ROBUST DESIGN OPTIMIZATION WITH DYNAMIC CONSTRAINTS USING NUMERICAL CONTINUATION

BY

J. COLE ANDERSON

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Mechanical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Harry Dankowicz

Abstract

This thesis develops a framework for performing robust design optimization of objective functions constrained by differential, algebraic, and integral constraints. A successive parameter continuation method combined with polynomial chaos expansions is used to locate stationary points. The use of such an expansion provides the benefit of being able to directly drive the mean and variance of a given response function (or an objective function that uses them) during continuation. A toolbox capable of constructing polynomial chaos expansions for system response functions evaluated on boundary value problems has been developed for this work. Its use is demonstrated and results are compared to analytically derived solutions of a linear, harmonically forced oscillator. The robust design optimization method is then applied a harmonically forced nonlinear oscillator.

To Danielle, Ian, and Eli

Acknowledgments

This has been a journey requiring both stamina and motivation, not always in easy supply. Though it at times felt solitary, now that it has come to an end it is obvious that there are so many people to thank.

First, I'd like to thank my employer Caterpillar for supporting my pursuit of this degree. Specifically thank you to my supervisors during this time, Mike Henry and Andrew Yun, for providing me the flexibility to take classes and attend on-campus meetings during normal work hours and for providing me encouragement during the process.

Thank you to my degree supervisor, Professor Harry Dankowicz. You offered excellent guidance throughout my time, opened my eyes to the interesting subject of nonlinear dynamics, and allowed me the flexibility to weave in the uncertainty quantification elements that interested me. Most importantly you were very understanding when it came to my work and family commitments. Without your patience in that regard, this degree would absolutely not have been possible.

Thank you to my lab mates. I did not spend as much time in the lab as a more traditional student, but when I was there I never felt like 'the old guy'. I wish you all the best in your academic and professional pursuits. A special thanks to Mingwu Li for all of the help you provided me with COCO and for helping me work through various problems that came up in my thesis project.

Thank you to my friends and family for all of the support you've provided. I'm looking forward to spending more time with all of you again!

To my boys, Ian and Eli, there were many times the past few years that I regretted missing time with you, but now 'thesis' is over. Hopefully at some point there will be some lesson about working hard to achieve a goal, but in the meantime... let's go to Legoland!

Finally, and most importantly, to my wife Danielle. You are the foundation that made this possible. You've juggled so many things, taken on so many responsibilities I could not keep up with, and lived effectively as a single parent for long stretches. Thank you so much for your encouragement and your belief in me. You're the best and I'm lucky to have you.

Table of Contents

Chapter 1

Introduction

Uncertainty is an ever-present reality in physical systems. Attempts to capture the effects of this reality in mathematical models is referred to as uncertainty quantification. The sources of uncertainty in a model are, of course, extremely varied. For example, if the model characterizes behavior of a product in a high-volume manufacturing environment, then the exact value of a given model parameter (e.g., material property, damping coefficient, material thickness, etc.) cannot be known for each realization of the product (at least not within typical cost constraints). Even if all parameters inherent to a design are known exactly, there still exists uncertainty in its eventual application. Examples include seismic loading of a structure [1] or underfoot conditions of an off-road vehicle [2].

It is a typical goal of engineering problems to seek solutions that are optimal in some sense. Examples include using the least amount of fuel to get from point A to point B or designing the strongest structure for a given weight restriction. This thesis will concern itself with *robust constrained* design optimization, which seeks optimal solutions in the presence of uncertainty, as well as equality constraints that must be satisfied by the design variables. It describes a method for solving such optimization problems that combines a continuation method for satisfying constraints and polynomial chaos expansions for approximating statistical moments.

The remainder of Chapter 1 covers preliminary material and terminology used throughout the thesis and is organized as follows. Section 1 includes a brief overview of the concepts of optimization, uncertainty quantification, and optimization under uncertainty. Section 2 discusses numerical and computational methods used in the thesis. Section 3 provides an outline of the remaining chapters of the thesis.

1.1 Optimization under Uncertainty

1.1.1 Optimization

The goal of an optimization problem is to seek extremal values (maxima or minima) of an objective function $f : X \to \mathbb{R}$ where X is the design variable space. Considering only the case of minimization (and noting that maximization of f is merely the minimization of the related objective function $f^* = -f$). An optimal solution $x^* \in S$ on a set $S \subseteq X$ satisfies the condition

$$f(x^*) \leq f(x) \, \forall x \in S. \tag{1.1}$$

The case where $S = X$ indicates that there are no restrictions on the possible solutions and corresponds to a problem of *unconstrained optimization*. The optimization problems of interest in this thesis, however, are described by a theory of *constrained optimization* since the selection of x is limited to a proper subset of X through the enforcement of constraints. Points in S are referred to as feasible points.

In addition to the constrained vs. unconstrained classification, optima can also be classified as *global* or *local*. Global optima satisfy equation 1.1 for all x in S. A local optimum, on the other hand, satisfies equation 1.1 only a sufficiently small subset U of S containing x^*. In this thesis, locally optimal solutions in the presence of constraints will be sought using the successive parameter continuation method outlined in Chapter 2.

