numerical Analysis of Oblique Projectors", SIMAX, 2011
% for additional details
% Inputs: rangeBasis = basis for range space of projection matrix
%         nullBasis = basis for null space of projection matrix
% Outputs: X = orthonormal basis for range of projector
%          Y = orthonormal basis for range of transpose of projector
%          N = inv(Y'*X)
%          Xperp = orthonormal basis whose span is orthogonal to span(X)
%          Yperp = orthonormal basis whose span is orthogonal to span(Y)
%          Nc = inv(Xperp'*Yperp)
% Assertion checks: None. Should check for consistency of matrix
% dimensions.

n = size(rangeBasis, 1);
k = size(rangeBasis, 2);

% Calculate Petrov-Galerkin matrices and projector. Use G. W. (Pete)
% Stewart's "On the Numerical Analysis of Oblique Projectors", SIMAX, 2011
% to guide algorithms; in this case, the algorithms being used is
% equation (5.1).

% Q and R will be temporary variables used for the result of QR
% factorizations.

% First, use QR factorization to find orthogonal matrices whose columns
% span the desired subspaces. Use Stewart's nomenclature.
[Q,~] = qr(rangeBasis);
X = Q(:,1:k);
Xperp = Q(:, k+1:n);

% Yperp is a basis for the nullspace; Y is a basis for the range of P'.
[Q,~] = qr(nullBasis);
Yperp = Q(:,1:k);
Y = Q(:,k+1:n);

% Calculate intermediate matrices for Stewart's XQRY representation.
M = Y'*X;
[Q,R] = qr(M);
N = R\Q';

Mc = Xperp'*Yperp;
[Q,R] = qr(Mc);
Nc = R\Q';

end

function P = CalcExplicitProjectors(X, Y, N)
% Purpose: From their respective XQRY representations, calculate the
% projection matrix with range equal to span(X) and whose transpose has
% range span(Y).
% Inputs: X = orthonormal basis for range of projector
%         Y = orthonormal basis for range of transpose of projector
%         N = inv(Y'*X)
% Outputs: P = projection matrix
% Assertion checks: None; should check for consistency of matrix
% dimensions.

% Stewart warns against calculating projection matrices directly, due
% to possible numerical error, but it is needed for some performance
% metrics (like the norm of P).
P = X*N*Y';

end
```

```
function [V, W, Vperp, Wperp] = CalcNorm1WRep(X, Y, N, Xperp, Yperp, Nc)
% Purpose: From their respective XQRY representations, calculate the V and
% W corresponding to the projection matrix with range equal to span(X) and
% whose transpose has range span(Y), such that the columns of W are
% orthonormal (i.e., norm(W, 2) = 1). Also calculate Wperp and Vperp,
% corresponding to the complementary projection matrix, such that the
% columns of Wperp are orthonormal (i.e., norm(Wperp, 2) = 1).
% Inputs: X = orthonormal basis for range of projector
%         Y = orthonormal basis for range of transpose of projector
%         N = inv(Y’*X)
%         Xperp = orthonormal basis whose span is orthogonal to span(X)
%         Yperp = orthonormal basis whose span is orthogonal to span(Y)
%         Nc = inv(Xperp’*Yperp)
% Outputs: V = basis for range of projector
%          W = orthonormal matrix such that ker(W’) is nullspace of
%          projector; V*W’ = projector
%          Wperp = basis for range of complementary projector, orthonormal
%          matrix
%          Vperp = matrix such that ker(Vperp’) is nullspace of
%          complementary projector; Wperp*Vperp’ = complementary projector
% Assertion checks: None; should make sure that matrix dimensions are
% consistent, and that X, Y and N are consistent, etc.

% XY representation of projector calculated using algorithm suggested at
% bottom of p. 323 in G. W. (Pete) Stewart’s "On the Numerical Analysis of
% Oblique Projectors", SIMAX, 2011.

V = X*N;
W = Y;
Vperp = Xperp*Nc’;
Wperp = Yperp;

end

function [V, W, Vperp, Wperp] = CalcNorm1VRep(X, Y, N, Xperp, Yperp, Nc)
% Purpose: From their respective XQRY representations, calculate the V and
% W corresponding to the projection matrix with range equal to span(X) and
% whose transpose has range span(Y), such that the columns of V are
% orthonormal (i.e., norm(V, 2) = 1). Also calculate Wperp and Vperp,
% corresponding to the complementary projection matrix, such that the
% columns of Vperp are orthonormal (i.e., norm(Vperp, 2) = 1).
% Inputs: X = orthonormal basis for range of projector
%         Y = orthonormal basis for range of transpose of projector
%         N = inv(Y’*X)
%         Xperp = orthonormal basis whose span is orthogonal to span(X)
%         Yperp = orthonormal basis whose span is orthogonal to span(Y)
%         Nc = inv(Xperp’*Yperp)
% Outputs: V = orthonormal matrix, basis for range of projector
%          W = matrix such that ker(W’) is nullspace of
%          projector; V*W’ = projector
%          Wperp = basis for range of complementary projector
%          Vperp = orthonormal matrix such that ker(Vperp’) is nullspace of
%          complementary projector; Wperp*Vperp’ = complementary projector
% Assertion checks: None; should make sure that matrix dimensions are
% consistent, and that X, Y and N are consistent, etc.

% XY representation of projector calculated using algorithm suggested at
% bottom of p. 323 in G. W. (Pete) Stewart's "On the Numerical Analysis of
% Oblique Projectors", SIMAX, 2011.

V = X;
W = (N*Y')';
Wperp = Yperp*Nc;
Vperp = Xperp';

end

function [gamma, muBar] = CalcThm4_1Consts(coeffMatrix, V, W, Wperp)
% Purpose: Calculate the constants that determine the error bounds in
% Theorem 4.1.
% Inputs: coeffMatrix = A matrix in \dot{y}(t) = A*y(t)
%         V = basis for range of projector
%         W = matrix such that ker(W') is the null space of the projector;
%         V*W' = projector
%         Wperp = basis for range of complementary projector
% Outputs: gamma = \gamma in Theorem 4.1, bound on Lipschitz constant of
%         W'*A in directions corresponding to W_{\perp}, where \dot{y}(t)
%         = A*y(t)
%         muBar = \bar{\mu} in Theorem 4.1, bound on logarithmic norm of
%         W'*A*V, where \dot{y}(t) = A*y(t)
% Assertion checks: None; should be consistency of dimensions of V, W,
% Wperp.

gamma = norm(W'*coeffMatrix*Wperp);
muBar = max(real(eig(W'*coeffMatrix*V + (W'*coeffMatrix*V)')))/2;

end

function [product] = ProjVecProdStewart(X, Y, v)
% Purpose: Calculate projector-vector product from an XQRY representation
% of a projector using algorithm (5.2) in G. W. (Pete)
% Stewart's "On the Numerical Analysis of Oblique Projectors", SIMAX, 2011.
% Inputs: X = orthonormal basis for range of projector
%         Y = orthonormal basis for range of transpose of projector
%         v = vector to be projected
% Outputs: product = P*v = projected vector
% Assertion checks: None.

[Q,R] = qr(Y'*X);
c1 = Y'*v;
c2 = Q'*c1;
c3 = R\c2;
product = X*c3;

end

function [gamma, muBar] = CalcThm4_4Consts(coeffMatrix, X, Y, Wperp)
% Purpose: Calculate the constants that determine the error bounds in
% Theorem 4.4.
% Inputs: coeffMatrix = A matrix in \dot{y}(t) = A*y(t)
%         X = orthonormal basis for range of projector
%         Y = orthonormal basis for range of transpose of projector
%         Wperp = orthonormal basis for range of complementary projector
% Outputs: gamma = \gamma in Theorem 4.4, bound on Lipschitz constant of
%         P*A in directions corresponding to W_{\perp}, where \dot{y}(t)
%         = A*y(t)
%         muBar = \bar{\mu} in Theorem 4.1, bound on logarithmic norm of
%         P*A, where \dot{y}(t) = A*y(t)
% Assertion checks: None; should check consistency of dimensions of
% coeffMatrix, X, Y, Wperp; check orthogonality of Wperp (should have
% 2-norm of 1).

% Calculate product projCoeffMatrix = P*coeffMatrix
projCoeffMatrix = ProjVecProdStewart(X, Y, coeffMatrix);

gamma = norm(projCoeffMatrix*Wperp);
muBar = max(real(eig(projCoeffMatrix + (projCoeffMatrix)')))/2;

end

function [epsilon, inSubSupNorm, inSub2Norm, totErr2Norm] = ...
    CalcErrors(T, initCond, coeffMatrix, X, Y, Xperp, Yperp)
% Purpose: Calculate solutions to the full and reduced models, plot these
% solutions, and then calculate various errors in the reduced model.
% Inputs: T = end time for numerical integration
%         initCond = initial condition for numerical integration
%         coeffMatrix = coefficient (A) matrix for linear system,
%         \dot{y}(t) = A*y(t), n by n
%         X = orthonormal basis for range of projector
%         Y = orthonormal basis for range of transpose of projector
%         Xperp = orthonormal basis whose span is orthogonal to span(X)
%         Yperp = orthonormal basis whose span is orthogonal to span(Y)
% Outputs: epsilon = function 2-norm of component of error in reduced model
%          solution in null space of projection matrix.
%          inSubSupNorm = function sup-norm of component of error in
%          reduced model solution in range of projection matrix
%          inSub2Norm = function 2-norm of component of error in reduced
%          model solution in range of projection matrix
%          totErr2Norm = function 2-norm of total error in reduced model
%          solution
% Assertion checks: None; check consistency of matrix dimensions.

n = size(coeffMatrix, 1);

% Now, to simulate system, emulating Rathinam and Petzold, SINUM, 2004.
tSpan = [0,T];

% Calculate product projCoeffMatrix = P*coeffMatrix
projCoeffMatrix = ProjVecProdStewart(X, Y, coeffMatrix);

% Note that error tolerances set very tightly to decrease numerical error
% due to integration. Use 4th-order Runge-Kutta integration because system
% is not stiff.
options = odeset(’RelTol’,1e-13,’AbsTol’,ones(1,n)*1e-25);
[tPts, fullSoln] = ode45(@linSys, tSpan, initCond, options, coeffMatrix);
[~,redSoln] = ode45(@linSys, tPts, initCond, options, projCoeffMatrix);

% Calculate error in reduced model solution
errSoln = redSoln - fullSoln;

% Calculate product inSubErrSoln = P*errSoln’
inSubErrSoln = ProjVecProdStewart(X, Y, errSoln’);

% Calculate product outSubErrSoln = (I-P)*errSoln’
outSubErrSoln = ProjVecProdStewart(Yperp, Xperp, errSoln’);

% Trapezoidal rule approximation to epsilon, which seems to work well.
epsilon = sqrt(trapz(tPts, sum(outSubErrSoln.^2,1)));

% Calculate sup-norm of in-subspace error.
inSubSupNorm = max(max(abs(inSubErrSoln)));
inSub2Norm = sqrt(trapz(tPts, sum(inSubErrSoln.^2,1)));
totErr2Norm = sqrt(trapz(tPts, sum((errSoln’).^2,1)));

% Plots comparing full and reduced model solutions
figure;
plot(tPts, fullSoln(:,1), ’r-’);
hold on;
plot(tPts, fullSoln(:,2), ’b-’);
plot(tPts, fullSoln(:,3), ’k-’);
plot(tPts, redSoln(:,1), ’r--’);
plot(tPts, redSoln(:,2), ’b--’);
plot(tPts, redSoln(:,3), ’k--’);
title(’Comparison of full and reduced model solutions’);
xlabel(’Time (t) [a.u.]’);
ylabel(’State variable (x_j) [a.u.]’);
legend(’full, 1’, ’full, 2’, ’full, 3’, ’reduced, 1’, ’reduced, 2’, ...
    ’reduced, 3’, ’Location’, ’Best’);

figure;
plot(tPts, fullSoln(:,4), ’r-’);
hold on;
plot(tPts, fullSoln(:,5), ’b-’);
plot(tPts, fullSoln(:,6), ’k-’);
plot(tPts, redSoln(:,4), ’r--’);
plot(tPts, redSoln(:,5), ’b--’);
plot(tPts, redSoln(:,6), ’k--’);
title(’Comparison of full and reduced model solutions’);
xlabel(’Time (t) [a.u.]’);
ylabel(’State variable (x_j) [a.u.]’);
legend(’full, 4’, ’full, 5’, ’full, 6’, ’reduced, 4’, ’reduced, 5’, ...
    ’reduced, 6’, ’Location’, ’Best’);

% Plots of errors
figure;
plot(tPts, errSoln(:,1), ’r-’);
```

```
hold on;
plot(tPts, errSoln(:,2), 'b-');
plot(tPts, errSoln(:,3), 'k-');
plot(tPts, inSubErrSoln(1,:), 'r--');
plot(tPts, inSubErrSoln(2,:), 'b--');
plot(tPts, inSubErrSoln(3,:), 'k--');
title('Error in reduced model solution');
xlabel('Time (t) [a.u.]');
ylabel('Error in state variable (e_j) [a.u.]');
legend('j=1', 'j=2', 'j=3', 'j=1, in-subspace',...
    'j=2, in-subspace', 'j=3, in-subspace','Location', 'Best');

figure;
plot(tPts, errSoln(:,4), 'r-');
hold on;
plot(tPts, errSoln(:,5), 'b-');
plot(tPts, errSoln(:,6), 'k-');
plot(tPts, inSubErrSoln(4,:), 'r--');
plot(tPts, inSubErrSoln(5,:), 'b--');
plot(tPts, inSubErrSoln(6,:), 'k--');
title('Error in reduced model solution');
xlabel('Time (t) [a.u.]');
ylabel('Error in state variable (e_j) [a.u.]');
legend('j=4', 'j=5', 'j=6', 'j=4, in-subspace',...
    'j=5, in-subspace', 'j=6, in-subspace', 'Location', 'Best');

end
```

B.2 Python Implementation

The Python 2.7.3 [209] implementation requires the installation of NumPy 1.6.2 (or later) [152], SciPy 0.10.1 (or later) [93], and Matplotlib 1.0.0 (or later) [90]. An attempt was made to keep the number of dependencies to a minimum. It is likely that the Python code below will work with Python 2.6 (or later).

```
#!/usr/bin/env python

import numpy
import scipy.integrate
import scipy.linalg
import matplotlib.pyplot
import math
import copy

def lin_sys(t, y, A):
    """
    Purpose:
    Auxiliary function encoding a linear ODE system for numerical integration.
    Arguments:
    t (float): time
    y (1-D numpy.ndarray): states
    A (2-D numpy.ndarray): square coefficient array, where A.shape[1] == len(y)

    Returns:
    f (1-D numpy.ndarray): right-hand side of ODE

    """

    # If y were a column vector, we would calculate the right-hand side as
    # A * y. Instead, y is a row vector (by default in numpy), so we calculate
    # y * A^{T} (as if we took the previous case, where y is a column vector,
    # and used (A * y)^{T} = y^{T} * A^{T}.
    f = numpy.dot(A, y)
    return f

def calc_err_bounds_thm_4_1(epsilon, gamma, mu_bar, T, V, V_perp):
    """
    Purpose:
    Calculate the error bounds given in Theorem 4.1.

    Arguments:
    epsilon (float): \varepsilon in Theorem 4.1, bound on function 2-norm of
        out-of-subspace error, \|\mathbf{e}_{c}\|_{2}
    gamma (float): \gamma in Theorem 4.1, bound on Lipschitz constant of
        W^{T} * A in directions corresponding to W_{\perp}, where \dot{y}(t)
        = A * y(t)
    mu_bar (float): \bar{\mu} in Theorem 4.1, bound on logarithmic norm of
        W^{T} * A * V, where \dot{y}(t) = A*y(t)
    T (float): end time of integration
    V (2-D numpy.ndarray of floats): \mathbf{V} matrix in Theorem 4.1, basis
        for range of projector
    V_perp (2-D numpy.ndarray of floats): \mathbf{V}_{\perp} matrix in
        Theorem 4.1, basis for orthogonal complement of range of projector

    Returns:
    in_sub_sup_norm_bound (float): bound on the function sup-norm of the in-
        subspace error, \|\mathbf{e}_{i}\|_{\infty} in Theorem 4.1
    in_sub_2_norm_bound (float): bound on the function 2-norm of the in-
        subspace error, \|\mathbf{e}_{i}\|_{2}
    tot_err_2_norm_bound (flooat): bound on the function 2-norm of the total
        error, \|\mathbf{e}\|_{2}

    """

    in_sub_sup_norm_bound = (epsilon * gamma *
        math.sqrt((math.exp(2 * mu_bar * T) - 1) / (2 * mu_bar)) *
        numpy.linalg.norm(V, 2) * numpy.linalg.norm(V_perp, 2))

    # Convenience variable used to hold intermediate result common to two
    # later expressions
    two_norm_scaling_factor = (gamma * numpy.linalg.norm(V, 2) *
        numpy.linalg.norm(V_perp, 2) * math.sqrt(
        (math.exp(2 * mu_bar * T) - 1 - 2 * mu_bar * T) / (4 * mu_bar ** 2)))

    in_sub_2_norm_bound = epsilon * two_norm_scaling_factor
    tot_err_2_norm_bound = epsilon * (1 + two_norm_scaling_factor)

    return (in_sub_sup_norm_bound, in_sub_2_norm_bound, tot_err_2_norm_bound)

def calc_XQRY_rep(range_basis, null_basis):
    """
    Purpose:
    Calculates the XQRY representation of a projection matrix and its
    complementary projection matrix, given a basis for the range and null
    space of the projection matrix. See G. W. (Pete) Stewart's "On the
    Numerical Analysis of Oblique Projectors", SIMAX, 2011 for additional
    details.