Constrained Optimization

As mentioned above, constrained optimization limits the set of feasible points to those which satisfy one or more constraint relations that have been applied to the variables describing a problem. Examples of constraints in an engineering context would be a restriction on the cost of a product, enforcement of a physical law that relates two quantities, or a requirement that a stress value remain below a failure threshold. As these descriptions imply, constraints can be either equality constraints or inequality constraints. A typical equality constrained optimization problem is given by

$$\min_x f(x), \quad \text{s.t. } g(x) = 0 \tag{1.2}$$

with objective function f and constraint function $g : X \to \mathbb{R}^m$. The method of Lagrange multipliers can be used for solving these types of problems by combining the objective function and constraints into a single Lagrangian

$$\mathcal{L}(x, \lambda) = f(x) - \lambda g(x), \tag{1.3}$$

where $\lambda \in \mathbb{R}^m$ are Lagrange multipliers for the m constraint equations in g. In the special case that $X = \mathbb{R}^n$, the first order necessary conditions for an optimal solution are then obtained by differentiating $\mathcal{L}(x, \lambda)$ with respect to both x and λ and setting the results equal to 0:

$$\nabla_x \mathcal{L}(x,\lambda) = f_{,x}(x) - \lambda g_{,x}(x) = 0, \tag{1.4}$$

$$\nabla_\lambda \mathcal{L}(x,\lambda) = g(x) = 0. \tag{1.5}$$

A critical point occurs at values of x and λ that satisfy these $n+m$ equations. It is worth mentioning that critical points for inequality constrained optimization problems must satisfy the more general Karush-Kuhn-Tucker (KKT) conditions [3]. However, this thesis will focus only on equality constrained problems, specifically those constrained by boundary-value problems as well integral and algebraic conditions.

1.1.2 Robust Design

Traditional means of dealing with uncertainty come in the form of engineering safety factors or worst case design. A safety factor is a hedge against modeling errors arising either from the parametric uncertainty discussed above, from unmodeled phenomena, or from any other possible source. Worst case design assumes bounds on some sources of uncertainty and then designs for the case that most adversely affects a design's performance. For parametric uncertainty in a manufacturing context the worst case limits could be enforced through an end-of-line inspection process. These methods are more conservative and while they can provide higher certainty of product success, they typically come at the expense of additional cost or reduced performance.

More modern techniques for designing around uncertainties take advantage of statistical methods and probability. Early statistical methods for evaluating a design's robustness include Design of Experiments (DOE) and Taguchi's extensions of those techniques. In general, Taguchi's methods seek to systematically explore the design space to highlight sources of variation and, if possible, eliminate them. If elimination of such sources is not possible, then modifications to the design are sought to reduce the effect of variation on the final design. An overview of some technical aspects of the Taguchi method as well as other methods of robust design are provided in [4].

It should be noted that while the early focus of Taguchi's methods dealt with actual physical experiments, it is common now for the experiment to be a numerical simulation of a physical phenomenon [5]. This has the advantage of exploring the design space in a faster, more affordable way. A reduced set of physical experiments can still be performed to validate the results of the numerical simulation.

1.1.3 Robust Design Optimization

With the knowledge that uncertainty is unavoidable, it is natural to seek optimal solutions that account for uncertainty. This is referred to as Optimization Under Uncertainty (OUU). One popular approach to OUU is called Robust Design Optimization (RDO) [6]. A typical formulation of an RDO problem is given by

$$\begin{aligned} \min_{d} \quad & F\left(\mu_f\left(d,p\right), \sigma_f^2\left(d,p\right)\right), \\ \text{s.t} \quad & G_i\left(\mu_{g_i}\left(d,p\right), \sigma_{g_i}^2\left(d,p\right)\right) \leq 0,\ i = 1, 2 \ldots m, \\ & Pr\left[d_{i,min} \leq d_i \leq d_{i,max}\right] \geq P_i,\ i = 1, 2, \ldots n_d, \end{aligned}$$

where $d \in \mathbb{R}^{n_d}$ is a vector of decision variables able to be set by the designer, and $p \in \mathbb{R}^{n_p}$ are additional problem parameters (possibly random variables) out of the designer's control. Because of the stochastic nature of the inputs, the objective function f is a random variable and the *robust* objective function F is a function of the mean μ_f and variance σ_f^2 of the objective function.

The robust constraint functions G_i depend on the statistical moments of the original constraint functions g_i (also random variables). Methods for handling robust constraints of the form of G_i will be discussed as part of this thesis for the equality constrained case. The last set of constraints in the problem description impose probabilistic allowable bounds on the decision variables. Constraints of this type will not be considered in this thesis.

1.2 Numerical and Computational Methods

1.2.1 Polynomial Chaos Expansion

It is clear that to solve an RDO problem, statistical moments of an objective function (and possibly constraints) must be calculated. In the absence of a statistical distribution for the objective function, calculating the moments requires numerical approximation. Taylor expansions are one option [4], but they are not well-suited to highly nonlinear problems or problems with large variability in the inputs [5]. Monte Carlo simulation is another option, but this converges to the true values of the statistical moments at a rate of $O\left(1/\sqrt{n}\right)$ for sample size n [7]. The computational expense can be prohibitive for complicated numerical models.

In this thesis, the method of Polynomial Chaos Expansion (PCE) [8] will be used. PCE considers a function of random variable(s) (called a response function), which is assumed to have finite variance, and writes this in terms of an expansion in orthogonal polynomials. Let the function r depend on a set of deterministic variables p and a set of random variables ξ. Then, a PCE expansion is given by

$$r\left(p,\xi\right)=\sum_{i=0}^{\infty}\alpha_i\left(p\right)\Psi_i\left(\xi\right). \tag{1.6}$$

The α_i's are expansion coefficients that depend only on the deterministic variables, while the Ψ_i's are orthogonal polynomial basis functions that depend only on the random variables.

One advantage of approximating statistical moments using PCE is that for properly chosen basis functions, the error convergence is exponential [8]. This typically means fewer evaluations of a numerical model relative to other methods like Monte Carlo simulation. Another advantage is that the statistical moments of the expansion have simple closed forms that make it possible to incorporate them into the successive parameter continuation method outlined in Chapter 2.

Drawbacks to PCE arise when the number of random variables becomes large. Implementations like the one outlined in Chapter 3 require exponentially increasing numbers of evaluations for higher stochastic dimension. Another issue is that if the coefficients are time-varying, the convergence rate of the expansion can degrade over long time intervals [9, 10]. There are methods to mitigate (though not eliminate) both of these concerns. These include sparse quadrature methods [11] and structured sampling regression methods [12] for higher dimensionality, and adaptive methods in time for long durations of integration [13, 10].

A more thorough introduction to PCE will be provided in Chapter 3. It will be formulated in a way that allows for solutions to robust design optimization to be performed with the successive parameter continuation optimization method discussed in Chapter 2.

1.2.2 Quadrature

Successful implementation of a PCE routine relies on the selection of appropriate numerical integration rules and corresponding orthogonal polynomial basis functions to maintain optimal convergence rates of the expansions [8], so some discussion of these rules is warranted. Quadrature rules approximate an integral as the weighted sum of evaluations of the integrand:

$$\int_a^b g\left(x\right)dx=\int_a^b \omega\left(x\right)f\left(x\right)dx\approx\sum_{i=1}^{n}w_i f\left(x_i\right). \tag{1.7}$$

The integrand g in (1.7) has been factored into a weight function $\omega\left(x\right)$ (possibly equal to 1) and a function f. The optimally accurate choice of nodes x_i and weights w_i for a numerical quadrature rule is dependent on the form of the weight function.

Gaussian quadrature rules approximate an integrand as a polynomial function (though the interpolant is not explicitly constructed). A Gaussian quadrature rule is characterized by a choice of nodes and weights such that an n-th order rule can exactly integrate a polynomial function of order $2n-1$. Gauss rules offer the maximal order and highest accuracy for a given number of integration nodes [7].

The formulation of Gaussian quadrature rules is closely tied to the theory of orthogonal polynomials and node-weight schemes are typically named for the polynomial set to which they are associated [14]. For example, Legendre polynomials are orthogonal with respect to the weight function $\omega(x) = 1$, so the Gauss quadrature rule developed using that weight function can be referred to as a Gauss-Legendre rule. Similarly, Hermite polynomials are orthogonal with respect to $\omega(x) = e^{-x^2/2}$, so a quadrature rule using that weight function is referred to as a Gauss-Hermite quadrature rule. Algorithms exist for generating node and weight schemes for various orthogonal polynomials [15].

1.2.3 Continuation

Continuation methods start with a known solution to a problem and seek nearby approximate solutions to form a family of solutions. In [16], a common formulation of continuation problems is developed. The continuation zero problem $\Phi(u) = 0$ is defined in terms of $\Phi : \mathbb{R}^n \to \mathbb{R}^m$, with $n \geq m \geq 1$ and a set continuation variables u. The components of Φ are called zero functions. If the Jacobian of Φ with respect to the continuation variables is full rank at a known solution, then, for $n > m$, all solutions near the known solution lie on a unique $(n-m)$-dimensional manifold and every point on this manifold near the known solution is a solution to the zero problem.

The extended continuation problem $F(u,\mu) = 0$ in terms of continuation variables u and continuation parameters $\mu \in \mathbb{R}^r$ is given in terms of the mapping

$$F \mapsto \begin{pmatrix} \Phi(u) \\ \Psi(u) - \mu \end{pmatrix}. \tag{1.8}$$

The components of Ψ are referred to as monitor functions and are some nonlinear functions of the continuation variables whose values are tracked by the continuation parameters μ.

The notion of inactive and active continuation parameters is arrived at by splitting the continuation parameters μ into $\mu_{\mathbb{I}}$ and $\mu_{\mathbb{J}}$ where $\mathbb{I} \subseteq \{1, 2, \ldots r\}$ is an index set with cardinality $|\mathbb{I}| \leq n-m$ and $\mathbb{J}$ is the complement of $\mathbb{I}$ in $\{1, 2, \ldots r\}$. The inactive continuation parameters $\mu_{\mathbb{I}}$ have their values fixed during continuation at $\mu_{\mathbb{I}} = \mu_{\mathbb{I}}^*$, and thus impose additional constraints on the continuation variables. The solution

u^*, then, must satisfy the reduced continuation problem

$$\begin{pmatrix} \Phi(u) \\ \Psi_{\mathbb{I}}(u) - \mu_{\mathbb{I}}^* \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{1.9}$$

The restricted continuation problem is given by

$$F(u, \mu)\,|_{\mu_{\mathbb{I}} = \mu_{\mathbb{I}}^*} = 0. \tag{1.10}$$

If the Jacobian of the restriction with respect to u and $\mu_{\mathbb{J}}$ at a known solution is full rank, the corresponding solution manifold is $(n - m - |\mathbb{I}|)$-dimensional.

1.2.4 The Computational Continuation Core and Toolboxes

Several software packages exist that specialize in continuation. These include AUTO [17], MatCont [18], and COCO [16]. COCO stands out by providing support for a staged construction approach to defining continuation problems wherein larger problems are built up from smaller ones.

In COCO both monitor functions and zero functions are added to a continuation problem structure via the core constructor `coco_add_func`. The construction of more complicated problem structures, like those required for the discretization of an ODE, can be generalized and packaged in a COCO toolbox. Toolbox constructors act as wrapper functions, setting up toolbox-specific data structures and making embedded calls to `coco_add_func` and other COCO core functions. The work in this thesis is enabled by COCO and the toolboxes `'coll'` and `'bvp'` which provide support for continuing solutions of ordinary differential equations and boundary-value problems, respectively. Additionally, the `'uq'` toolbox for performing uncertainty quantification of boundary-value problems is developed for the investigations in this thesis. Those seeking more information on continuation or COCO in general (including its many existing toolboxes and core functions) are referred to the text [16] and the tutorials available in the COCO Sourceforge repository [19].