    Arguments:
    range_basis (2-D numpy.ndarray of floats): basis for range space of
        projection matrix
    null_basis (2-D numpy.ndarray of floats): basis for null space of
        projection matrix

    Returns:
    X (2-D numpy.ndarray of floats): orthonormal basis for range of projector
    Y (2-D numpy.ndarray of floats): orthonormal basis for range of transpose
        of projector
    N (2-D numpy.ndarray of floats): inv(Y^{T} * X)
    X_perp (2-D numpy.ndarray of floats): orthonormal basis whose span is
        orthogonal to span(X)
    Y_perp (2-D numpy.ndarray of floats): orthonormal basis whose span is
        orthogonal to span(Y)
    N_c (2-D numpy.ndarray of floats): inv(X_perp^{T} * Y_perp)

    Assertion checks:
    None. Should check for consistency of matrix dimensions.

    """

    (n, k) = range_basis.shape

    # Calculate Petrov-Galerkin matrices and projector. Use G. W. (Pete)
    # Stewart's "On the Numerical Analysis of Oblique Projectors", SIMAX, 2011
    # to guide algorithms; in this case, the algorithms being used is (5.1).

    # Q and R will be temporary variables used for the results of QR
    # factorizations.

    # First, use QR factorization to find orthogonal matrices whose columns
    # span the desired subspaces. Use Stewart's nomenclature.
    (Q, _) = scipy.linalg.qr(range_basis)
    X = Q[:, 0:k]
    X_perp = Q[:, k:]
    # Y_perp is a basis for the null space; Y is a basis for the range of P^{T}
    (Q, _) = scipy.linalg.qr(null_basis)
    Y_perp = Q[:, 0:k]
    Y = Q[:, k:]

    # Calculate intermediate matrices for Stewart's XQRY representation.
    M = numpy.dot(Y.transpose(), X)
    (Q, R) = scipy.linalg.qr(M)
    N = numpy.linalg.solve(R, Q.transpose())

    M_c = numpy.dot(X_perp.transpose(), Y_perp)
    (Q, R) = scipy.linalg.qr(M_c)
    N_c = numpy.linalg.solve(R, Q.transpose())

    return (X, Y, N, X_perp, Y_perp, N_c)

def calc_explicit_projector(X, Y, N):
    """
    Purpose:
    From their respective XQRY representations, calculate the projection
    matrix with range equal to span(X) and whose transpose has range span(Y).

    Arguments:
    X (2-D numpy.ndarray of floats): orthonormal basis for range of projector
    Y (2-D numpy.ndarray of floats): orthonormal basis for range of transpose
        of projector
    N (2-D numpy.ndarray of floats): inv(Y^{T} * X)

    Returns:
    P (2-D numpy.ndarray of floats): projection matrix

    Assertion checks:
    None. Should check for consistency of matrix dimensions.

    """

    # Stewart warns against calculating projection matrices directly, due to
    # possible numerical error, but it is needed for some performance metrics,
    # (like the norm of P).
    P = numpy.dot(X, numpy.dot(N, Y.transpose()))

    return P

def calc_norm_1_W_rep(X, Y, N, X_perp, Y_perp, N_c):
    """
    Purpose:
    From their respective XQRY representations, calculate the V and
    W corresponding to the projection matrix with range equal to
    span(X) and whose transpose has range span(Y), such that the columns
    of W are orthonormal (i.e., numpy.linalg.norm(W, 2) = 1). Also calculate
    W_perp and V_perp, corresponding to the complementary projection matrix,
    such that the columns of W_perp are orthonormal (i.e.,
    numpy.linalg.norm(W_perp, 2) = 1).
    Arguments:
    X (2-D numpy.ndarray of floats): orthonormal basis for range of projector
    Y (2-D numpy.ndarray of floats): orthonormal basis for range of transpose
        of projector
    N (2-D numpy.ndarray of floats): inv(Y^{T} * X)
    X_perp (2-D numpy.ndarray of floats): orthonormal basis whose span is
        orthogonal to span(X)
    Y_perp (2-D numpy.ndarray of floats): orthonormal basis whose span is
        orthogonal to span(Y)
    N_c (2-D numpy.ndarray of floats): inv(X_perp^{T} * Y_perp)

    Returns:
    V (2-D numpy.ndarray of floats): basis for range of projector
    W (2-D numpy.ndarray of floats): orthonormal matrix such that ker(W^{T})
        is null space of projector; V * W^{T} = projector
    W_perp (2-D numpy.ndarray of floats): basis for rnage of complementary
        projector, orthonormal matrix
    V_perp (2-D numpy.ndarray of floats): matrix such that ker(V_perp^{T}) is
        null space of complemntary projector; W_perp * V_perp^{T} =
        complementary projector

    Assertion checks:
    None; should make sure that matrix dimensions are
    consistent, and that X, Y, and N are consistent, etc.

    """

    # XY representation of projector calculated using algorithm suggested at
    # bottom of p. 323 in G. W. (Pete) Stewart's "On the Numerical Analysis of
    # Oblique Projectors", SIMAX, 2011.

    V = numpy.dot(X, N)
    W = Y
    V_perp = numpy.dot(X_perp, N_c.transpose())
    W_perp = Y_perp

    return (V, W, V_perp, W_perp)

def calc_norm_1_V_rep(X, Y, N, X_perp, Y_perp, N_c):
    """
    Purpose:
    From their respective XQRY representations, calculate the V and W
    corresponding to the projection matrix with range equal to span(X) and
    whose transpose has range span(Y), such that the columns of V are
    orthonormal (i.e., numpy.linalg.norm(V, 2) = 1). Also calculate W_perp and
    V_perp, corresponding to the complementary projection matrix, such that
    the columns of V_perp are orthonormal (i.e.,
    numpy.linalg.norm(V_perp, 2) = 1).

    Arguments:
    X (2-D numpy.ndarray of floats): orthonormal basis for range of projector
    Y (2-D numpy.ndarray of floats): orthonormal basis for range of transpose
        of projector
    N (2-D numpy.ndarray of floats): inv(Y^{T} * X)
    X_perp (2-D numpy.ndarray of floats): orthonormal basis whose span is
        orthogonal to span(X)
    Y_perp (2-D numpy.ndarray of floats): orthonormal basis whose span is
        orthogonal to span(Y)
    N_c (2-D numpy.ndarray of floats): inv(X_perp^{T} * Y_perp)

    Returns:
    V (2-D numpy.ndarray of floats): orthonormal matrix, basis for range of
        projector
    W (2-D numpy.ndarray of floats): matrix such that ker(W^{T}) is null
        space of projector; V * W^{T} = projector
    W_perp (2-D numpy.ndarray of floats): basis for range of complementary
        projector
    V_perp (2-D numpy.ndarray of floats): orthonormal matrix such that
        ker(V_perp^{T}) is null space of complementary projector;
        W_perp * V_perp^{T} = projector

    Assertion checks:
    None; should make sure that matrix dimensions are consistent, and that
    X, Y, and N are consistent, etc.

    """

    # XY representation of projector calculated using algorithm suggested at
    # bottom of p. 323 in G. W. (Pete) Stewart's "On the Numerical Analysis of
    # Oblique Projectors", SIMAX, 2011.

    V = X
    W = numpy.dot(N, Y.transpose()).transpose()
    W_perp = numpy.dot(Y_perp, N_c)
    V_perp = X_perp.transpose()

    return (V, W, V_perp, W_perp)

def calc_thm_4_1_consts(coeff_matrix, V, W, W_perp):
    """
    Purpose:
    Calculate the constants that determine the error bounds in Theorem 4.1.

    Arguments:
    coeff_matrix (2-D numpy.ndarray of floats): A matrix in
        \dot{y}(t) = A*y(t)
    V (2-D numpy.ndarray of floats): basis for range of projector
    W (2-D numpy.ndarray of floats): matrix such that ker(W^{T}) is the null
        space of the projector; V * W^{T} = projector
    W_perp (2-D numpy.ndarray of floats): basis for range of complementary
        projector

    Returns:
    gamma (float): \gamma in Theorem 4.1, bound on Lipschitz constant of
        W^{T} * A in directions corresponding to W_{\perp}, where \dot{y}(t)
        = A * y(t)
    mu_bar (float): \bar{\mu} in Theorem 4.1, bound on logarithmic norm of
        W^{T} * A * V, where \dot{y}(t) = A * y(t)
    Assertion checks:
    None; should be consistency of dimensions of V, W, W_perp.

    """

    gamma = numpy.linalg.norm(numpy.dot(W.transpose(),
                       numpy.dot(coeff_matrix, W_perp)), 2)

    matrix = numpy.dot(W.transpose(), numpy.dot(coeff_matrix,V))
    mu_bar = numpy.max(numpy.real(numpy.linalg.eigvals(
        matrix + matrix.transpose()))) / 2

    return gamma, mu_bar

def calc_thm_4_4_consts(coeff_matrix, X, Y, W_perp):
    """
    Purpose: Calculate the constants that determine the error bounds in
    Theorem 4.4.

    Arguments:
    coeff_matrix (2-D numpy.ndarray of floats): A matrix in
        \dot{y}(t) = A*y(t)
    X (2-D numpy.ndarray of floats): orthonormal basis for range of projector
    Y (2-D numpy.ndarray of floats): orthonormal basis for range of transpose
        of projector
    W_perp (2-D numpy.ndarray of floats): basis for range of complementary
        projector

    Returns:
    gamma (float): \gamma in Theorem 4.4, bound on Lipschitz constant of
        P * A in directions corresponding to W_{\perp}, where \dot{y}(t)
        = A * y(t)
    mu_bar (float): \bar{\mu} in Theorem 4.1, bound on logarithmic norm of
        P * A, where \dot{y}(t) = A * y(t)

    Assertion checks:
    None; should check consistency of dimensions of coeff_matrix, X, Y,
    W_perp; check orthogonality of W_perp (should have 2-norm of 1).

    """

    proj_coeff_matrix = proj_vec_prod_stewart(X, Y, coeff_matrix)

    gamma = numpy.linalg.norm(numpy.dot(proj_coeff_matrix, W_perp))
    mu_bar = numpy.max(numpy.real(numpy.linalg.eigvals(
        proj_coeff_matrix + proj_coeff_matrix.transpose()))) / 2

    return gamma, mu_bar


def proj_vec_prod_stewart(X, Y, v):
    """
    Purpose:
```

```
    Calculate projector-vector product from an XQRY representation of a
    projector using algorithm (5.2) in G. W. (Pete) Stewart's "On the
    Numerical Analysis of Oblique Projectors", SIMAX, 2011.

    Arguments:
    X (2-D numpy.ndarray of floats): orthonormal basis for range of projector
    Y (2-D numpy.ndarray of floats): orthonormal basis for range of transpose
        of projector
    v (1-D numpy.ndarray of floats): vector to be projected

    Returns:
    product (1-D numpy.ndarray of floats): P*v = projected vector

    Assertion checks: None.

    """

    (Q, R) = scipy.linalg.qr(numpy.dot(Y.transpose(), X))
    c1 = numpy.dot(Y.transpose(), v)
    c2 = numpy.dot(Q.transpose(), c1)
    c3 = numpy.linalg.solve(R, c2)
    product = numpy.dot(X, c3)

    return product

def calc_errors(T, init_cond, coeff_matrix, X, Y, X_perp, Y_perp):
    """
    Purpose:
    Calculate solutions to the full and reduced models, plot these
    solutions, and then calculate various errors in the reduced model.

    Arguments:
    T (float): end time for numerical integration
    init_cond (1-D numpy.ndarray of floats): initial condition for
        numerical integration
    coeff_matrix (2-D numpy.ndarray of floats): coefficient (A) matrix for
        linear system, \dot{y}(t) = A * y(t), n by n
    X (2-D numpy.ndarray of floats): orthonormal basis for range of projector
    Y (2-D numpy.ndarray of floats): orthonormal basis for range of transpose
        of projector
    X_perp (2-D numpy.ndarray of floats): orthonormal basis whose span is
        orthogonal to span(X)
    Y_perp (2-D numpy.ndarray of floats): orthonormal basis whose span is
        orthogonal to span(Y)

    Returns:
    epsilon (float): function 2-norm of component of error in reduced model
        solution in null space of projection matrix
    in_sub_sup_norm (float): function sup-norm of component of error in
        reduced model solution in range of projection matrix
    in_sub_2_norm (float): function 2-norm of component of error in reduced
        model solution in range of projection matrix
    tot_err_2_norm (float): function 2-norm of total error in reduced model
        solution
```

```
soln_fig_1, soln_fig_2 (matplotlib.pyplot.Figure): plots of full and
    reduced model solutions
err_fig_1, err_fig_2 (matplotlib.pyplot.Figure): plots of in-subspace and
    total error

Assertion checks:
None; check consistency of matrix dimensions.

"""

# Simulate system, emulating Rathinam and Petzold, SINUM, 2004.
t_begin = 0
t_end = T
n_time_pts = 1000
t = numpy.linspace(t_begin, t_end, n_time_pts)

# Calculate product proj_coeff_matrix = P * coeff_matrix
proj_coeff_matrix = proj_vec_prod_stewart(X, Y, coeff_matrix)

# Set up numerical integrators. Note that the error tolerances are set
# very tightly to decrease numerical error due to integration. Use 7th-order
# explicit Runge-Kutta integration because systems are not stiff.
full_sys = scipy.integrate.ode(lin_sys)
full_sys.set_integrator('dop853', atol=1e-25, rtol=1e-13, nsteps=10000000)
full_sys.set_initial_value(init_cond, 0)
full_sys.set_f_params(coeff_matrix)

red_sys = scipy.integrate.ode(lin_sys)
red_sys.set_integrator('dop853', atol=1e-25, rtol=1e-3, nsteps=10000000)
red_sys.set_initial_value(init_cond, 0)
red_sys.set_f_params(proj_coeff_matrix)

# Run integration loops; use numpy.vstack to avoid the need for copying
# state of integrators.
full_soln = init_cond
for point in t[1:]:
    if not full_sys.successful(): break
    full_sys.integrate(point)
    full_soln = numpy.vstack((full_soln, full_sys.y))

red_soln = init_cond
for point in t[1:]:
    if not red_sys.successful(): break
    red_sys.integrate(point)
    red_soln = numpy.vstack((red_soln, red_sys.y))

# Calculate error in reduced model solution
err_soln = red_soln - full_soln

# Calculate product in_sub_err_soln = P * err_soln^{T}
in_sub_err_soln = proj_vec_prod_stewart(X, Y, err_soln.transpose())

# Calculate product out_sub_err_soln = (I - P) * err_soln^{T}
out_sub_err_soln = proj_vec_prod_stewart(Y_perp, X_perp,
                                        err_soln.transpose())

# Trapezoidal rule approximation to epsilon, which seems to work well.
epsilon = numpy.sqrt(numpy.trapz(
    numpy.sum(out_sub_err_soln ** 2, axis=0), t))

# Calculate remaining norms
in_sub_sup_norm = numpy.max(numpy.abs(in_sub_err_soln))
in_sub_2_norm = numpy.sqrt(numpy.trapz(
    numpy.sum(in_sub_err_soln ** 2, axis=0), t))
tot_err_2_norm = numpy.sqrt(numpy.trapz(
    numpy.sum(err_soln.transpose() ** 2, axis=0), t))