1.3 Thesis Outline

The remainder of this document will develop the equations required to solve robust design optimization problems in the presence of boundary-value problem constraints using numerical continuation.

Chapter 2 shows how to find candidate solutions of optimization problems using a method of numerical continuation. Necessary conditions are formulated and solved in stages. To illustrate the method, an optimization problem with linear periodic dynamic constraints is solved using both analytical and numerical means.

Chapter 3 outlines the PCE method and discusses the necessary zero and monitor functions needed to calculate the expansion as part of a continuation problem in COCO. A toolbox constructor is introduced that enables numerical approximation of the coefficients of expansions for response functions associated with boundary-value problems. The numerical method is applied to a linear periodic dynamic system with uncertainty and the results are compared to an analytical solution.

Chapter 4 combines the work of the previous two chapters, showing that the equations required for robust design optimization in the presence of dynamic constraints fit within the framework of Chapter 2.

Chapter 5 applies the robust design optimization formulation to a harmonically forced Duffing oscillator with periodic boundary conditions. Convergence of the statistical moments approximated by the PCE is briefly discussed. Finally, the effects of the relative weighting of the statistical moments in a robust objective function is investigated.

Chapter 2

Optimization with Dynamic Constraints

The goal of this chapter is to derive first order optimality conditions for an objective function constrained by a finite-dimensional boundary-value problem as well as integral and algebraic constraints. Additionally, a method for finding a stationary point through successive continuation is discussed. The chapter begins with an example problem in terms of an objective function constrained by a linear, periodic boundary-value problem. An analytical solution is found using the successive continuation method. The optimality conditions and solution method are then generalized for use in later chapters.

2.1 A Motivating Example

This section closely follows the approach to optimization illustrated in Section 8 of [20]. Consider the harmonically forced, damped linear oscillator with periodic boundary conditions given by

$$\dot{x}_1(t) = x_2(t)\,, \tag{2.1}$$

$$\dot{x}_2(t) = \cos(t+\phi) - x_2(t) - kx_1(t)\,, \tag{2.2}$$

$$x_1(0) = x_1(2\pi)\,, \tag{2.3}$$

$$x_2(0) = x_2(2\pi)\,. \tag{2.4}$$

We seek a stiffness, k, and phase angle, ϕ, that correspond to a stationary point of the implicitly defined objective function $x_1(0)$ (the example in [20] considers the same boundary-value problem, but seeks an optimum of $x_2(0)$). To that end, consider the Lagrangian

$$\begin{aligned}\mathcal{L}\left(x(t)\,,k,\phi,\mu_k,\mu_\phi,\mu_1,\lambda(t)\,,\lambda_{bc},\eta_k,\eta_\phi,\eta_1\right) = \mu_1 + \int_0^{2\pi} \lambda_1(t)\left(\dot{x}_1(t) - x_2(t)\right)dt \\ + \int_0^{2\pi} \lambda_2(t)\left(\dot{x}_2(t) + x_2(t) + kx_1(t) - \cos(t+\phi)\right)dt + \lambda_{bc,1}\left(x_1(2\pi) - x_1(0)\right) \\ + \lambda_{bc,2}\left(x_2(2\pi) - x_2(0)\right) + \eta_k\left(k-\mu_k\right) + \eta_\phi\left(\phi - \mu_\phi\right) + \eta_1\left(x_1(0) - \mu_1\right).\end{aligned} \tag{2.5}$$

The objective function has been replaced by the parameter μ_1 and the original objective function $x_1(0)$ is treated as equality constrained to the new parameter μ_1. Necessary conditions for a stationary point are obtained by letting the variation of the Lagrangian equal zero for arbitrary variations of its arguments. Here,

$$\begin{aligned}
\delta\mathcal{L} = \delta\mu_1 &+ \int_0^{2\pi} \left(\delta\lambda_1(t)\left(\dot{x}_1(t) - x_2(t)\right) + \lambda_1(t)\left(\delta\dot{x}_1(t) - \delta x_2(t)\right)\right) dt \\
&+ \int_0^{2\pi} \left(\delta\lambda_2(t)\left(\dot{x}_2(t) + x_2(t) + kx_1(t) - \cos(t+\phi)\right)\right. \\
&\left. + \lambda_2(t)\left(\delta\dot{x}_2(t) + \delta x_2(t) + \delta k x_1(t) + k\delta x_1(t) + \sin(t+\phi)\delta\phi\right)\right) dt \\
&+ \delta\lambda_{bc,1}\left(x_1(2\pi) - x_1(0)\right) + \lambda_{bc,1}\left(\delta x_1(2\pi) - \delta x_1(0)\right) \\
&+ \delta\lambda_{bc,2}\left(x_2(2\pi) - x_2(0)\right) + \lambda_{bc,2}\left(\delta x_2(2\pi) - \delta x_2(0)\right) \\
&+ \delta\eta_k\left(k - \mu_k\right) + \delta\eta_\phi\left(\phi - \mu_\phi\right) + \delta\eta_1\left(x_1(0) - \mu_1\right) \\
&+ \eta_k\left(\delta k - \delta\mu_k\right) + \eta_\phi\left(\delta\phi - \delta\mu_\phi\right) + \eta_1\left(\delta x_1(0) - \delta\mu_1\right).
\end{aligned} \tag{2.6}$$

The $\delta\dot{x}_1$ and $\delta\dot{x}_2$ variations in 2.6 are not independent of the other variations. To get an independent set of variations, integration by parts is used on the corresponding terms:

$$\int_0^{2\pi} \lambda_1(t)\,\delta\dot{x}_1(t)\,dt = \lambda_1(2\pi)\,\delta x_1(2\pi) - \lambda_1(0)\,\delta x_1(0) - \int_0^{2\pi} \dot{\lambda}_1(t)\,\delta x_1(t)\,dt, \tag{2.7}$$

$$\int_0^{2\pi} \lambda_2(t)\,\delta\dot{x}_2(t)\,dt = \lambda_2(2\pi)\,\delta x_2(2\pi) - \lambda_2(0)\,\delta x_2(0) - \int_0^{2\pi} \dot{\lambda}_2(t)\,\delta x_2(t)\,dt. \tag{2.8}$$

In order for the condition $\delta\mathcal{L} = 0$ to be true for arbitrary variations of the arguments of $\mathcal{L}$, the coefficients of each independent variation must equal zero. To that end, (2.7) and (2.8) are substituted into (2.6) and the coefficients of each independent variation are collected and set equal to zero, giving the following system of equations:

$$\delta\lambda_1(t): \quad \dot{x}_1(t) - x_2(t) = 0, \tag{2.9}$$

$$\delta\lambda_2(t): \quad \dot{x}_2(t) + x_2(t) + kx_1(t) - \cos(t+\phi) = 0, \tag{2.10}$$

$$\delta\lambda_{bc,1}: \quad x_1(2\pi) - x_1(0) = 0, \tag{2.11}$$

$$\delta\lambda_{bc,2}: \quad x_2(2\pi) - x_2(0) = 0, \tag{2.12}$$

$$\delta\eta_k: \quad k - \mu_k = 0, \tag{2.13}$$

$$\delta\eta_\phi: \quad \phi - \mu_\phi = 0, \tag{2.14}$$

$$\delta\eta_1: \quad x_1(0) - \mu_1 = 0, \tag{2.15}$$

$$\delta x_1(t): \quad -\dot{\lambda}_1(t) + k\lambda_2(t) = 0, \tag{2.16}$$

$$\delta x_2(t): \quad -\dot{\lambda}_2(t) + \lambda_2(t) - \lambda_1(t) = 0, \tag{2.17}$$

$$\delta x_1(0): \quad -\lambda_1(0) + \eta_1 + \lambda_{bc,1} = 0, \tag{2.18}$$

$$\delta x_2(0): \quad -\lambda_2(0) + \lambda_{bc,2} = 0, \tag{2.19}$$

$$\delta x_1(2\pi): \quad \lambda_1(2\pi) - \lambda_{bc,1} = 0, \tag{2.20}$$

$$\delta x_2(2\pi): \quad \lambda_2(2\pi) - \lambda_{bc,2} = 0, \tag{2.21}$$

$$\delta k: \quad \int_0^{2\pi} x_1(t)\,\lambda_2(t)\,dt + \eta_k = 0, \tag{2.22}$$

$$\delta\phi: \quad \int_0^{2\pi} \sin(t+\phi)\lambda_2(t)\,dt + \eta_\phi = 0, \tag{2.23}$$

and $1-\eta_1 = 0$, $\eta_k = 0$, and $\eta_\phi = 0$ (for $\delta\mu_1$, $\delta\mu_k$, and $\delta\mu_\phi$, respectively). The above system of equations can be split between (2.9)-(2.15) and (2.16)-(2.23). The first set arises from variations with respect to the Lagrange multipliers, λ_1, λ_2, λ_{bc}, η_ϕ, η_k, and η_1. It is made up the original boundary-value problem and additional equations that introduce continuation parameters. This first system can be solved independently of the second set equations. The latter equations arise from variations with respect to the variables in the original problem and depend on the solution of the original system (through (2.22) and (2.23)) and the vanishing of η_k and η_ϕ. It is worth noting that because the objective function is here treated as a constraint, the adjoint system (2.16)-(2.23) is satisfied by the trivial solution where all Lagrange multipliers equal zero.

Equations (2.9)-(2.23) combined with $\eta_1 - \nu_1 = 0$, $\eta_k - \nu_k = 0$, $\eta_\phi - \nu_\phi = 0$ represent an extended continuation problem in the continuation variables $(x(t), k, \phi, \lambda(t), \lambda_{bc}, \eta_1, \eta_k, \eta_\phi)$ and continuation parameters $(\mu_1, \mu_k, \mu_\phi, \nu_1, \nu_k, \nu_\phi)$. Equations (2.9)-(2.14) and (2.16)-(2.19) can be used to solve for $x_1(t)$, $x_2(t)$, k, ϕ, $\lambda_1(t)$, $\lambda_2(t)$, $\lambda_{bc,1}$, $\lambda_{bc,2}$, η_k, η_ϕ in terms of η_1, μ_k, and μ_ϕ. Notably, since only periodic solutions for $x_1(t)$

and $x_2(t)$ are sought, the focus is on solutions particular to the harmonic forcing function (and linearity) that are of the form

$$x_1(t) = A\sin(t) + B\cos(t). \tag{2.24}$$

The values of A and B can be determined through the method of undetermined coefficients by substituting (2.24) into (2.9) and (2.10). After substitution of the solution into (2.13) and (2.14), the resulting periodic solutions to the original ODE system are given by

$$x_1(t) = \frac{(\mu_k - 1)\cos(t+\mu_\phi) + \sin(t+\mu_\phi)}{(\mu_k-1)^2+1}, \tag{2.25}$$

$$x_2(t) = \frac{-(\mu_k - 1)\sin(t+\mu_\phi) + \cos(t+\mu_\phi)}{(\mu_k-1)^2+1}. \tag{2.26}$$