# Plots comparing full and reduced model solutions
soln_fig_1 = matplotlib.pyplot.figure()
matplotlib.pyplot.plot(t, full_soln[:, 0], 'r-')
matplotlib.pyplot.plot(t, full_soln[:, 1], 'b-')
matplotlib.pyplot.plot(t, full_soln[:, 2], 'k-')
matplotlib.pyplot.plot(t, red_soln[:, 0], 'r--')
matplotlib.pyplot.plot(t, red_soln[:, 1], 'b--')
matplotlib.pyplot.plot(t, red_soln[:, 2], 'k--')
matplotlib.pyplot.title('Comparison of full and reduced model solutions')
matplotlib.pyplot.xlabel('Time (t) [a.u.]')
matplotlib.pyplot.ylabel('State variable (x_j) [a.u.]')
matplotlib.pyplot.legend( ('full, 1', 'full, 2', 'full, 3',
    'reduced, 1', 'reduced, 2', 'reduced, 3'), loc='best')

soln_fig_2 = matplotlib.pyplot.figure()
matplotlib.pyplot.plot(t, full_soln[:, 3], 'r-')
matplotlib.pyplot.plot(t, full_soln[:, 4], 'b-')
matplotlib.pyplot.plot(t, full_soln[:, 5], 'k-')
matplotlib.pyplot.plot(t, red_soln[:, 3], 'r--')
matplotlib.pyplot.plot(t, red_soln[:, 4], 'b--')
matplotlib.pyplot.plot(t, red_soln[:, 5], 'k--')
matplotlib.pyplot.title('Comparison of full and reduced model solutions')
matplotlib.pyplot.xlabel('Time (t) [a.u.]')
matplotlib.pyplot.ylabel('State variable (x_j) [a.u.]')
matplotlib.pyplot.legend( ('full, 4', 'full, 5', 'full, 6',
    'reduced, 4', 'reduced, 5', 'reduced, 6'), loc='best')

err_fig_1 = matplotlib.pyplot.figure()
matplotlib.pyplot.plot(t, err_soln[:, 0], 'r-')
matplotlib.pyplot.plot(t, err_soln[:, 1], 'b-')
matplotlib.pyplot.plot(t, err_soln[:, 2], 'k-')
matplotlib.pyplot.plot(t, in_sub_err_soln[0, :], 'r--')
matplotlib.pyplot.plot(t, in_sub_err_soln[1, :], 'b--')
matplotlib.pyplot.plot(t, in_sub_err_soln[2, :], 'k--')
matplotlib.pyplot.title('Error in reduced model solution')
matplotlib.pyplot.xlabel('Time (t) [a.u.]')
matplotlib.pyplot.ylabel('Error in state variable (e_j) [a.u.]')
matplotlib.pyplot.legend( ('total, 1', 'total, 2', 'total, 3',
    'in-subspace, 1', 'in-subspace, 2', 'in-subspace, 3'), loc='best')

err_fig_2 = matplotlib.pyplot.figure()
```

```
    matplotlib.pyplot.plot(t, err_soln[:, 3], 'r-')
    matplotlib.pyplot.plot(t, err_soln[:, 4], 'b-')
    matplotlib.pyplot.plot(t, err_soln[:, 5], 'k-')
    matplotlib.pyplot.plot(t, in_sub_err_soln[3, :], 'r--')
    matplotlib.pyplot.plot(t, in_sub_err_soln[4, :], 'b--')
    matplotlib.pyplot.plot(t, in_sub_err_soln[5, :], 'k--')
    matplotlib.pyplot.title('Error in reduced model solution')
    matplotlib.pyplot.xlabel('Time (t) [a.u.]')
    matplotlib.pyplot.ylabel('Error in state variable (e_j) [a.u.]')
    matplotlib.pyplot.legend( ('total, 4', 'total, 5', 'total, 6',
        'in-subspace, 4', 'in-subspace, 5', 'in-subspace, 6'), loc='best')

    return (epsilon, in_sub_sup_norm, in_sub_2_norm, tot_err_2_norm,
            soln_fig_1, soln_fig_2, err_fig_1, err_fig_2)

def comparison_of_models(coeff_matrix, range_basis, null_basis, init_cond):
    """
    Purpose:
    Integrates the linear system \dot{\mathbf{x}} = coeff_matrix * \mathbf{x},
    and also integrates the projected system \dot{\hat{\mathbf{x}}} =
    \mathbf{P} * coeff_matrix * \hat{\mathbf{x}}. \mathbf{P} is a projection
    matrix that has range equal to the space spanned by the columns of
    range_basis, and null space equal to the space spanned by null_basis.

    Arguments:
    coeff_matrix (2-D numpy.ndarray of floats): square coefficient matrix of
        linear ODE system; coeff_matrix.shape[0] == len(init_cond)
    range_basis (2-D numpy.ndarray of floats): columns determine range space
        of projection matrix; coeff_matrix.shape[0] == range_basis.shape[0]
    null_basis (2-D numpy.ndarray of floats): columns determine null space of
        projection matrix; coeff_matrix.shape[0] == null_basis.shape[0], and
        coeff.matrix.shape[1] == (null_basis.shape[1] + range_basis.shape[1])
    init_cond (1-D numpy.ndarray of floats): initial condition for integration

    Returns:
    soln_fig_1, soln_fig_2 (matplotlib.pyplot.Figure): plots of full and
        reduced model solutions
    err_fig_1, err_fig_2 (matplotlib.pyplot.Figure): plots of in-subspace and
        total error

    Assertion checks:
    None; doesn't check for consistency, although error checks could be added
    later. NOTE: Many plots assume that n = 6 (# of variables); these can be
    generalized.

    """

    # Get size of problem and number of range basis vectors
    (n, k) = range_basis.shape

    # Calculate Petrov-Galerkin matrices and projector. Use G. W. (Pete)
    # Stewart's "On the Numerical Analysis of Oblique Projectors", SIMAX,
    # Vol. 32, No. 1, pp. 309-348 (2011), to guide algorithms.
    (X, Y, N, X_perp, Y_perp, N_c) = calc_XQRY_rep(range_basis, null_basis)
```

```
# Stewart warns against calculating projection matrices directly, due to
# possible numerical error, but here, it is needed for some performance
# metrics (the norm of P).
P = calc_explicit_projector(X, Y, N)

# Calculate V, W, V_perp, W_perp where numpy.linalg.norm(W, 2) =
# numpy.linalg.norm(W_perp, 2) = 1.
(V, W, V_perp, W_perp) = calc_norm_1_W_rep(X, Y, N, X_perp, Y_perp, N_c)

# Calculate V, W, V_perp, W_perp where numpy.linalg.norm(V, 2) =
# numpy.linalg.norm(V_perp, 2) = 1.
(V_prime,
 W_prime,
 V_prime_perp,
 W_prime_perp) = calc_norm_1_V_rep(X, Y, N, X_perp, Y_perp, N_c)

# Calculate constant in Theorem 4.1 and Corollary 4.4:
# Original choice of V, W, V_perp, W_perp
(gamma, mu_bar) = calc_thm_4_1_consts(coeff_matrix, V, W, W_perp)

# Constants after change of basis for V, W, V_perp, and W_perp
(gamma_prime,
 mu_bar_prime) = calc_thm_4_1_consts(coeff_matrix, V_prime,
                                     W_prime, W_prime_perp)

# Corollary 4.4: Constants using projection matrix instead
(gamma_proj,
 mu_bar_proj) = calc_thm_4_4_consts(coeff_matrix, X, Y, W_perp)

# Now, to simulate full and reduced system, emulating Rathinam and
# Petzold, "A New Look at Proper Orthogonal Decomposition", SINUM,
# Vol. 41, No. 5, pp. 1893-1925 (2004).
T = 5
(epsilon, in_sub_sup_norm, in_sub_2_norm, tot_err_2_norm,
 soln_fig_1, soln_fig_2,
 err_fig_1,
 err_fig_2) = calc_errors(T, init_cond, coeff_matrix,
                          X, Y, X_perp, Y_perp)

# Calculate error bounds
(in_sub_sup_norm_bound,
 in_sub_2_norm_bound,
 tot_err_2_norm_bound) = calc_err_bounds_thm_4_1(epsilon, gamma,
                                                  mu_bar, T, V, V_perp)

# Results.
print '------------------------------------------------------------'
print 'Size of matrix A (n by n), n = {}'.format(n)
print 'Size of reduced order model, k = {}'.format(k)
print 'gamma = {}'.format(gamma)
print 'mu_bar = {}'.format(mu_bar)
print '2-norm of P = {}'.format(numpy.linalg.norm(P, 2))
print '2-norm of V = {}'.format(numpy.linalg.norm(V, 2))
```

```
    print '2-norm of W = {}'.format(numpy.linalg.norm(W, 2))
    print 'gamma_prime (gamma under change of basis) = {}'.format(gamma_prime)
    print 'mu_bar_prime (mu_bar under change of basis) = {}'.format(
        mu_bar_prime)
    print '2-norm of V_prime = {}'.format(numpy.linalg.norm(V_prime, 2))
    print '2-norm of W_prime = {}'.format(numpy.linalg.norm(W_prime, 2))
    print '2-norm of V_prime_perp = {}'.format(numpy.linalg.norm(
        V_prime_perp, 2))
    print 'gamma_proj (gamma using P) = {}'.format(gamma_proj)
    print 'mu_bar_proj (mu_bar using P) = {}'.format(mu_bar_proj)
    print 'Condition number of N = {}'.format(numpy.linalg.cond(N))
    print 'Using gamma and mu_bar:'
    print '2-norm of out-of-subspace error, epsilon = {}'.format(epsilon)
    print '2-norm of in-subspace error = {}'.format(in_sub_2_norm)
    print 'Bound on 2-norm of in-subspace error = {}'.format(
        in_sub_2_norm_bound)
    print 'Sup-norm of in-subspace error = {}'.format(in_sub_sup_norm)
    print 'Bound on sup-norm of in-subspace error = {}'.format(
        in_sub_sup_norm_bound)
    print '2-norm of total error = {}'.format(tot_err_2_norm)
    print 'Bound on 2-norm of total error = {}'.format(tot_err_2_norm_bound)
    print '------------------------------------------------------------'

    return (soln_fig_1, soln_fig_2, err_fig_1, err_fig_2)

def main_function():
    """
    Purpose:
    Main driver function.

    Arguments:
    None.

    Returns:
    None.

    """

    # Set problem size
    n = 6

    # Set the seed of the random number generator for reproducibility.
    numpy.random.seed(0)
    numpy.random.rand(n)

    # "A" matrix from Rathinam and Petzold, "A New Look at Proper Orthogonal
    # Decomposition", SINUM, Vol. 41, No. 5, pp. 1893-1925 (2004).
    petzold_A_1 = numpy.asarray([[-0.1, 0, 0],
                                 [0, -0.1732, 2],
                                 [0, -2, -0.1732]])
    petzold_A_2 = numpy.asarray([[-1.0, 0, 0],
                                 [0, -1.226, -0.7080],
                                 [0, 0.7080, -1.226]])
    petzold_A_12 = numpy.asarray([[0.3893, 0.5179, -1.543],
                              [1.390, 1.3, 0.8841],
                              [0.06293, -0.9078, -1.184]])

petzold_A = numpy.vstack(
    (numpy.hstack((petzold_A_1, petzold_A_12)),
     numpy.hstack((numpy.zeros((3, 3)), petzold_A_2))))

# Use symmetrized Petzold matrix with spectrum shifted downward, so that
# spectrum is real, and its logarithmic 2-norm is negative. Set range
# basis to last three eigenvectors (which have nonzero entries in their
# last three components). Set null space equal to the first three standard
# Euclidean basis vectors.
symm_A = copy.copy(petzold_A)
symm_A[1, 2] = -2
symm_A[5, 4] = -0.7080
symm_A = symm_A - 2 * numpy.eye(n)
(eigen_val, eigen_vec) = numpy.linalg.eig(symm_A)
range_basis = eigen_vec[:, 3:6]
null_basis = numpy.vstack((numpy.zeros((3,3)), numpy.eye(3)))

# Generate random initial condition
#init_cond = numpy.random.rand(n)
init_cond = numpy.ones(n)

# Run case study on this coefficient matrix and choice of bases
(example_1_soln_1, example_1_soln_2, example_1_err_1,
 example_1_err_2) = comparison_of_models(symm_A,
    range_basis, null_basis, init_cond)

# Scale the upper right block of the symmetrized, shifted Petzold matrix,
# which corresonds to halving gamma in our bounds.
lower_gamma_A = copy.copy(symm_A)
lower_gamma_A[0:3, 3:6] = .5 * symm_A[0:3, 3:6]
(eigen_val, eigen_vec) = numpy.linalg.eig(lower_gamma_A)
range_basis = eigen_vec[:, 3:6]

# Run case study on coefficient matrix for second example
(example_2_soln_1, example_2_soln_2, example_2_err_1,
 example_2_err_2) = comparison_of_models(lower_gamma_A,
    range_basis, null_basis, init_cond)

# Scale the lower right block of the symmetrized, shifted Petzold matrix
# which corresponds to increasing mu in our bounds.
higher_mu_A = copy.copy(symm_A)
higher_mu_A[3:6, 3:6] = .715 * symm_A[3:6, 3:6]
(eigen_val, eigen_vec) = numpy.linalg.eig(higher_mu_A)
range_basis = eigen_vec[:, 3:6]

# Run case study on coefficient matrix for third example
(example_3_soln_1, example_3_soln_2, example_3_err_1,
 example_3_err_2) = comparison_of_models(higher_mu_A,
    range_basis, null_basis, init_cond)

matplotlib.pyplot.show()
```

```
    return

if __name__ == "__main__":
    main_function()
```

Appendix C

Implementation of Examples for Chapter 4

Examples for Chapter 4 were implemented in both MATLAB r2012a [133] and in Python 2.7.3 [209].

C.1 MATLAB Implementation

The MATLAB r2012a [133] implementation requires the installation of Sundials 2.4 (or later), and SundialsTB [85].

```
function IllustrativeCaseStudy()
% Case study that uses Petzold and Rathinam’s example in SINUM, 2004, but
% uses a different reduced model to illustrate a more general result.

% Close all plots
close all;
format long e;

% Get the default random number generation stream in MATLAB, and reset it
% for reproducibility. See "Loren on the Art of MATLAB", November 5, 2008,
% http://blogs.mathworks.com/loren/2008/11/05/ (continued on next line)
% new-ways-with-random-numbers-part-i/
stream0 = RandStream(’mt19937ar’,’Seed’,0);
RandStream.setDefaultStream(stream0);

% Size of matrices
n = 6;

% "A" matrix from Rathinam and Petzold, SINUM, 2004.
```

```
petzoldA1 = [-0.1, 0, 0;
              0, -0.1732, 2;
              0, -2.0, -0.1732;];

petzoldA2 = [-1.0, 0, 0;
             0, -1.226, -0.7080;
             0, 0.7080, -1.226;];

petzoldA12 = [0.3893, 0.5179, -1.543;
              1.390, 1.3, 0.8841;
              0.06293, -0.9078, -1.184;];

petzoldA = [petzoldA1, petzoldA12;
            zeros(3,3), petzoldA2];

% Block factors, used to make it so that gamma is determined by the upper
% block and muBar is determined by the lower block.
upperBlockFactor = 2;
lowerBlockFactor = 5;

% Modification of Petzold's coefficient matrix in order to make the example
% more presentable.
model1 = [upperBlockFactor * petzoldA1, petzoldA12;
       zeros(3,3), lowerBlockFactor * petzoldA2];

% Create reduced model by zeroing out the upper right 3 by 3 block of the
% full model coefficient matrix.
redModel1 = model1;
redModel1(1:3,4:6) = zeros(3,3);

% Random initial condition.
% initCond = rand(1,n);
initCond = ones(1,6);

upperBlockFactor = 2;
lowerBlockFactor = 10;

% Scaling factor for A12 block so that epsilon is unchanged in model2.
couplingFactor = 1.974500693397877;

% Modification of Petzold's coefficient matrix in order to make the example
% more presentable.
model2 = [upperBlockFactor * petzoldA1, couplingFactor * petzoldA12;
       zeros(3,3), lowerBlockFactor * petzoldA2];

% Create reduced model by zeroing out the upper right 3 by 3 block of the
% full model coefficient matrix.
redModel2 = model2;
redModel2(1:3,4:6) = zeros(3,3);

upperBlockFactor = 1;
lowerBlockFactor = 5;

% Modification of Petzold's coefficient matrix in order to make the example
% more presentable.
model3 = [upperBlockFactor * petzoldA1, petzoldA12;
       zeros(3,3), lowerBlockFactor * petzoldA2];