It follows from (2.16)-(2.19) that

$$\lambda_1(t) = e^{t/2}\left(\frac{\sqrt{1-4\mu_k}\,(\eta_1+\lambda_{bc,1})\cosh\left(\frac{1}{2}\sqrt{1-4\mu_k}\,t\right) - (\eta_1 - 2\mu_k\lambda_{bc,2}+\lambda_{bc,1})\sinh\left(\frac{1}{2}\sqrt{1-4\mu_k}\,t\right)}{\sqrt{1-4\mu_k}}\right), \tag{2.27}$$

$$\lambda_2(t) = e^{t/2}\left(\frac{(\lambda_{bc,2} - 2(\eta_1+\lambda_{bc,1}))\sinh\left(\frac{1}{2}\sqrt{1-4\mu_k}\,t\right) + \lambda_{bc,2}\sqrt{1-4\mu_k}\cosh\left(\frac{1}{2}\sqrt{1-4\mu_k}\,t\right)}{\sqrt{1-4\mu_k}}\right). \tag{2.28}$$

The boundary conditions (2.20)-(2.21) imply that

$$\lambda_{bc,1} = \frac{\eta_1\left(e^{\pi}(1-4\mu_k) + \sqrt{1-4\mu_k}\sinh\left(\pi\sqrt{1-4\mu_k}\right) + (4\mu_k-1)\cosh\left(\pi\sqrt{1-4\mu_k}\right)\right)}{2(4\mu_k-1)\left(\cosh(\pi) - \cosh\left(\pi\sqrt{1-4\mu_k}\right)\right)}, \tag{2.29}$$

$$\lambda_{bc,2} = -\frac{\eta_1\sinh\left(\pi\sqrt{1-4\mu_k}\right)}{\sqrt{1-4\mu_k}\left(\cosh(\pi) - \cosh\left(\pi\sqrt{1-4\mu_k}\right)\right)}. \tag{2.30}$$

Substituting (2.29)-(2.30) and $\tilde{\mu}_k = \sqrt{1-4k}$ into (2.27)-(2.28) then yields

$$\lambda_1(t) = -\frac{\left(\sinh\left(\frac{\tilde{\mu}_k t}{2}\right) - \tilde{\mu}_k\cosh\left(\frac{\tilde{\mu}_k t}{2}\right) + e^{\pi}\left(\sinh\frac{\tilde{\mu}_k}{2}(2\pi - t) + \tilde{\mu}_k\cosh\frac{\tilde{\mu}_k}{2}(2\pi-t)\right)\right)}{2\tilde{\mu}_k(\cosh(\pi) - \cosh(\pi\tilde{\mu}_k))}\eta_1 e^{\frac{t}{2}-\pi}, \tag{2.31}$$

$$\lambda_2(t) = -\frac{\left(\sinh\left(\frac{\tilde{\mu}_k t}{2}\right) + e^{\pi}\sinh\frac{\tilde{\mu}_k}{2}(2\pi - t)\right)}{\tilde{\mu}_k(\cosh(\pi) - \cosh(\pi\tilde{\mu}_k))}\eta_1 e^{\frac{t}{2}-\pi}. \tag{2.32}$$

Finally, the integrals in (2.22)-(2.23) can be evaluated to give the following expressions for η_k and η_ϕ:

$$\eta_k = -\frac{\eta_1(2(\mu_k - 1)\sin(\mu_\phi) + (\mu_k - 2)\mu_k\cos(\mu_\phi))}{((\mu_k - 1)^2 + 1)^2}, \tag{2.33}$$

$$\eta_\phi = \frac{\eta_1((1 - \mu_k)\sin(\mu_\phi) + \cos(\mu_\phi))}{(\mu_k - 1)^2 + 1}. \tag{2.34}$$

The remaining equations are then given by

$$\frac{(\mu_k - 1)\cos(\mu_\phi) + \sin(\mu_\phi)}{(\mu_k - 1)^2 + 1} - \mu_1 = 0, \tag{2.35}$$
$$-\frac{\eta_1(2(\mu_k - 1)\sin(\mu_\phi) + (\mu_k - 2)\mu_k\cos(\mu_\phi))}{((\mu_k - 1)^2 + 1)^2} - \nu_k = 0, \tag{2.36}$$
$$\frac{\eta_1((1 - \mu_k)\sin(\mu_\phi) + \cos(\mu_\phi))}{(\mu_k - 1)^2 + 1} - \nu_\phi = 0, \tag{2.37}$$
$$\eta_1 - \nu_1 = 0. \tag{2.38}$$

These are 4 equations in 7 unknowns $(\mu_1, \mu_k, \mu_\phi, \eta_1, \nu_1, \nu_k, \nu_\phi)$, so fixing μ_ϕ and setting $\nu_k = 0$ (which satisfies the necessary condition for η_k) results in three possible one-dimensional solution manifolds. The first is given by

$$\mu_1 = \frac{(\mu_k - 1)\cos(\mu_\phi) + \sin(\mu_\phi)}{(\mu_k - 1)^2 + 1}, \eta_1 = \nu_1 = \nu_\phi = 0 \tag{2.39}$$

and corresponds to the trivial solution for all Lagrange multipliers. The other two solutions can be identified by solving (2.36) for μ_k and substituting the result into the remaining equations to yield

$$\mu_k = 1 - \tan(\mu_\phi) - \sec(\mu_\phi), \mu_1 = \frac{1}{2}(\sin(\mu_\phi) - 1), \eta_1 = \nu_1, \nu_\phi = \frac{\nu_1}{2}\cos(\mu_\phi) \tag{2.40}$$