% Create reduced model by zeroing out the upper right 3 by 3 block of the
% full model coefficient matrix.
redModel3 = model3;
redModel3(1:3,4:6) = zeros(3,3);

comparisonOfModels(model1, redModel1, initCond);
comparisonOfModels(model2, redModel2, initCond);
comparisonOfModels(model3, redModel3, initCond);

end

% Auxiliary function encoding a linear system for numerical integration;
% here, A is the coefficient matrix, y is the state vector, and t is time.
function dy = linSys(t,y,A)
dy = A * y;
end

function comparisonOfModels(fullMatrix, redMatrix, initCond)
% Purpose: Integrates the linear system \dot{\mathbf{x}} = fullMatrix *
% \mathbf{x}, and also integrates the projected system
% \dot{\hat{\mathbf{x}}} = \mathbf{P} * fullMatrix * \hat{\mathbf{x}}.
% \mathbf{P} is a projection matrix that has range equal to the space
% spanned by the columns of rangeBasis, and null space equal to the space
% spanned by the columns of nullBasis.
% Input: fullMatrix = n by n, coefficient matrix of linear ODE system; n
%                       is determined by size of matrix
%        redMatrix = n by n, coefficient matrix of reduced model linear ODE
%                     system
%        initCond = 1 by n, initial condition for integration; row vector
%                    due to peculiarities of MATLAB syntax
% Output: None to scope outside of call. Will output information relevant
% to error bounds to terminal, either as text, or as plots.
% Assertion checks: None! Doesn't check for consistency, although error
% checks could be added later. NOTE: Many plots assume that n = 6; these
% can be generalized.

n = size(fullMatrix, 1);

% Calculate constants in Theorem 4.1 and Corollary 4.4:
% Original choice of V, W, Vperp, Wperp
gamma = norm(redMatrix);
muBar = max(real(eig(redMatrix + redMatrix')))/2;

% Now, to simulate system, emulating Rathinam and Petzold, SINUM, 2004.
tSpan = [0,5];
T = tSpan(2);

% Note that error tolerances set very tightly to decrease numerical error
% due to integration. Use 4th-order Runge-Kutta integration because system
% is not stiff.
```

```
options = odeset('RelTol',1e-13,'AbsTol',ones(1,n)*1e-25);
[tPts, fullSoln] = ode45(@linSys, tSpan, initCond, options, fullMatrix);
[~,redSoln] = ode45(@linSys, tPts, initCond, options, redMatrix);

% Calculate error in reduced model solution
errSoln = redSoln - fullSoln;

% Calculate both components of error.
outSubErrSoln = cumtrapz(tPts, (redMatrix - fullMatrix)*fullSoln', 2)';
inSubErrSoln = errSoln - outSubErrSoln;

% Trapezoidal rule approximation to epsilon, which seems to work well.
epsilon = sqrt(trapz(tPts, sum(outSubErrSoln.^2,2)));

% Calculate infinity norm of in-subspace error and compare to its predicted
% bound.
inSubInfNormBound = epsilon * gamma * ...
    sqrt( (exp(2 * muBar * T) - 1) / (2 * muBar) );
inSubInfNorm = max(max(abs(inSubErrSoln)));

inSub2NormBound = epsilon * gamma * ...
    sqrt( (exp(2 * muBar * T) - 1 - 2 * muBar * T) / (4 * muBar^2));
inSub2Norm = sqrt(trapz(tPts, sum(inSubErrSoln.^2,2)));

% Calculate 2-norm of total error and compare to its predicted bound.
% 2-norm of total error approximated using trapezoidal rule.
totErr2NormBound = epsilon * (1 + gamma * ...
    sqrt( (exp(2 * muBar * T) - 1 - 2 * muBar * T) / (4 * muBar^2)) );
totErr2Norm = sqrt(trapz(tPts, sum((errSoln').^2,1)));

% Statements to check code, and results.
fprintf(1, '------------------------------------------------------------\n')
fprintf(1, 'Size of matrix A (n by n), n = %e\n', n)
fprintf(1, 'gamma = %e\n', gamma);
fprintf(1, 'muBar = %e\n', muBar);
fprintf(1, '2-norm of truncating error, epsilon = %e\n', epsilon);
fprintf(1, '2-norm of propagating error = %e\n', inSub2Norm);
fprintf(1, 'Bound on 2-norm of propagating error = %e\n', inSub2NormBound);
fprintf(1, 'Sup-norm of propagating error = %e\n', inSubInfNorm);
fprintf(1, 'Bound on Sup-norm of propagating error = %e\n', ...
    inSubInfNormBound);
fprintf(1, '2-norm of total error = %e\n', totErr2Norm);
fprintf(1, 'Bound on 2-norm of total error = %e\n', totErr2NormBound)
fprintf(1, '------------------------------------------------------------\n')

% Plots:
% First, note that x_1, x_2, and x_3 should be different between the two
% models. Also, note that the only nonzero components of the error should
% be e_1, e_2, and e_3.

figure;
plot(tPts, fullSoln(:,1), 'r-');
hold on;
plot(tPts, fullSoln(:,2), 'b-');
plot(tPts, fullSoln(:,3), 'k-');
plot(tPts, redSoln(:,1), 'r--');
plot(tPts, redSoln(:,2), 'b--');
plot(tPts, redSoln(:,3), 'k--');
title('Comparison of full and reduced model solutions');
xlabel('Time (t) [a.u.]');
ylabel('State variable (x_j) [a.u.]');
legend('full, 1', 'full, 2', 'full, 3', 'reduced, 1', 'reduced, 2', ...
    'reduced, 3', 'Location', 'Best');

figure;
plot(tPts, errSoln(:,1), 'r-');
hold on;
plot(tPts, errSoln(:,2), 'b-');
plot(tPts, errSoln(:,3), 'k-');
plot(tPts, inSubErrSoln(:,1), 'r--');
plot(tPts, inSubErrSoln(:,2), 'b--');
plot(tPts, inSubErrSoln(:,3), 'k--');
title('Error in reduced model solution');
xlabel('Time (t) [a.u.]');
ylabel('Error in state variable (e_j) [a.u.]');
legend('total, 1', 'total, 2', 'total, 3', 'propagating, 1', ...
    'propagating, 2', 'propagating, 3', 'Location', 'Best');

end
```

C.2 Python Implementation

The Python 2.7.3 [209] implementation requires the installation of NumPy 1.6.2 (or later) [152], SciPy 0.10.1 (or later) [93], and Matplotlib 1.0.0 (or later) [90]. An attempt was made to keep the number of dependencies to a minimum. It is likely that the Python code below will work with Python 2.6 (or later).

```
#!/usr/bin/env python

import numpy
import scipy.integrate
import matplotlib.pyplot
import math
import copy

def lin_sys(t, y, A):
    """
    Purpose:
    Auxiliary function encoding a linear ODE system for numerical integration.

    Arguments:
    t (float): time
    y (1-D numpy.ndarray): states
    A (2-D numpy.ndarray): square coefficient array, where A.shape[1] == len(y)

    Returns:
    f (1-D numpy.ndarray): right-hand side of ODE

    """

    f = numpy.dot(A, y)
    return f

def comparison_of_models(full_matrix, red_matrix, init_cond):
    """
    Purpose:
    Integrates the linear system \dot{\mathbf{x}} = full_matrix * \mathbf{x},
    and also integrates the reduced system \dot{\hat{\mathbf{x}}} =
    red_matrix * \hat{\mathbf{x}}.

    Arguments:
    full_matrix (2-D numpy.ndarray of floats): coefficient matrix of "full"
    linear ODE system, square array
    red_matrix (2-D numpy.ndarray of floats): coefficient matrix of "reduced"
    linear ODE system, square array, red_matrix.shape == full_matrix.shape
    init_cond (1-D numpy.ndarray of floats): initial condition for integration,
    row vector due to numpy syntax; len(init_cond) == full_matrix.shape[1]

    Returns:
    soln_fig (matplotlib.pyplot.Figure): figure containing plots of full and
    reduced solution
    err_fig (matplotlib.pyplot.Figure): figure containing plots of total
    error and propagating error

    Assertion checks:
    None; doesn't check for consistency, though error checks could be added later.
    Many plots assume that len(init_cond) == 6; these could be generalized

    """

    # Calculate constants in Theorem 4.1 and Corollary 4.4:
    gamma = numpy.linalg.norm(red_matrix, 2)
    mu_bar = numpy.max(numpy.real(
        numpy.linalg.eigvals(red_matrix + red_matrix.transpose())))/2

    # Simulate system, emulating Rathinam and Petzold, SINUM, 2004.
    t_begin = 0
    t_end = 5
    n_time_pts = 1000
    t = numpy.linspace(t_begin, t_end, n_time_pts)

    # Set up numerical integrators. Note that the error tolerances are set
    # very tightly to decrease numerical error due to integration. Use 7th-order
    # explicit Runge-Kutta integration because systems are not stiff.
    full_sys = scipy.integrate.ode(lin_sys)
    full_sys.set_integrator('dop853', atol=1e-25, rtol=1e-13, nsteps=10000000)
full_sys.set_initial_value(init_cond, 0)
full_sys.set_f_params(full_matrix)

red_sys = scipy.integrate.ode(lin_sys)
red_sys.set_integrator(’dop853’, atol=1e-25, rtol=1e-3, nsteps=10000000)
red_sys.set_initial_value(init_cond, 0)
red_sys.set_f_params(red_matrix)

# Run integration loops; use numpy.vstack to avoid the need for copying
# state of integrators.
full_soln = init_cond
for point in t[1:]:
    if not full_sys.successful(): break
    full_sys.integrate(point)
    full_soln = numpy.vstack((full_soln, full_sys.y))

red_soln = init_cond
for point in t[1:]:
    if not red_sys.successful(): break
    red_sys.integrate(point)
    red_soln = numpy.vstack((red_soln, red_sys.y))

# Calculate error in reduced model solution
err_soln = red_soln - full_soln

# Calculate both components of error
#out_sub_err_soln = scipy.integrate.cumtrapz(
#    numpy.dot((red_matrix - full_matrix), full_soln.transpose()),
#    t, axis=1).transpose()
t_matrix = numpy.tile(t, (len(init_cond), 1)).transpose()
out_sub_err_soln = scipy.integrate.cumtrapz(
    numpy.dot(full_soln, (red_matrix - full_matrix).transpose()),
    t_matrix, axis=0)
out_sub_err_soln = numpy.vstack((numpy.zeros(len(init_cond)),
                                 out_sub_err_soln))
in_sub_err_soln = err_soln - out_sub_err_soln

# Trapezoidal rule approximation to epsilon, which seems to work well.
epsilon = numpy.sqrt(numpy.trapz(
    numpy.sum(out_sub_err_soln ** 2, axis=1), t))

# Calculate infinity norm of in-subspace error and compare to its
# predicted bound.
T = t_end
in_sub_inf_norm_bound = (epsilon * gamma * math.sqrt(
    (math.exp(2 * mu_bar * T) - 1) / (2 * mu_bar)))
in_sub_inf_norm = numpy.max(numpy.abs(in_sub_err_soln))

in_sub_2_norm_bound = (epsilon * gamma * math.sqrt(
    (math.exp(2 * mu_bar * T) - 1 - 2 * mu_bar * T) / (4 * mu_bar ** 2)))
in_sub_2_norm = math.sqrt(numpy.trapz(
    numpy.sum(in_sub_err_soln ** 2, axis=1), t))

# Calculate 2-norm of total error and compare to its predicted bound.
# 2-norm of total error approximated using trapezoidal rule.
tot_err_2_norm_bound = (epsilon * (1 + gamma * math.sqrt(
    (math.exp(2 * mu_bar * T) - 1 - 2 * mu_bar * T) / (4 * mu_bar ** 2))))
tot_err_2_norm = numpy.sqrt(numpy.trapz(
    numpy.sum(err_soln ** 2, axis=1), t))

# Statements to check code, and results
print '-----------------------------------------------------------'
print 'Size of matrix A (n by n), n = {}'.format(full_matrix.shape[0])
print 'gamma = {}'.format(gamma)
print 'mu_bar = {}'.format(mu_bar)
print '2-norm of truncating error, epsilon = {}'.format(epsilon)
print '2-norm of propagating error = {}'.format(in_sub_2_norm)
print 'Bound on 2-norm of propagating error = {}'.format(
    in_sub_2_norm_bound)
print 'Sup-norm of propagating error = {}'.format(in_sub_inf_norm)
print 'Bound on Sup-norm of propagating error = {}'.format(
    in_sub_inf_norm_bound)
print '2-norm of total error = {}'.format(tot_err_2_norm)
print 'Bound on 2-norm of total error = {}'.format(tot_err_2_norm_bound)
print '-----------------------------------------------------------'

# Plots:
# For the inputs given, x_1, x_2, and x_3 should be different between the
# two models. Also, note that the only nonzero error components for these
# inputs should be e_1, e_2, and e_3.
soln_fig = matplotlib.pyplot.figure()
matplotlib.pyplot.plot(t, full_soln[:, 0], 'r-')
matplotlib.pyplot.plot(t, full_soln[:, 1], 'b-')
matplotlib.pyplot.plot(t, full_soln[:, 2], 'k-')
matplotlib.pyplot.plot(t, red_soln[:, 0], 'r--')
matplotlib.pyplot.plot(t, red_soln[:, 1], 'b--')
matplotlib.pyplot.plot(t, red_soln[:, 2], 'k--')
matplotlib.pyplot.title('Comparison of full and reduced model solutions')
matplotlib.pyplot.xlabel('Time (t) [a.u.]')
matplotlib.pyplot.ylabel('State variable (x_j) [a.u.]')
matplotlib.pyplot.legend( ('full, 1', 'full, 2', 'full, 3',
    'reduced, 1', 'reduced, 2', 'reduced, 3'), loc='best')

err_fig = matplotlib.pyplot.figure()
matplotlib.pyplot.plot(t, err_soln[:, 0], 'r-')
matplotlib.pyplot.plot(t, err_soln[:, 1], 'b-')
matplotlib.pyplot.plot(t, err_soln[:, 2], 'k-')
matplotlib.pyplot.plot(t, in_sub_err_soln[:, 0], 'r--')
matplotlib.pyplot.plot(t, in_sub_err_soln[:, 1], 'b--')
matplotlib.pyplot.plot(t, in_sub_err_soln[:, 2], 'k--')
matplotlib.pyplot.title('Error in reduced model solution')
matplotlib.pyplot.xlabel('Time (t) [a.u.]')
matplotlib.pyplot.ylabel('Error in state variable (e_j) [a.u.]')
matplotlib.pyplot.legend( ('total, 1', 'total, 2', 'total, 3',
    'propagating, 1', 'propagating, 2', 'propagating, 3'), loc='best')

return soln_fig, err_fig
```

```
def main_function():
    """
    Purpose:
    Main driver function.

    Arguments:
    None

    Returns:
    None.