and

$$\mu_k = 1 - \tan(\mu_\phi) + \sec(\mu_\phi), \mu_1 = \frac{1}{2}(\sin(\mu_\phi) + 1), \eta_1 = \nu_1, \nu_\phi = \frac{\nu_1}{2}\cos(\mu_\phi), \tag{2.41}$$

which intersect the trivial manifold at

$$\mu_k = 1 - \tan(\mu_\phi) - \sec(\mu_\phi), \mu_1 = \frac{1}{2}(\sin(\mu_\phi) - 1), \eta_1 = \nu_1 = \nu_\phi = 0 \tag{2.42}$$

and

$$\mu_k = 1 - \tan(\mu_\phi) + \sec(\mu_\phi), \mu_1 = \frac{1}{2}(\sin(\mu_\phi)+1), \eta_1 = \nu_1 = \nu_\phi = 0, \tag{2.43}$$

respetively. These points are stationary points of μ_1 evaluated along the first manifold. Interestingly, the second set of manifolds include points with $\nu_1 = 1$, namely

$$\mu_k = 1 - \tan(\mu_\phi) - \sec(\mu_\phi), \mu_1 = \frac{1}{2}(\sin(\mu_\phi)-1), \eta_1 = 1, \nu_\phi = \frac{1}{2}\cos(\mu_\phi) \tag{2.44}$$

and

$$\mu_k = 1 - \tan(\mu_\phi) + \sec(\mu_\phi), \mu_1 = \frac{1}{2}(\sin(\mu_\phi)+1), \eta_1 = 1, \nu_\phi = \frac{1}{2}\cos(\mu_\phi). \tag{2.45}$$

Each of these points lies on a one-dimensional manifold parameterized by μ_ϕ. The final necessary condition $\eta_\phi = 0$ is satisfied when μ_ϕ takes on values of $\frac{\pi}{2}$ along the first manifold and $-\frac{\pi}{2}$ along the second manifold. The resulting optimal stiffness value is given by $\mu_k = 1$.

As a final step in this analysis the Lagrange multipliers $\lambda_{bc,1}$ and $\lambda_{bc,2}$ can now be calculated. While they are of no practical interest to the solution, they do provide another point of comparison for the numerical solution later in the chapter. Substitution into (2.29)-(2.30) yields

$$\lambda_{bc,1} = \frac{3e^\pi + \sqrt{3}\sin\left(\sqrt{3}\pi\right) - 3\cos\left(\sqrt{3}\pi\right)}{6\cos\left(\sqrt{3}\pi\right) - 6\cosh(\pi)} \approx -1.0088, \tag{2.46a}$$

$$\lambda_{bc,2} = \frac{\sin\left(\sqrt{3}\pi\right)}{\sqrt{3}\left(\cos\left(\sqrt{3}\pi\right) - \cosh(\pi)\right)} \approx 0.039412. \tag{2.46b}$$

2.2 Adjoint Derivation

To generalize the successive continuation method of seeking optimal solutions in the previous section, consider the following optimization problem with algebraic, differential, and integral constraints:

$$\begin{aligned} \text{minimize:} \quad & F = \Phi(T, x(0), x(T), p) + \textstyle\int_0^T g(t, x(t), p)dt \\ \text{subject to:} \quad & \dot{x}(t) - f(t, x(t), p) = 0, \quad B(T, x(0), x(T), p) = 0, \quad \textstyle\int_0^T h(t, x(t), p)dt = 0. \end{aligned}$$

Note that the objective function(al) maps the time-dependent function $x(t)$ to a scalar. Here, the objective function consists of two parts. The function Φ operates on the boundary values of x, interval length T, and problem parameters p. The second term integrates the function g over the entire trajectory.

As discussed in Section 1.1.1, the method of Lagrange multipliers applied to an equality constrained optimization problem is associated with the formulation of a Lagrangian. The vanishing of the first order variations of this Lagrangian under variation of each of its arguments generates a system of equations, the solutions of which correspond to potential extrema of F. Following [21], the Lagrangian for the above optimization problem is given by

$$\begin{aligned}\mathcal{L}\left(T, x(t), p, \mu_1, \mu_2, \lambda_1(t), \lambda_2, \lambda_3, \eta_1, \eta_2\right) = \mu_1 \\ + \int_0^T \lambda_1^T(t)\left(\dot{x}(t) - f\left(t, x(t), p\right)\right) dt + \lambda_2^T B\left(T, x(0), x(T), p\right) + \lambda_3 \int_0^T h\left(t, x(t), p\right) dt \\ + \eta_1 \left(\Phi(T, x(0), x(T), p) + \int_0^T g(t, x(t), p) dt - \mu_1\right) + \eta_2^T \left(p - \mu_2\right), \quad (2.47)\end{aligned}$$

where the λ_i's and η_i's are Lagrange multipliers. Note that μ_1 is here treated as the objective function while the original objective function has, in effect, been converted to a constraint on the problem. Similar to the example in the previous section, this results in adjoint equations that are linear in the Lagrange multipliers and simplifies the task of satisfying them at the outset of the optimization method.