    """

    # Set problem size
    n = 6

    # Set the seed of the random number generator for reproducibility.
    numpy.random.seed(0)
    numpy.random.rand(n)

    # "A" matrix from Rathinam and Petzold, "A New Look at Proper Orthogonal
    # Decomposition", SINUM, Vol. 41, No. 5, pp. 1893-1925 (2004).
    petzold_A_1 = numpy.asarray([[-0.1, 0, 0],
                                 [0, -0.1732, 2],
                                 [0, -2, -0.1732]])
    petzold_A_2 = numpy.asarray([[-1.0, 0, 0],
                                 [0, -1.226, -0.7080],
                                 [0, 0.7080, -1.226]])
    petzold_A_12 = numpy.asarray([[0.3893, 0.5179, -1.543],
                                  [1.390, 1.3, 0.8841],
                                  [0.06293, -0.9078, -1.184]])

    petzold_A = numpy.vstack(
        (numpy.hstack((petzold_A_1, petzold_A_12)),
         numpy.hstack((numpy.zeros((3, 3)), petzold_A_2))))

    # Block factors, used to make it so that gamma is determined by the upper
    # block and mu_bar is determined by the lower block.
    upper_block_factor_1 = 2
    lower_block_factor_1 = 5

    # Modification of Petzold and Rathinam's coefficient matrix in order to
    # make the example more presentable.
    model_1 = numpy.vstack(
        (numpy.hstack((upper_block_factor_1 * petzold_A_1, petzold_A_12)),
         numpy.hstack((numpy.zeros((3, 3)),
         lower_block_factor_1 * petzold_A_2)))
         )

    # Create reduced model by zeroing out the upper right 3 by 3 block of the
    # full model coefficient matrix
    red_model_1 = copy.copy(model_1)
    red_model_1[0:3,3:6] = numpy.zeros((3,3))
    # Random initial condition
    #init_cond = numpy.random.rand(n)
    init_cond = numpy.ones(n)

    # Second set of block factors, used to make it so that gamma is determined
    # by the upper block and mu_bar and is determined by the lower block.
    # Gamma is doubled from the previous example. The coupling_factor,
    # which is a scaling factor for the A_12 block, is applied so that epsilon
    # is unchanged from the first example.
    upper_block_factor_2 = 2
    lower_block_factor_2 = 10
    coupling_factor_2 = 1.974500693397877

    # Second example matrix
    model_2 = numpy.vstack(
        (numpy.hstack((upper_block_factor_2 * petzold_A_1,
         coupling_factor_2 * petzold_A_12)),
         numpy.hstack((numpy.zeros((3, 3)), lower_block_factor_2 * petzold_A_2)))
         )

    # Create reduced model from second example matrix by zeroing out the upper
    # right 3 by 3 block of the full model coefficient matrix
    red_model_2 = copy.copy(model_2)
    red_model_2[0:3, 3:6] = numpy.zeros((3,3))

    # Third set of block factors
    upper_block_factor_3 = 1
    lower_block_factor_3 = 5

    # Third set of block factors to make the third example
    model_3 = numpy.vstack(
        (numpy.hstack((upper_block_factor_3 * petzold_A_1, petzold_A_12)),
         numpy.hstack((numpy.zeros((3, 3)),
         lower_block_factor_3 * petzold_A_2)))
         )

    # Create reduced model by zeroing out the upper right 3 by 3 block of the
    # full model coefficient matrix
    red_model_3 = copy.copy(model_3)
    red_model_3[0:3, 3:6] = numpy.zeros((3,3))

    (soln_fig_1,
     err_fig_1) = comparison_of_models(model_1, red_model_1, init_cond)
    (soln_fig_2,
     err_fig_2) = comparison_of_models(model_2, red_model_2, init_cond)
    (soln_fig_3,
     err_fig_3) = comparison_of_models(model_3, red_model_3, init_cond)

    matplotlib.pyplot.show()

    return

if __name__ == "__main__":
    main_function()
```

Appendix D

Implementation of Point-Constrained Reaction Elimination and Point-Constrained Simultaneous Reaction and Species Elimination Formulations in Chapter 5

A reference implementation of point-constrained reaction elimination and point-constrained simultaneous reaction and species elimination is given in Python so that it may be used as a basis for future reproducible research. In addition, unit tests are also given to ensure that

D.1 Python Implementation

The Python 2.7.3 [209] implementation requires the installation of Cantera 2.0.0b3 (or later) [73], the Cantera Python interface, NumPy 1.6.2 (or later) [152], and PuLP 1.4.9 (or later) [135]. The PuLP package includes interfaces to multiple open-source and proprietary solvers (including CPLEX and Gurobi). An attempt was made to keep the number of dependencies to a minimum. It is likely that the Python code below will work with Python 2.6 (or later).

```
#/usr/bin/env python

# Requirements:
# NumPy
# Cantera
# PuLP

import numpy
import Cantera
import pulp

# Optional:
# Installed LP solvers (Gurobi, CPLEX, CBC, COIN)

def calc_cond_indep_data(ideal_gas):
    """
    Purpose: Calculate the condition-independent data needed for reaction
    elimination: the molar mass-stoichiometric matrix product

    Arguments:
    ideal_gas (Cantera.Solution): Cantera.Solution object specifying
    a chemical reaction mechanism and the thermodynamic properties of
    its constituent species

    Returns:
    mass_stoich_prod (2-D numpy.ndarray of floats): product of diagonal
    matrix of molar masses and stoichiometry matrix
    """

    molar_mass = ideal_gas.molarMasses()
    stoich_matrix = (ideal_gas.productStoichCoeffs() -
        ideal_gas.reactantStoichCoeffs())
    mass_stoich_prod = numpy.dot(numpy.diag(molar_mass), stoich_matrix)

    return stoich_matrix, mass_stoich_prod

def calc_cond_dep_data(state, ideal_gas):
    """
    Purpose: Calculate the condition-dependent data needed for reaction
    elimination:
    - species mass enthalpies
    - reaction rates
    - mass-based constant pressure heat capacity
    - mass density

    Arguments:
    state (list of floats, or 1-D numpy.ndarray of floats): Reaction
    conditions consisting of temperature and species mass fractions
    (in the order that they are specified in the Cantera mechanism).
    Temperature must be the first element in the system state list
    (or 1-D numpy.ndarray); subsequent elements must be species mass
    fractions, in the order that they are specified in the Cantera
    mechanism.
    ideal_gas (Cantera.Solution): Cantera.Solution object specifying a
    chemical reaction mechanism and the thermodynamic properties of its
    constituent species; uses state of ideal_gas to calculate properties

    Returns:
    rxn_rate (1-D numpy.ndarray of floats): (row) vector of reaction rates
    cp_mass (float): mass-based constant pressure heat capacity
    enthalpy_mass (1-D numpy.ndarray of floats): (row) vector of species
        mass (or specific) enthalpies
    rho (float): mass density

    """

    ideal_gas.setTemperature(state[0])
    ideal_gas.setMassFractions(state[1:])

    rxn_rate = ideal_gas.netRatesOfProgress()
    cp_mass = ideal_gas.cp_mass()
    enthalpy_mass = (ideal_gas.enthalpies_RT() *
        ideal_gas.temperature() * Cantera.GasConstant)
    rho = ideal_gas.density()

    return (rxn_rate, cp_mass, enthalpy_mass, rho)

def error_constraint_data(state, ideal_gas, mass_stoich_prod, atol, rtol):
    """
    Purpose: Calculates all of the coefficients for the error constraints
    in the point-constrained reaction and species elimination integer
    linear programming formulations.

    Arguments:
    state (list of floats, or 1-D numpy.ndarray of floats): Reaction
    conditions consisting of temperature and species mass fractions
    (in the order that they are specified in the Cantera mechanism).
    Temperature must be the first element in the system state list
    (or 1-D numpy.ndarray); subsequent elements must be species mass
    fractions, in the order that they are specified in the Cantera
    mechanism.
    ideal_gas (Cantera.Solution): Cantera.Solution object specifying a
    chemical reaction mechanism and the thermodynamic properties of its
    constituent species; uses state of ideal_gas to calculate properties
    atol (1-D numpy.ndarray of floats): list of absolute tolerances;
    len(atol) == states.shape[1] == ideal_gas.nSpecies() + 1
    rtol (1-D numpy.ndarray of floats): list of relative tolerances;
    len(rtol) == states.shape[1] == ideal_gas.nSpecies() + 1
    mass_stoich_prod (2-D numpy.ndarray of floats): product of diagonal
    matrix of molar masses and stoichiometry matrix

    Returns:
    coeffs_temp (1-D numpy.ndarray of floats): coefficients for constraints
        on error in time derivative of temperature
    coeffs_y (2-D numpy.ndarray of floats): coefficients for constraints on
        on error in time derivatives of species mass fractions
    rhs_temp (float): right-hand side of constraints on error in time
        derivative of temperature
    rhs_y (1-D numpy.ndarray of floats): right-hand side of constraints on
        error in time derivatives of species mass fractions

    Comments:
    Could refactor this function to use the internal state of ideal_gas,
    but the additional state argument was chosen to make the dependency
    much more explicit.

    """

    (rxn_rate,
     cp_mass,
     enthalpy_mass,
     rho) = calc_cond_dep_data(state, ideal_gas)

    coeffs_temp = numpy.dot(enthalpy_mass, numpy.dot(mass_stoich_prod,
        numpy.diag(rxn_rate))) / (rho * cp_mass)
    temp_dot = numpy.dot(coeffs_temp, rxn_rate)
    rhs_temp = atol[0] + rtol[0] * abs(temp_dot)

    ydot = numpy.dot(mass_stoich_prod, rxn_rate) / rho
    coeffs_y = numpy.dot(mass_stoich_prod, numpy.diag(rxn_rate)) / rho
    rhs_y = atol[1:] + numpy.dot(abs(ydot), numpy.diag(rtol[1:]))

    return coeffs_temp, coeffs_y, rhs_temp, rhs_y

def reaction_elim(states, ideal_gas, atol, rtol,
                  lpsolver=pulp.solvers.GLPK_CMD()):
    """
    Purpose: Carries out reaction elimination (Bhattacharjee, et al.,
    Comb Flame, 2003) on the mechanism specified in ideal_gas at the
    conditions specified in states, using the absolute tolerances
    specified in atol, and relative tolerances specified in rtol.

    Arguments:
    states (list of list of floats, or 2-D numpy.ndarray of floats):
    each element of the outer list (or each row of the 2-D
    numpy.ndarray) corresponds to a system state (or condition).
    Conditions consist of temperature and species mass fractions
    (in the order that they are specified in the Cantera mechanism).
    Temperature must be the first element in the system state list
    (or 1-D numpy.ndarray); subsequent elements must be species mass
    fractions, in the order that they are specified in the Cantera
    mechanism.
    ideal_gas (Cantera.Solution): Cantera.Solution object specifying
    a chemical reaction mechanism and the thermodynamic properties of
    its constituent species
    atol (list of floats or 1-D numpy.ndarray of floats): list of
    absolute tolerances; len(atol) == states.shape[1] ==
    ideal_gas.nSpecies() + 1
```

```
rtol (list of floats or 1-D numpy.ndarray of floats): list of
relative tolerances; len(rtol) == states.shape[1] ==
ideal_gas.nSpecies() + 1
lpsolver (pulp solver command): One of the solver commands listed
when running pulp.pulpTestAll(), such as:

pulp.solvers.PULP_CBC_CMD()
pulp.solvers.CPLEX_DLL()
pulp.solvers.CPLEX_CMD()
pulp.solvers.CPLEX_PY()
pulp.solvers.COIN_CMD()
pulp.solvers.COINMP_DLL()
pulp.solvers.GLPK_CMD()
pulp.solvers.XPRESS()
pulp.solvers.GUROBI()
pulp.solvers.GUROBI_CMD()
pulp.solvers.PYGLPK()
pulp.solvers.YAPOSIB()

These solvers also have optional arguments; see the PuLP documentation
for details. This argument allows one to change the solver and solver
options in the API call.

Returns:
z (list of ints, or 1-D numpy.ndarray of ints): binary variables
indicating which reactions should be kept, and which should be
eliminated
status (str): indicates the LP solver status; is one of "Not
Solved", "Infeasible", "Unbounded", "Undefined", "Optimal"

Warnings:
This function alters the state of ideal_gas. If the state of that
object prior to calling this function needs to be preserved,
copy the object.

"""

# Convert lists to numpy.ndarrays because the data structure is useful
# for the operators.
atol = numpy.asarray(atol)
rtol = numpy.asarray(rtol)

# Set up the lists needed for indexing
rxn_list = range(0, ideal_gas.nReactions())
rxn_strings = [str(n+1) for n in rxn_list]

# Instantiate binary variables for integer linear program
z_var = pulp.LpVariable.dicts('rxn_', rxn_strings, 0, 1, 'Integer')

# Instantiate integer linear program and objective function
rxn_elim_ILP = pulp.LpProblem("Reaction Elimination", pulp.LpMinimize)
rxn_elim_ILP += pulp.lpSum([z_var[s] for s
                           in rxn_strings]), "Number of reactions"
```

```
    # Calculate condition-independent data and store
    (stoich_matrix, mass_stoich_prod) = calc_cond_indep_data(ideal_gas)
    ideal_gas.setPressure(Cantera.OneAtm)

    # Add constraints: loop over data points
    for k in range(0, len(states)):

        # Calculate condition-dependent data
        (coeffs_temp, coeffs_y, rhs_temp,
        rhs_y) = error_constraint_data(states[k],
            ideal_gas, mass_stoich_prod, atol, rtol)

        # Add two temperature error constraints (lower, upper bounds)
        rxn_elim_ILP += pulp.lpSum([coeffs_temp[i] *
            (1 - z_var[rxn_strings[i]]) for i in rxn_list]) >= -rhs_temp, \
            "Temperature Error Lower Bound for Data Point " + str(k+1)
        rxn_elim_ILP += pulp.lpSum([coeffs_temp[i] *
            (1 - z_var[rxn_strings[i]]) for i in rxn_list]) <= rhs_temp, \
            "Temperature Error Upper Bound for Data Point " + str(k+1)

        # Add constraints: Loop over species mass fractions
        for j in range(0, ideal_gas.nSpecies()):

            # Add two species mass fraction error constraints (lower, upper
            # bounds)
            rxn_elim_ILP += pulp.lpSum([coeffs_y[j, i] *
                (1 - z_var[rxn_strings[i]]) for i in rxn_list]) >= -rhs_y[j], \
                "Mass Fraction Species " + str(j+1) + \
                " Error Lower Bound for Data Point " + str(k+1)
            rxn_elim_ILP += pulp.lpSum([coeffs_y[j, i] *
                (1 - z_var[rxn_strings[i]]) for i in rxn_list]) <= rhs_y[j], \
                "Mass Fraction Species " + str(j+1) + \
                " Error Upper Bound for Data Point " + str(k+1)

    # Solve integer linear program
    rxn_elim_ILP.solve(solver=lpsolver)

    # Return list of binary variables, solver status
    z = [int(z_var[i].value()) for i in rxn_strings]
    #z = [int(v.value()) for v in rxn_elim_ILP.variables()]
    return z, pulp.LpStatus[rxn_elim_ILP.status]

def reaction_and_species_elim(states, ideal_gas, atol, rtol,
                              lpsolver=pulp.solvers.GLPK_CMD()):
    """
    Purpose: Carries out simultaneous reaction and species
    elimination (Mitsos, et al.,Comb Flame, 2008;
    Mitsos, 2008, unpublished) on the mechanism specified in
    ideal_gas at the conditions specified in states, using the
    absolute tolerances specified in atol, and relative tolerances
    specified in rtol. Mitsos' unpublished formulation is used here,
    which decreases the number of integer variables in the mixed-integer
    linear programming formulation, which decreases its run time compared
    to the original formulation in Combustion and Flame.
```

```
Arguments:
states (list of list of floats, or 2-D numpy.ndarray of floats):
each element of the outer list (or each row of the 2-D
numpy.ndarray) corresponds to a system state (or condition).
Conditions consist of temperature and species mass fractions
(in the order that they are specified in the Cantera mechanism).
Temperature must be the first element in the system state list
(or 1-D numpy.ndarray); subsequent elements must be species mass
fractions, in the order that they are specified in the Cantera
mechanism.
ideal_gas (Cantera.Solution): Cantera.Solution object specifying
a chemical reaction mechanism and the thermodynamic properties of
its constituent species
atol (list of floats or 1-D numpy.ndarray of floats): list of
absolute tolerances; len(atol) == states.shape[1] ==
ideal_gas.nSpecies() + 1
rtol (list of floats or 1-D numpy.ndarray of floats): list of
relative tolerances; len(rtol) == states.shape[1] ==
ideal_gas.nSpecies() + 1
lpsolver (pulp solver command): One of the solver commands listed
when running pulp.pulpTestAll(), such as:

pulp.solvers.PULP_CBC_CMD()
pulp.solvers.CPLEX_DLL()
pulp.solvers.CPLEX_CMD()
pulp.solvers.CPLEX_PY()
pulp.solvers.COIN_CMD()
pulp.solvers.COINMP_DLL()
pulp.solvers.GLPK_CMD()
pulp.solvers.XPRESS()
pulp.solvers.GUROBI()
pulp.solvers.GUROBI_CMD()
pulp.solvers.PYGLPK()
pulp.solvers.YAPOSIB()

These solvers also have optional arguments; see the PuLP documentation
for details. This argument allows one to change the solver and solver
options in the API call.

Returns:
z (list of ints, or 1-D numpy.ndarray of ints): binary variables
indicating which reactions should be kept, and which should be
eliminated
w (list of ints, or 1-D numpy.ndarray of ints): binary variables
indicating which species should be kept, and which should be
eliminated
status (str): indicates the LP solver status; is one of "Not
Solved", "Infeasible", "Unbounded", "Undefined", "Optimal"

Warnings:
This function alters the state of ideal_gas. If the state of that
object prior to calling this function needs to be preserved,
copy the object.
```

```
"""

# Convert lists to numpy.ndarrays because the data structure is useful
# for the operators.
atol = numpy.asarray(atol)
rtol = numpy.asarray(rtol)

# Set up the lists needed for indexing
rxn_list = range(0, ideal_gas.nReactions())
rxn_strings = [str(n+1) for n in rxn_list]

species_list = range(0, ideal_gas.nSpecies())
species_strings = [str(n+1) for n in species_list]

# Instantiate binary variables for integer linear program
z_var = pulp.LpVariable.dicts('rxn_', rxn_strings, 0, 1, 'Integer')
w_var = pulp.LpVariable.dicts('species_', species_strings, 0, 1,
                              'Continuous')

# Instantiate integer linear program and objective function
rxn_elim_ILP = pulp.LpProblem("Reaction Elimination", pulp.LpMinimize)
rxn_elim_ILP += pulp.lpSum([w_var[s] for s
                            in species_strings]), "Number of species"

# Calculate condition-independent data and store
(stoich_matrix, mass_stoich_prod) = calc_cond_indep_data(ideal_gas)
ideal_gas.setPressure(Cantera.OneAtm)

# Add participation constraints from alternative Mitsos formulation
for j in range(0, ideal_gas.nSpecies()):
    for i in range(0, ideal_gas.nReactions()):
        if stoich_matrix[j, i] != 0:
            rxn_elim_ILP += \
                w_var[species_strings[j]] - z_var[rxn_strings[i]] >= 0, \
                "Participation of species " + str(j+1) + \
                " and reaction " + str(i+1)

# Add error constraints: loop over data points
for k in range(0, len(states)):

    # Calculate condition-dependent data
    (coeffs_temp, coeffs_y, rhs_temp,
    rhs_y) = error_constraint_data(states[k],
        ideal_gas, mass_stoich_prod, atol, rtol)

    # Add two temperature error constraints (lower, upper bounds)
    rxn_elim_ILP += pulp.lpSum([coeffs_temp[i] *
        (1 - z_var[rxn_strings[i]]) for i in rxn_list]) >= -rhs_temp, \
        "Temperature Error Lower Bound for Data Point " + str(k+1)
    rxn_elim_ILP += pulp.lpSum([coeffs_temp[i] *
        (1 - z_var[rxn_strings[i]]) for i in rxn_list]) <= rhs_temp, \
        "Temperature Error Upper Bound for Data Point " + str(k+1)
        # Add constraints: Loop over species mass fractions
        for j in range(0, ideal_gas.nSpecies()):

            # Add two species mass fraction error constraints (lower, upper
            # bounds)
            rxn_elim_ILP += pulp.lpSum([coeffs_y[j, i] *
                (1 - z_var[rxn_strings[i]]) for i in rxn_list]) >= -rhs_y[j], \
                "Mass Fraction Species " + str(j+1) + \
                " Error Lower Bound for Data Point " + str(k+1)
            rxn_elim_ILP += pulp.lpSum([coeffs_y[j, i] *
                (1 - z_var[rxn_strings[i]]) for i in rxn_list]) <= rhs_y[j], \
                "Mass Fraction Species " + str(j+1) + \
                " Error Upper Bound for Data Point " + str(k+1)

    # Solve integer linear program
    rxn_elim_ILP.solve(solver=lpsolver)

    # Return list of binary variables, solver status
    z = [int(z_var[i].value()) for i in rxn_strings]
    w = [int(w_var[j].value()) for j in species_strings]
    return z, w, pulp.LpStatus[rxn_elim_ILP.status]
```

D.2 Python Unit Tests

Unit tests are provided here to ensure that any modifications to the code do not break existing functionality, and to guard against errors in the Python implementation above.

```
import chemReduce
import Cantera
import unittest
import numpy

class TestCoeffIdentities(unittest.TestCase):
    def setUp(self):
        #methodName='runTest', file_name='gri30.cti', temp=1000,
        #press=Cantera.OneAtm, mass_frac='CH4:.05, O2:.075, N2:.9', atol=1e-6,
        #rtol=1e-6):

        file_name = 'gri30.cti'
        temp = 1000
        press = Cantera.OneAtm
        mass_frac = 'CH4:.05, O2:.075, N2:.9"'
        atol = 1e-6
        rtol = 1e-6

        # Initialize thermodynamic and kinetic data and set state
        self.gas=Cantera.IdealGasMix('gri30.cti')
        self.gas.set(T=temp, P=press, Y=mass_frac)
        self.state = numpy.zeros(self.gas.nSpecies() + 1)
        self.state[0] = self.gas.temperature()
        self.state[1:] = self.gas.massFractions()

        # Calculate condition-independent data and store
        (self.stoich_matrix,
         self.mass_stoich_prod) = chemReduce.calc_cond_indep_data(self.gas)

        # Calculate condition-dependent data and store
        (self.rxn_rate,
         self.cp_mass,
         self.enthalpy_mass,
         self.rho) = chemReduce.calc_cond_dep_data(self.state, self.gas)

        self.atol = numpy.ones(self.gas.nSpecies() + 1) * atol
        self.rtol = numpy.ones(self.gas.nSpecies() + 1) * rtol

        self.float_tol = 1e-6

    def test_row_sums(self):
        """
        Purpose: The sum of the entries in coeffs_temp should equal temp_dot.
        The sum of the entries in each row of coeffs_y should equal
        numpy.asarray([ydot]).transpose().

        Arguments:
        None

        Returns:
        None

        """
        (coeffs_temp,
         coeffs_y,
         rhs_temp,
         rhs_y) = chemReduce.error_constraint_data(self.state,
            self.gas, self.mass_stoich_prod, self.atol, self.rtol)

        # Test identity for temperature
        rhs_temp_test = self.atol[0] + self.rtol[0] * numpy.sum(coeffs_temp)
        self.assertAlmostEqual(numpy.max(abs(rhs_temp_test - rhs_temp)), 0,
                               delta=self.float_tol)

        # Test identity for each species
        rhs_y_test = numpy.zeros(self.gas.nSpecies())
        for j in range(0, self.gas.nSpecies()):
            rhs_y_test[j] = (self.atol[j + 1] + self.rtol[j + 1] *
                             numpy.sum(coeffs_y[j,:]))

        self.assertAlmostEqual(numpy.max(abs(rhs_y_test - rhs_y)), 0,
                               delta=self.float_tol)

    def test_col_sums(self):
        """
        Purpose: The sum of over each column in coeffs_y, where each row is
        scaled by enthalpy_mass[j] / cp_mass, should equal coeffs_t.

        Arguments:
        None

        Returns:
        None
        """
        (coeffs_temp,
         coeffs_y,
         _,
         _) = chemReduce.error_constraint_data(self.state,
            self.gas, self.mass_stoich_prod, self.atol, self.rtol)

        row_total = numpy.zeros(self.gas.nReactions())
        for j in range(0, self.gas.nSpecies()):
            row_total += self.enthalpy_mass[j] * coeffs_y[j] / self.cp_mass

        self.assertAlmostEqual(numpy.max(abs(row_total - coeffs_temp)), 0,
                               delta = self.float_tol)

    def test_naive_summation(self):
        """
        Purpose: Calculate the entries of coeffs_temp, coeffs_y, rhs_temp,
        rhs_y using loops instead of vectorizing. Will be slow, but should
        yield same answer.

        Arguments:
        None

        Returns:
        None
        """

        molarMass = self.gas.molarMasses()
        stoichMatrix = (self.gas.productStoichCoeffs() -
            self.gas.reactantStoichCoeffs())

        coeffs_y_loop = numpy.zeros((self.gas.nSpecies(),
                                    self.gas.nReactions()))
        coeffs_temp_loop = numpy.zeros(self.gas.nReactions())

        for i in range(0, self.gas.nReactions()):
            coeffs_temp_loop[i] = numpy.sum(
                [self.enthalpy_mass[j] * molarMass[j] * stoichMatrix[j,i] *
                 self.rxn_rate[i] / (self.cp_mass * self.rho)
                 for j in range(0, self.gas.nSpecies())])
            for j in range(0, self.gas.nSpecies()):
                coeffs_y_loop[j,i] = (molarMass[j] * stoichMatrix[j,i] *
                    self.rxn_rate[i] / self.rho)

        temp_dot = numpy.sum(coeffs_temp_loop)
        y_dot = numpy.sum(coeffs_y_loop, axis=1)
        rhs_temp_loop = self.atol[0] + self.rtol[0] * abs(temp_dot)
        rhs_y_loop = numpy.zeros(self.gas.nSpecies())
        for j in range(0, self.gas.nSpecies()):
            rhs_y_loop[j] = self.atol[j+1] + self.rtol[j+1] * abs(y_dot[j])

        (coeffs_temp,
         coeffs_y,
         rhs_temp,
         rhs_y) = chemReduce.error_constraint_data(self.state,
            self.gas, self.mass_stoich_prod, self.atol, self.rtol)

        self.assertAlmostEqual(rhs_temp, rhs_temp_loop, delta=self.float_tol)

        self.assertAlmostEqual(numpy.max(abs(rhs_y - rhs_y_loop)), 0,
                               delta=self.float_tol)

        self.assertAlmostEqual(numpy.max(abs(coeffs_temp - coeffs_temp)), 0,
                               delta=self.float_tol)

        self.assertAlmostEqual(numpy.max(abs(coeffs_y - coeffs_y_loop)), 0,
                               delta=self.float_tol)

    def test_run_reaction_elim(self):
        """
        Purpose: Just run reaction_elim on a simple test case to make sure
        there are no syntax errors.

        Arguments:
        None

        Returns:
        None

        """
        chemReduce.reaction_elim([self.state], self.gas, self.atol, self.rtol)


    def test_run_reaction_and_species_elim(self):
        """
        Purpose: Just run reaction_and_species_elim on a simple test case
        to make sure there are no syntax errors.

        Arguments:
        None

        Returns:
        None

        """

        chemReduce.reaction_and_species_elim([self.state], self.gas,
                                             self.atol, self.rtol)
```

```
if __name__ == '__main__':
    unittest.main()
    #suite = unittest.TestSuite()
    #suite.addTest(TestCoeffIdentities('test_naive_summation'))
    #suite.addTest(TestCoeffIdentities('test_col_sums'))
    #suite.addTest(TestCoeffIdentities('test_row_sums'))
    #suite.addTest(TestCoeffIdentities('test_run_reaction_elim'))
    #suite.addTest(TestCoeffIdentities('test_run_reaction_and_species_elim'))
    #suite.debug()
```

Bibliography

[1] Clean Air Act. Technical report, Code of Laws of the United States, Title 42, Chapter 85, 1970.

[2] The Montreal Protocol on Substances that Deplete the Ozone Layer. Technical report, Ozone Secretariat United Nations Environment Programme, 1999.

[3] J. Å kesson, K.-E. Å rzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit. Modeling and optimization with Optimica and JModelica.orgLanguages and tools for solving large-scale dynamic optimization problems. *Computers & Chemical Engineering*, 34(11):1737–1749, November 2010.

[4] Joshua W Allen. PyDAS, 2010.

[5] Ioannis P. Androulakis. Kinetic mechanism reduction based on an integer programming approach. *AIChE Journal*, 46(2):361–371, February 2000.

[6] AC Antoulas and DC Sorensen. Projection methods for balanced model reduction. *Linear Algebra and Its Applications*, 2004.

[7] A.C. Antoulas and DC Sorensen. Approximation of large-scale dynamical systems: An overview. In *Large Scale Systems 2004: Theory and Applications (LSS'04): a Proceedings Volume from the 10th IFAC/IFORS/IMACS/IFIP Symposium, Osaka, Japan, 26-28 July 2004*, volume 11, page 19. Elsevier for the International Federation of Automatic Control, 2005.

[8] AC Antoulas, DC Sorensen, and S. Gugercin. A survey of model reduction methods for large-scale systems. In *Structured Matrices in Mathematics, Computer Science, and Engineering: Proceedings of an AMS-IMS-SIAM Joint Summer Research Conference, University of Colorado, Boulder, June 27-July 1, 1999*, page 193. Amer Mathematical Society, 2001.

[9] Athanasios Constantinos Antoulas. *Approximation of Large-Scale Dynamical Systems*. 2005.

[10] Michael Artin. *Algebra*. Prentice Hall, 1st edition, 1991.

[11] Uri Ascher and Linda Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM: Society for Industrial and Applied Mathematics, 1998.

[12] Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM: Society of Industrial and Applied Mathematics, 2000.

[13] A. Ben-Israel and T.N.E. Greville. *Generalized inverses: Theory and applications*. Springer Verlag, New York, 2nd edition, 2003.

[14] Gal Berkooz, Philip Holmes, and John L Lumley. The Proper Orthogonal Decomposition in the analysis of turbulent flows. *Annual Review of Fluid Mechanics*, 25:539–575, 1993.

[15] Luigi Carlo Berselli, Traian Iliescu, and William J Layton. *Mathematics of Large Eddy Simulation of Turbulent Flows*. Springer, 2010.

[16] Martin Berz and Kyoko Makino. Verified Integration of ODEs and Flows Using Differential Algebraic Methods on High-Order Taylor Models. *Reliable Computing*, 4:361–369, 1998.

[17] B Bhattacharjee, D A Schwer, P I Barton, and W H Green. Optimally-reduced kinetic models: reaction elimination in large-scale kinetic mechanisms. *Combustion and Flame*, 135:191–208, 2003.

[18] B. Bhattacharjee, D.A. Schwer, P.I. Barton, and W.H. Green. Optimally-reduced kinetic models: reaction elimination in large-scale kinetic mechanisms. *Combustion and Flame*, 135(3):191–208, 2003.

[19] M Bodenstein and H Lutkemeyer. The photochemical formation of hydrogen bromide and the formation rate of the bromine molecules from the atoms. *ZEITSCHRIFT FUR PHYSIKALISCHE CHEMIE–STOCHIOMETRIE UND*, 114(3/4):208–236, December 1924.

[20] B. Bond. *Parameterized model order reduction of nonlinear dynamical systems*. Master's thesis, Massachusetts Institute of Technology, 2006.

[21] J. R. Bowen, A. Acrivos, and A. K. Oppenheim. Singular perturbation refinement to quasi-steady state approximation in chemical kinetics. *Chemical Engineering Science*, 18(3):177–188, March 1963.

[22] R.B. Brad, A.S. Tomlin, M. Fairweather, and J.F. Griffiths. The application of chemical reduction methods to a combustion system exhibiting complex dynamics. *Proceedings of the Combustion Institute*, 31(1):455–463, 2007.

[23] T. Bui-Thanh, K. Willcox, and O. Ghattas. Model Reduction for Large-Scale Systems with High-Dimensional Parametric Input Space. *SIAM Journal on Scientific Computing*, 30(6):3270–3288, 2008.

[24] Y. Cao, S. Li, L. Petzold, and R. Serban. Adjoint Sensitivity Analysis for Differential-Algebraic Equations (Part 2). *SIAM Journal on Scientific Computing*, 24(3):1076–1099, 2003.

[25] Yang Cao and Linda Petzold. A posteriori error estimation and global error control for ordinary differential equations by the adjoint method. *SIAM Journal on Scientific Computing*, 26(2):359–374, 2004.

[26] Makis Caracotsios and Warren E. Stewart. Sensitivity analysis of initial value problems with mixed odes and algebraic equations. *Computers & Chemical Engineering*, 9(4):359–365, 1985.

[27] Kevin Carlberg, C. Bou-Mosleh, and Charbel Farhat. Efficient non-linear model reduction via a least-squares PetrovGalerkin projection and compressive tensor approximations. *International Journal for Numerical Methods in Engineering*, 86(2):155–181, 2011.

[28] D L Chapman and L K Underhill. The interaction of chlorine and hydrogen. The influence of mass. *Journal of the Chemical Society Transactions*, 103:496–508, 1913.

[29] S. Chaturantabut and Danny C Sorensen. A state space estimate for POD-DEIM Nonlinear Model Reduction. Technical report, 2010.

[30] S. Chaturantabut and D.C. Sorensen. Discrete empirical interpolation for nonlinear model reduction. In *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, volume 339, pages 4316–4321. Ieee, 2004.

[31] Saifon Chaturantabut and Danny C Sorensen. Nonlinear model reduction via discrete empirical interpolation. *SIAM J. Sci. Comput.*, 32(5):2737–2764, 2010.

[32] Saifon Chaturantabut and D.C. Sorensen. Application of POD and DEIM to Dimension Reduction of Nonlinear Miscible Viscous Fingering in Porous Media. *Math. Comput. Model. Dyn. Syst., to appear*, 2009.

[33] Jacqueline H. Chen. Petascale direct numerical simulation of turbulent combustionfundamental insights towards predictive models. *Proceedings of the Combustion Institute*, 33(1):99–123, 2011.

[34] E Chiavazzo, IV Karlin, and AN Gorban. Comparison of invariant manifolds for model reduction in chemical kinetics. *Communications in Computational Physics*, 2(5):964–992, October 2007.

[35] Yunfei Chu, Mitchell Serpas, and Juergen Hahn. State-preserving nonlinear model reduction procedure. *Chemical Engineering Science*, 66(17):3907–3913, September 2011.

[36] R. R. Coifman, I. G. Kevrekidis, S. Lafon, M. Maggioni, and B. Nadler. Diffusion maps, reduction coordinates, and low dimensional representation of stochastic systems. *Multiscale Model.*, 7(2):842–864, 2008.

[37] R. R. Coifman, S. Lafon, A. B. Lee, M. Maggioni, B. Nadler, F. Warner, and S. W. Zucker. Geometric diffusions as a tool for harmonic analysis and structure definition of data: diffusion maps. *Proceedings of the National Academy of Sciences of the United States of America*, 102(21):7426–31, May 2005.

[38] Germund Dahlquist. *Stability and Error Bounds in the Numerical Solution of Ordinary Differential Equations*. Phd thesis, Stockholm University, 1958.

[39] M J Davis and A S Tomlin. Spatial Dynamics of Steady Flames 2. Low-Dimensional Manifolds and the Role of Transport Processes. *J. Phys. Chem. A*, 112:7784–7805, 2008.

[40] Michael J Davis. Low-dimensional manifolds in reaction-diffusion equations: {2. Numerical} analysis and method development. *Journal of Physical Chemistry A*, 110:5257–5272, 2006.

[41] Michael J Davis and Rex T Skodje. Geometric investigation of low-dimensional manifolds in systems approaching equilibrium. *Journal of Chemical Physics*, 111(3):859–874, 1999.

[42] Michael J Davis and Alison S Tomlin. Spatial Dynamics of Steady Flames 1. Phase Space Structure and the Dynamics of Individual Trajectories. *J. Phys. Chem. A*, 112:7768–7783, 2008.

[43] MJ Davis. Low-dimensional manifolds in reaction-diffusion equations. 1. Fundamental aspects. *J. Phys. Chem. A*, 110(16):5235–5256, 2006.

[44] AP Davison. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. *Computing in Science and Engineering*, 2012.

[45] Jan de Leeuw. Reproducible Research: The Bottom Line. Technical report, UCLA Department of Statistics Paper 2001031101, 2001.

[46] R.P. Dickinson and R.J. Gelinas. Sensitivity Analysis of Ordinary Differential Equation Systems – A Direct Method. *Journal of Computational Physics*, 21(2):123–143, 1976.

[47] Kai Diethelm. The Limits of Reproducibility in Numerical Simulation. *Computing in Science & Engineering*, 14(1):64–72, January 2012.

[48] R. Djouad, B. Sportisse, and N. Audiffren. Reduction of multiphase atmospheric chemistry. *Journal of Atmospheric Chemistry*, 46(2):131–157, 2003.

[49] David L. Donoho and Carrie Grimes. Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data. *Proceedings of the National Academy of Sciences of the United States of America*, 100(10):5591–5596, 2003.

[50] David L. Donoho, Arian Maleki, Inam Ur Rahman, Morteza Shahram, and Victoria Stodden. Reproducible Research in Computational Harmonic Analysis. *Computing in Science & Engineering*, 11(1):8–18, January 2009.

[51] Eugene P. Dougherty and Herschel Rabitz. A computational algorithm for the Green's function method of sensitivity analysis in chemical kinetics. *International Journal of Chemical Kinetics*, 11(12):1237–1248, December 1979.

[52] C. H. Edwards. *Advanced Calculus of Several Variables*. Dover Publications, revised edition, 1995.

[53] K. Edwards, TF Edgar, and VI Manousiouthakis. Reaction mechanism simplification using mixed-integer nonlinear programming. *Computers and Chemical Engineering*, 24(1):67–79, 2000.

[54] D Estep, V Ginting, D Ropp, JN Shadid, and S Tavener. An A posteriori-A priori Analysis of Multiscale Operator Splitting. *SIAM Journal on Numerical Analysis*, 46(3):1116–1146, 2008.

[55] D Estep, V Ginting, and S Tavener. A posteriori Analysis of a Multirate Numerical Methods for Ordinary Differential Equations. Technical report, 2010.

[56] D Estep, S Tavener, and T Wildey. A posteriori Analysis and Improved Accuracy for an Operator Decomposition Solution of a Conjugate Heat Transfer Problem. *SIAM Journal on Numerical Analysis*, 46(4):2068–2089, 2008.

[57] Donald Estep. A Posteriori Error Bounds and Global Error Control for Approximation of Ordinary Differential Equations. *SIAM Journal on Numerical Analysis*, 32(1):1–48, 1995.

[58] Donald J Estep. Error Estimates for Multiscale Operator Decomposition For Multiphysics Models. In *Multiscale methods: bridging the scales in science and engineering*, pages 305–388. Oxford University Press, USA, 2009.

[59] P. Faucher. Isopycnal empirical orthogonal functions (EOFs) in the North and tropical Atlantic and their use in estimation problems. *Journal of Geophysical Research*, 107(C8):1–17, 2002.

[60] W.F. Feehery, J.E. Tolsma, and P.I. Barton. Efficient sensitivity analysis of large-scale differential-algebraic systems. *Applied Numerical Mathematics*, 25(1):41–54, October 1997.

[61] N Fenichel. Persistence and Smoothness of Invariant Manifolds for Flows. *Indiana University Mathematics Journal*, 21(3):193–226, 1971.

[62] N Fenichel. Asymptotic Stability with Rate Conditions. *Indiana University Mathematics Journal*, 23(12):1109–1137, 1974.

[63] N Fenichel. Geometric Singular Perturbation Theory for Ordinary Differential Equations. *Journal of Differential Equations*, 31:53–98, 1979.

[64] M. Fjeld, O. A. Asbjø rnsen, and K. J. Å ström. Reaction invariants and their importance in the analysis of eigenvectors, state observability and controllability of the continuous stirred tank reactor. *Chemical Engineering Science*, 29(9):1917–1926, September 1974.

[65] Sergey Fomel and Jon F Claerbout. Reproducible Research. *Computing in Science & Engineering*, pages 5–7, 2009.

[66] Simon J Fraser. Slow manifold for a bimolecular association mechanism. *Journal of Chemical Physics*, 120:3075–3085, 2004.

[67] S.J. Fraser. The steady state and equilibrium approximations: A geometrical picture. *The Journal of Chemical Physics*, 88(8):4732–4738, 1988.

[68] Juliana Freire and Claudio T Silva. Making Computations and Publications Reproducible with VisTrails. *Computing in Science and Engineering*, 2012.

[69] S. Gadewar, M. F. Doherty, and M. F. Malone. A systematic method for reaction invariants and mole balances for complex chemistries. *Computers & Chemical Engineering*, 25(9-10):1199–1217, September 2001.

[70] Eric Garnier, Nikolaus Adams, and Pierre Sagaut. *Large Eddy Simulation for Compressible Flows*. Springer, 2009.

[71] Robert Gentleman and Duncan Temple Lang. Statistical Analyses and Reproducible Research. *Journal of Computational and Graphical ...*, 2007.

[72] S. K. Godunov. A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics. *Matematicheskii Sbornik*, 47(89):271–306, 1959.

[73] D G Goodwin. An open-source, extensible software suite for CVD process simulation. *Chemical Vapor Deposition XVI and ...*, 98(40):10147, 2003.

[74] A Gorban and I V Karlin. Method of invariant manifold for chemical kinetics. *Chemical Engineering Science*, 58(21):4751–4768, 2003.

[75] Alexander N Gorban and Iliya V Karlin. Invariant grids for reaction kinetics. *Physica A: Statistical and Theoretical Physics*, 333:106–154, 2004.

[76] D.A. Goussis and M. Valorani. An efficient iterative algorithm for the approximation of the fast and slow dynamics of stiff systems. *Journal of Computational Physics*, 214(1):316–346, 2006.

[77] Thomas H Gronwall. Note on the derivatives with respect to a parameter of the solutions of a system of differential equations. *The Annals of Mathematics*, 20(4):292–296, 1919.

[78] John Guckenheimer and Philip Holmes. *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*. Springer, New York, 2002.

[79] B Haasdonk and M. Ohlberger. Efficient Reduced Models and A-Posteriori Error Estimation for Parametrized Dynamical Systems by Offline/Online Decomposition. *Mathematical and Computer Modelling of Dynamical Systems*, (1):1–17, 2011.

[80] Bernard Haasdonk and Mario Ohlberger. Reduced basis method for finite volume approximations of parametrized linear evolution equations. *ESAIM: Mathematical Modelling and Numerical Analysis*, 42(2):277–302, March 2008.

[81] Bernard Haasdonk and Mario Ohlberger. Efficient reduced models for parameterized dynamical systems by offline/online decomposition. In *Proc. MATHMOD 2009, 6th Vienna International Conference on Mathematical Modelling*, 2009.

[82] Bernard Haasdonk, Mario Ohlberger, and Gianluigi Rozza. A Reduced Basis Method for Evolution Schemes with Parameter-Dependent Explicit Operators. *Electronic Transactions on Numerical Analysis*, 32:145–161, 2008.

[83] Ernst Hairer, Syvert P Nø rsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, Berlin, Germany, second rev edition, 2000.

[84] Gary W. Harrison. Dynamic models with uncertain parameters. volume 1 of *Proceedings of the First International Conference on Mathematical Modeling*, pages 295–304, 1977.

[85] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, D A N E Shumaker, and Carol S Woodward. SUNDIALS : Suite of Nonlinear and Differential / Algebraic Equation Solvers. 31(3):363–396, 2005.

[86] C. Homescu, L.R. Petzold, and R. Serban. Error estimation for reduced-order models of dynamical systems. *SIAM Review*, 49(2):277–299, 2006.

[87] C. Homescu, L.R. Petzold, and R. Serban. Error estimation for reduced-order models of dynamical systems. *SIAM Journal on Numerical Analysis*, 43(4):1693–1714, 2006.

[88] Bill Howe. Virtual Appliances, Cloud Computing, and Reproducible Research. *Computing in Science & Engineering*, 14(4):36–41, July 2012.

[89] H. Huang, M. Fairweather, J.F. Griffiths, A.S. Tomlin, and R.B. Brad. A systematic lumping approach for the reduction of comprehensive kinetic models. *Proceedings of the Combustion Institute*, 30(1):1309–1316, January 2005.

[90] JD Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, pages 90–95, 2007.

[91] Barbara R. Jasny, Gilbert Chin, Lisa Chong, and Sacha Vignieri. Again, and Again, and Again.... *Science*, 334(December):2011, 2011.

[92] Christopher Jones. Geometric singular perturbation theory. In *Dynamical Systems, Lecture Notes in Mathematics,*, pages 44–118. 1995.

[93] Eric Jones, Travis Oliphant, Pearu Peterson, and Others. SciPy: Open Source Scientific Tools for Python, 2001.

[94] W Jones and S Rigopoulos. Rate-controlled constrained equilibrium: Formulation and application to nonpremixed laminar flames. *Combustion and Flame*, 142(3):223–234, 2005.

[95] H.G. Kaper and T.J. Kaper. Asymptotic analysis of two reduction methods for systems of chemical reactions. *Physica D: Nonlinear Phenomena*, 165(1-2):66–93, 2002.

[96] James C. Keck. Rate-controlled constrained-equilibrium theory of chemical reactions in complex systems. *Progress in Energy and Combustion Science*, 16(2):125–154, 1990.

[97] James C. Keck and David Gillespie. Rate-controlled partial-equilibrium method for treating reacting gas mixtures. *Combustion and Flame*, 17(2):237–241, October 1971.

[98] I G Kevrekidis, A E Deane, G E Karniadakis, and S A Orszag. Low-dimensional models for complex geometry flows: Application to grooved channels and circular cylinders. *Physics of Fluids A*, 3(10):2337–2354, 1991.

[99] David J Knezevic. Reduced Basis approximation and a posteriori error estimates for a Multiscale Liquid Crystal Model. *Mathematical and Computer Modelling of Dynamical Systems*, 2010.

[100] Mark a. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE Journal*, 37(2):233–243, February 1991.

[101] K Kunisch and S Volkwein. Control of the Burgers equation by a reduced-order approach using proper orthogonal decomposition. *Journal of Optimization Theory and Applications*, 102(2):345–371, 1999.

[102] K Kunisch and S Volkwein. Galerkin Proper Orthogonal Decomposition Methods for a General Equation in Fluid Dynamics. *SIAM Journal on Numerical Analysis*, 40:492–515, 2002.

[103] S H Lam. Using CSP to understand complex chemical kinetics. *Combustion Science and Technology*, 89(5):375–404, 1993.

[104] SH Lam and DA Goussis. The CSP method for simplifying kinetics. *International Journal of Chemical Kinetics*, 26(4):461–486, 1994.

[105] J. Lang and J G Verwer. On Global Error Estimation and Control for Initial Value Problems. *SIAM Journal on Scientific Computing*, 27(4):21, 2007.

[106] Oliver F. Lange and Helmut Grubmüller. Full correlation analysis of conformational protein dynamics. *Proteins: Structure, Function, and Bioinformatics*, pages 1294–1312, 2007.

[107] JC Lee, HN Najm, S. Lefantzi, J. Ray, M. Frenklach, M. Valorani, and DA Goussis. A CSP and tabulation-based adaptive chemistry model. *Combustion Theory and Modelling*, 11(1):73–102, 2007.

[108] J.M. Lee. *Introduction to smooth manifolds*. Springer Verlag, New York, 2003.

[109] Patrick A Legresley and Juan J Alonso. Dynamic Domain Decomposition and Error Correction for Reduced Order Models. In *41st Aerospace Sciences Meeting and Exhibit*, 2003.

[110] Randall J LeVeque, Ian M Mitchell, and Victoria Stodden. Reproducible Research for Scientific Computing: Tools and Strategies for Changing the Culture. *Computing in Science & ...*, pages 13–17, 2012.

[111] RJ LeVeque. Python Tools for Reproducible Research on Hyperbolic Problems. *Computing in Science & Engineering*, pages 19–27, 2009.

[112] G. Li. A lumping analysis in mono- or/and bimolecular reaction systems. *Chemical Engineering Science*, 39(7-8):1261–1270, 1984.

[113] G. Li and H. Rabitz. A General Analysis of Exact Lumping in Chemical Kinetics. *Chemical engineering science*, 44(6):1413–1430, 1989.

[114] G. Li and H. Rabitz. A General Analysis of Approximate Lumping in Chemical Kinetics. *Chemical engineering science*, 45(4):977–1002, 1990.

[115] G. Li and H. Rabitz. A general lumping analysis of a reaction system coupled with diffusion. *Chemical engineering science*, 46(8):2041–2053, 1991.

[116] G. Li and H. Rabitz. Determination of constrained lumping schemes for nonisothermal first-order reaction systems. *Chemical Engineering Science*, 46(2):583–596, 1991.

[117] G. Li and H. Rabitz. New approaches to determination of constrained lumping schemes for a reaction system in the whole composition space. *Chemical Engineering Science*, 46(1):95–111, 1991.

[118] Y Lin and M Stadtherr. Validated solutions of initial value problems for parametric ODEs. *Applied Numerical Mathematics*, 57(10):1145–1162, October 2007.

[119] T. Lø vås, P. Amnéus, F. Mauss, and E. Mastorakos. Comparison of automatic reduction procedures for ignition chemistry. *Proceedings of the Combustion Institute*, 29(1):1387–1393, 2002.

[120] Michel Loève. *Probability Theory.* Van Nostrand, 1955.

[121] T Lu and C Law. On the applicability of directed relation graphs to the reduction of reaction mechanisms. *Combustion and Flame*, 146(3):472–483, 2006.

[122] T Lu and C Law. A criterion based on computational singular perturbation for the identification of quasi steady state species: A reduced mechanism for methane oxidation with NO chemistry. *Combustion and Flame*, 154(4):761–774, 2008.

[123] T Lu and C Law. Strategies for mechanism reduction for large hydrocarbons: n-heptane. *Combustion and Flame*, 154(1-2):153–163, 2008.

[124] T. Lu and C.K. Law. Systematic approach to obtain analytic solutions of quasi steady state species in reduced mechanisms. *Journal of Physical Chemistry A*, 110(49):13202–13208, December 2006.

[125] T. Lu and C.K. Law. Toward accommodating realistic fuel chemistry in large-scale computations. *Progress in Energy and Combustion Science*, 35(2):192–215, 2009.

[126] Tianfeng Lu and Chung K. Law. A directed relation graph method for mechanism reduction. *Proceedings of the Combustion Institute*, 30(1):1333–1341, January 2005.

[127] Tianfeng Lu and Chung K. Law. Linear time reduction of large kinetic mechanisms with directed relation graph: n-Heptane and iso-octane. *Combustion and Flame*, 144(1-2):24–36, January 2006.

[128] D.J. Lucia, P.I. King, P.S. Beran, and M.E. Oxley. Reduced order modeling for a one-dimensional nozzle flow with moving shocks. In *15th AIAA Computational Fluid Dynamics Conference*, volume paper, 2001.

[129] X. Ma and G.E. Karniadakis. A low-dimensional model for simulating three-dimensional cylinder flow. *Journal of Fluid Mechanics*, 458:181–190, 2002.

[130] Ulrich Maas and Steven B Pope. Simplifying chemical kinetics: Intrinsic low-dimensional manifolds in chemical composition space. *Combustion and Flame*, 88:239–264, 1992.

[131] T. Maly and L.R. Petzold. Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Applied Numerical Mathematics*, 20(1-2):57–79, August 1996.

[132] S Margolis. Time-dependent solution of a premixed laminar flame. *Journal of Computational Physics*, 27(3):410–427, June 1978.

[133] MATLAB. *Version 7.14.0 (R2012a)*. The MathWorks Inc., Natick, Massachusetts, 2012.

[134] Jill P Mesirov. Accessible Reproducible Research. *Science*, 327(January):415–416, 2010.

[135] Stuart Mitchell, Stuart Mitchell Consulting, Michael O Sullivan, and Iain Dunning. PuLP : A Linear Programming Toolkit for Python. 2011.

[136] A M Mitsos. Alternative Formulation for Species Elimination. Technical report, RWTH Aachen, Aachen, Germany, 2008.

[137] Alexander Mitsos, Geoffrey M Oxberry, Paul I Barton, and William H Green. Optimal automatic reaction and species elimination in kinetic mechanisms. *Combustion and Flame*, 155:118–132, 2008.

[138] Parviz Moin and Krishnan Mahesh. Direct Numerical Simulation: A Tool in Turbulence Research. *Annual Review of Fluid Mechanics*, 30(1):539–578, January 1998.

[139] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society of Industrial and Applied Mathematics, 2009.

[140] R.E. Moore. *Methods and applications of interval analysis*. Society for Industrial Mathematics, Philadelphia, 1987.

[141] K R Müller, S Mika, G Rätsch, K Tsuda, and B Schölkopf. An introduction to kernel-based learning algorithms. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 12(2):181–201, January 2001.

[142] J Munkres. *Analysis on Manifolds*. Westview Press, 1st edition, 1990.

[143] B Nadler, S Lafon, R Coifman, and I Kevrekidis. Diffusion maps, spectral clustering and reaction coordinates of dynamical systems. *Applied and Computational Harmonic Analysis*, 21(1):113–127, July 2006.

[144] J. Nafe and U. Maas. A general algorithm for improving ILDMs. *Combustion Theory and Modelling*, 6(4):697–709, 2002.

[145] T. Nagy and T. Turányi. Reduction of very large reaction mechanisms using methods based on simulation error minimization. *Combustion and Flame*, 156(2):417–428, February 2009.

[146] M Neher, KR Jackson, and NS Nedialkov. On Taylor Model Based Integration of ODEs. *SIAM Journal on Numerical . . .* , 45(1):236–262, 2007.

[147] Arnold Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 2008.

[148] N Nguyen. A posteriori error estimation and basis adaptivity for reduced-basis approximation of nonaffine-parametrized linear elliptic partial differential equations. *Journal of Computational Physics*, 227(2):983–1006, December 2007.

[149] PH Nguyen. Complexity of Free Energy Landscapes of Peptides Revealed by Nonlinear Principal Component Analysis. *Proteins: Structure, Function, and Bioinformatics*, 913(October):898–913, 2006.

[150] Regents of the University of California. The BSD 3-Clause License. Web site.

[151] Miles S. Okino and Michael L. Mavrovouniotis. Simplification of Mathematical Models of Chemical Reaction Systems. *Chemical Reviews*, 98(2):391–408, April 1998.

[152] Travis E. Oliphant. Python for Scientific Computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[153] O O Oluwole, P I Barton, and W H Green. Obtaining accurate solutions using reduced chemical kinetic models: a new model reduction method for models rigorously validated over ranges. *Combustion Theory and Modelling*, 11(1):127–146, 2007.

[154] O O Oluwole, B Bhattacharjee, J E Tolsma, P I Barton, and W H Green. Rigorous valid ranges for optimally reduced kinetic models. *Combustion and Flame*, 146(1-2):348–365, 2006.

[155] Geoffrey M. Oxberry. State-space error bounds for projection-based reduced model ODEs. 2012.

[156] Geoffrey M. Oxberry, Paul I. Barton, and William H. Green. Projection-based model reduction in combustion. 2012.

[157] Roger Peng. Reproducible Research in Computational Science. *Science*, 334(December):1226–1227, January 2011.

[158] Roger D. Peng and Sandrah P. Eckel. Distributed Reproducible Research Using Cached Computations. *Computing in Science & Engineering*, 11(1):28–34, January 2009.

[159] P. Pepiot-Desjardins and H. Pitsch. An efficient error-propagation-based reduction method for large chemical kinetic mechanisms. *Combustion and Flame*, 154(1-2):67–81, July 2008.

[160] Linda Petzold and Wenjie Zhu. Model Reduction for Chemical Kinetics: An Optimization Approach. *AIChE Journal*, 45(4):869–886, 1999.

[161] Heinz Pitsch. Large-Eddy Simulation of Turbulent Combustion. *Annual Review of Fluid Mechanics*, 38(1):453–482, January 2006.

[162] Thierry Poinsot, Sebastien Candel, and Arnaud Trouve. Applications of direct numerical simulation to premixed turbulent combustion. *Progress in Energy and Combustion ...*, 21(95):531–576, 1995.

[163] N. Ramdani, N. Meslem, and Y. Candau. A Hybrid Bounding Method for Computing an Over-Approximation for the Reachable Set of Uncertain Nonlinear Systems. *IEEE Transactions on Automatic Control*, 54(10):2352–2364, October 2009.

[164] A Rapaport and D Dochain. Interval observers for biochemical processes with uncertain kinetics and inputs. *Mathematical Biosciences*, 193(2):235–253, February 2005.

[165] M. Rathinam and L.R. Petzold. A new look at proper orthogonal decomposition. *SIAM Journal on Numerical Analysis*, 41(5):1893–1925, 2004.

[166] Z Ren, S B Pope, A Vladimirsky, and J M Guckenheimer. The invariant constrained equilibrium edge preimage curve method for the dimension reduction of chemical kinetics. *Journal of Chemical Physics*, 124(11):114111(1–10), 2006.

[167] Z. Ren and S.B. Pope. The use of slow manifolds in reactive flows. *Combustion and Flame*, 147(4):243–261, 2006.

[168] Z. Ren and S.B. Pope. Reduced description of complex dynamics in reactive systems. *J. Phys. Chem. A*, 111(34):8464–8474, 2007.

[169] M. Rewienski and J. White. A trajectory piecewise-linear approach to model order reduction and fast simulation of nonlinear circuits and micromachined devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(2):155–170, February 2003.

[170] M Rewienski and J White. Model order reduction for nonlinear dynamical systems based on trajectory piecewise-linear approximations. *Linear Algebra and its Applications*, 415(2-3):426–454, June 2006.

[171] C. Rhodes, M. Morari, and S. Wiggins. Identification of low order manifolds: Validating the algorithm of Maas and Pope. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 9(1):108–123, 1999.

[172] W. Richardson, L. Volk, KH Lau, SH Lin, and H. Eyring. Application of the singular perturbation method to reaction kinetics. *Proceedings of the National Academy of Sciences*, 70(5):1588–1592, 1973.

[173] Anthony Rossini and Friedrich Leisch. Literate Statistical Practice. Technical Report March 2003, University of Washington Biostatistics Working Paper Series 194, Seattle, WA, USA, 2003.

[174] Marc R Roussel and Simon J Fraser. Geometry of the steady-state approximation: Perturbation and accelerated convergence methods. *Journal of Chemical Physics*, 93(2):1072–1081, 1990.

[175] Marc R Roussel and Simon J Fraser. On the geometry of transient relaxation. *Journal of Chemical Physics*, 94(11):7106–7113, 1991.

[176] M.R. Roussel and S.J. Fraser. Invariant manifold methods for metabolic model reduction. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 11(1):196–206, 2001.

[177] S. T. Roweis. Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, 290(5500):2323–2326, December 2000.

[178] G. Rozza, D.B.P. Huynh, and A.T. Patera. Reduced basis approximation and a posteriori error estimation for affinely parameterized elliptic coercive partial differential equations. *Archives of Computational Methods in Engineering*, 15:229–275, 2008.

[179] Yousef Saad. *Iterative Methods for Sparse Linear Systems Second Edition*. SIAM: Society for Industrial and Applied Mathematics, 2003.

[180] Pierre Sagaut. *Large Eddy Simulation for Incompressible Flows: An Introduction*. Springer, third edition, 2005.

[181] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear Component Analysis as a Kernel Eigenvalue Problem. *Neural Computation*, 10(5):1299–1319, July 1998.

[182] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. Making scientific computations reproducible. *Computing in Science & . . .*, 2(6):61–67, 2000.

[183] D.A. Schwer, J.E. Tolsma, W.H. Green, and P.I. Barton. On upgrading the numerics in combustion chemistry codes. *Combustion and Flame*, 128(3):270–291, 2002.

[184] Douglas A Schwer, Pisi Lu, William H Green, and Viriato Semiao. A consistent-splitting approach to computing stiff steady-state reacting flows with adaptive chemistry. *Combustion Theory and Modelling*, 7:383–399, 2003.

[185] Radu Serban, Chris Homescu, and Linda R Petzold. The effect of problem perturbations on nonlinear dynamical systems and their reduced-order models. *SIAM J. Sci. Comput.*, 29(6):2621–2643, 2007.

[186] Valeria Simoncini and Daniel B Szyld. Interpreting IDR as a Petrov-Galerkin Method. *Society*, 32(4):1898–1912, 2010.

[187] Adam B. Singer and Paul I. Barton. Bounding the Solutions of Parameter Dependent Nonlinear Ordinary Differential Equations. *SIAM Journal on Scientific Computing*, 27(6):2167, 2006.

[188] M.A. Singer and W.H. Green. Using adaptive proper orthogonal decomposition to solve the reactiondiffusion equation. *Applied Numerical Mathematics*, 59(2):272–279, 2009.

[189] S Singh, J M Powers, and S Paolucci. On slow manifolds of chemically reactive systems. *The Journal of Chemical Physics*, 117(4):1482–1496, 2002.

[190] R.D. Skeel. Thirteen Ways to Estimate Global Error. *Numerische Mathematik*, 48(1):1–20, 1986.

[191] Rex T Skodje and Michael J Davis. Geometrical Simplification of Complex Kinetic Systems. *Journal of Physical Chemistry A*, 105:10356–10365, 2001.

[192] Gustaf Söderlind. The logarithmic norm. History and modern theory. *BIT Numerical Mathematics*, 46(3):631–652, August 2006.

[193] B. Sportisse and R. Djouad. Use of proper orthogonal decomposition for the reduction of atmospheric chemical kinetics. *Journal of Geophysical Research-Atmospheres*, 112(D6):D06303, 2007.

[194] B Srinivasan, M Amrhein, and D Bonvin. Reaction and Flow Variants/Invariants in Chemical Reaction Systems with Inlet and Outlet Streams. *AIChE Journal*, 44(8):1858–1867, 1998.

[195] GW Stewart. On the numerical analysis of oblique projectors. *SIAM Journal on Matrix Analysis and Applications*, 32(1):309–348, 2011.

[196] Victoria Stodden. The Legal Framework for Reproducible Scientific Research: Licensing and Copyright. *Computing in Science & Engineering*, 11(1):35–40, January 2009.

[197] Victoria Stodden. Reproducible Research: Tools and Strategies for Scientific Computing. *Computing in Science & Engineering*, pages 11–12, 2012.

[198] Victoria Stodden, David Donoho, Sergey Fomel, Michael Friedlander, Mark Gerstein, Randall J LeVeque, Ian Mitchell, Lisa Larrimore Ouellette, Chris Wiggins, Nicholas W Bramble, Patrick O Brown, Vincent J Carey, Laura DeNardis, Robert Gentleman, J Daniel Gezelter, Alyssa Goodman, Matthew G Knepley, Joy E Moore, Frank A Pasquale, Joshua Rolnick, Michael Seringhaus, and Ramesh Subramanian. Reproducible Research: Addressing the need for data and code sharing in computational science. *Journal of Computing Science and Engineering*, 2010.

[199] G. Strang. On the construction and comparison of difference schemes. *SIAM Journal on Numerical Analysis*, 5(3):506–517, 1968.

[200] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley Cambridge Press, 4th edition, 2009.

[201] H F Stripling, M Anitescu, and M L Adams. A Generalized Adjoint Framework for Sensitivity and Global Error Estimation in Time-Dependent Nuclear Reactor Simulations. Technical Report 979, Preprint ANL/MCS-P1963-1011, Mathematics and Computer Science Division, Argonne National Laboratory, 2011.

[202] J B Tenenbaum, V de Silva, and J C Langford. A global geometric framework for nonlinear dimensionality reduction. *Science (New York, N.Y.)*, 290(5500):2319–23, December 2000.

[203] Alison S Tomlin, Tamás Turányi, and Michael J Pilling. Mathematical tools for the construction, investigation and reduction of combustion mechanisms. In *Comprehensive Chemical Kinetics, Volume 35: Low-Temperature Combustion and Autoignition*, chapter 4, pages 293–437. Elsevier, 1997.

[204] J. Toth, G. Li, H. Rabitz, and A.S. Tomlin. The effect of lumping and expanding on kinetic differential equations. *SIAM Journal on Applied Mathematics*, 57(6):1531–1556, 1997.

[205] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[206] M. Valorani, H.N. Najm, and D.A. Goussis. CSP analysis of a transient flame-vortex interaction: time scales and manifolds. *Combustion and Flame*, 134(1-2):35–53, 2003.

[207] Mauro Valorani and Dimitris A Goussis. Explicit Time-Scale Splitting Algorithm for Stiff Problems: Auto-ignition of Gaseous Mixtures behind a Steady Shock. *Journal of Computational Physics*, 169:44–79, 2001.

[208] Mauro Valorani, Dimitris A Goussis, Francesco Creta, and Habib N Najm. Higher order corrections in the approximation of low-dimensional manifolds and the construction of simplified problems with the CSP method. *Journal of Computational Physics*, 209:754–786, 2005.

[209] Guido van Rossum. The Python Programming Language, 1991.

[210] Ioan Vlad. Reproducibility in computer-intensive sciences. *Ad Astra*, 1(2):1–2, 2002.

[211] K.V. Waller and P.M. Makila. Chemical reaction invariants and variants and their use in reactor modeling, simulation, and control. *Industrial & Engineering Chemistry Process Design and Development*, 20(1):1–11, 1981.

[212] H Wang and M Frenklach. Detailed reduction of reaction mechanisms for flame modeling. *Combustion and Flame*, 87(3-4):365–370, December 1991.

[213] J. Wei and J.C.W. Kuo. A lumping analysis in monomolecular reaction systems. *Industrial Engineering and Chemistry Fundamentals*, 8(1):114–123, 1969.

[214] Kilian Q. Weinberger and Lawrence K. Saul. Unsupervised Learning of Image Manifolds by Semidefinite Programming. *International Journal of Computer Vision*, 70(1):77–90, May 2006.

[215] Charles K. Westbrook, Yasuhiro Mizobuchi, Thierry J. Poinsot, Phillip J. Smith, and Jürgen Warnatz. Computational combustion. *Proceedings of the Combustion Institute*, 30(1):125–157, January 2005.

[216] Rowand Wilson and OSS Watch. The Modified BSD License – An Overview. Web site, May 2012.

[217] D. Wirtz and Bernard Haasdonk. Efficient a-posteriori error estimation for nonlinear kernel-based reduced systems. *Outlook*, 2(2010):48, 2011.

[218] Pedro E. Zadunaisky. On the estimation of errors propagated in the numerical integration of ordinary differential equations. *Numerische Mathematik*, 71(1):20–39, March 1976.

[219] A. Zagaris, HG Kaper, and TJ Kaper. Analysis of Computational Singular Perturbation Reduction Method for Chemical Kinetics. *Journal of Nonlinear Science*, 14(1):59–91, 2004.

[220] A. Zagaris, H.G. Kaper, and T.J. Kaper. Fast and slow dynamics for the computational singular perturbation method. *Multiscale Model. Simul.*, 2(4):613–638, 2004.

[221] X.L. Zheng, T.F. Lu, and C.K. Law. Experimental counterflow ignition temperatures and reaction mechanisms of 1,3-butadiene. *Proceedings of the Combustion Institute*, 31(1):367–375, January 2007.

[222] OC Zienkiewicz, RL Taylor, and P Nithiarasu. *The Finite Element Method for Fluid Dynamics*. Butterworth-Heinemann, sixth edition, 2005.