Rescaling time by the transformation $\{t = T\tau \mid \tau \in [0, 1]\}$ results in

$$\begin{aligned}\mathcal{L}\left(T, \tilde{x}(\tau), p, \mu_1, \mu_2, \tilde{\lambda}_1(\tau), \lambda_2, \lambda_3, \eta_1, \eta_2\right) = \mu_1 \\ + \int_0^1 \tilde{\lambda}_1^T(\tau)\left(\tilde{x}'(\tau) - T f\left(T\tau, \tilde{x}(\tau), p\right)\right) d\tau + \lambda_2^T B\left(T, \tilde{x}(0), \tilde{x}(1), p\right) + \lambda_3 \left(T \int_0^1 h\left(T\tau, \tilde{x}(\tau), p\right) d\tau\right) \\ + \eta_1 \left(\Phi(T, \tilde{x}(0), \tilde{x}(1), p) + T \int_0^1 g(T\tau, \tilde{x}(\tau), p) d\tau - \mu_1\right) + \eta_2^T \left(p - \mu_2\right), \quad (2.48)\end{aligned}$$

where $\tilde{x}(\tau) = x(t(\tau))$ and $\tilde{\lambda}_1(\tau) = \lambda_1(t(\tau))$. Proceed to consider partial variations of the Lagrangian in (2.48) under variations in each of its arguments. For $\delta\tilde{\lambda}_1(\tau)$, the corresponding partial variation is given by:

$$\int_0^1 \delta\tilde{\lambda}_1^T(\tau)\left(\tilde{x}'(\tau) - T f\left(T\tau, \tilde{x}(\tau), p\right)\right) d\tau. \quad (2.49)$$

The partial variation corresponding to $\delta\lambda_2$ is given by

$$\delta\lambda_2^T B\left(T, \tilde{x}(0), \tilde{x}(1), p\right). \quad (2.50)$$

The partial variation corresponding to $\delta\lambda_{bc,1}$ is given by

$$\delta\lambda_3 \left(T \int_0^1 h\left(T\tau, \tilde{x}(\tau), p\right) d\tau\right). \quad (2.51)$$

The partial variation corresponding to $\delta\eta_1$ is given by

$$\delta\eta_1 \left(\Phi(T, \tilde{x}(0), \tilde{x}(1), p) + T \int_0^1 g(T\tau, \tilde{x}(\tau), p) d\tau - \mu_1 \right). \tag{2.52}$$

The partial variation corresponding to $\delta\eta_2$ is given by

$$\delta\eta_2^T (p - \mu_2). \tag{2.53}$$

The partial variation corresponding to $\tilde{x}'(\tau)$ is given by

$$\int_0^1 \tilde{\lambda}_1^T(\tau) \, \delta\tilde{x}'(\tau) \, d\tau. \tag{2.54}$$

However, the variation of $\tilde{x}'(\tau)$ is not independent of the other variations. Integration by parts gives

$$\tilde{\lambda}_1^T(1) \, \delta\tilde{x}(1) - \tilde{\lambda}_1^T(0) \, \delta\tilde{x}(0) - \int_0^1 \tilde{\lambda}_1'^T(\tau) \, \delta\tilde{x}(\tau) \, d\tau. \tag{2.55}$$

The partial variation corresponding to $\delta\tilde{x}(\tau)$ (including the result of (2.55)) then becomes

$$\int_0^1 \left(-\tilde{\lambda}_1'^T(\tau) - T\tilde{\lambda}_1^T(\tau) f_{,\tilde{x}}(T\tau, \tilde{x}(\tau), p) + T\lambda_3 h_{,\tilde{x}}(T\tau, \tilde{x}(\tau), p) + T\eta_1 g_{,\tilde{x}}(T\tau, \tilde{x}(\tau), p) \right) \delta\tilde{x}(\tau) \, d\tau, \tag{2.56}$$

where subscripts preceded by a comma indicate partial differentiation corresponding to the sub-scripted variable. The partial variation corresponding to $\delta\tilde{x}(0)$ (including the result of (2.55)) gives

$$\left(-\tilde{\lambda}_1^T(0) + \lambda_2^T B_{,\tilde{x}(0)}(T, \tilde{x}(0), \tilde{x}(1), p) + \eta_1 \Phi_{,\tilde{x}(0)}(T, \tilde{x}(0), \tilde{x}(1), p) \right) \delta\tilde{x}(0). \tag{2.57}$$

The partial variation corresponding to $\delta\tilde{x}(1)$ (including the result of (2.55)) gives

$$\left(\tilde{\lambda}_1^T(1) + \lambda_2^T B_{,\tilde{x}(1)}(T, \tilde{x}(0), \tilde{x}(1), p) + \eta_1 \Phi_{,\tilde{x}(1)}(T, \tilde{x}(0), \tilde{x}(1), p) \right) \delta\tilde{x}(1). \tag{2.58}$$

The partial variation corresponding to δT is given by

$$\Bigg(\lambda_2^T B_{,T}\left(T,\tilde{x}\left(0\right),\tilde{x}\left(1\right),p\right)+\eta_1\Phi_{,T}\left(T,\tilde{x}\left(0\right),\tilde{x}\left(1\right),p\right)$$
$$+\int_0^1\Big(-\tilde{\lambda}_1^T\left(\tau\right)\left(f\left(T\tau,\tilde{x},p\right)+T\tau f_{,t}\left(T\tau,\tilde{x},p\right)\right)+\lambda_3\left(h\left(T\tau,\tilde{x},p\right)+T\tau h_{,t}\left(T\tau,\tilde{x},p\right)\right)$$
$$+\eta_1\left(g\left(T\tau,\tilde{x},p\right)+T\tau g_{,t}\left(T\tau,\tilde{x},p\right)\right)\Big)d\tau\Bigg)\delta T. \quad (2.59)$$

The partial variation corresponding to δp is given by

$$\Bigg(\eta_2^T+\lambda_2^T B_{,p}\left(T,\tilde{x}\left(0\right),\tilde{x}\left(1\right),p\right)+\eta_1\Phi_{,p}\left(T,\tilde{x}\left(0\right),\tilde{x}\left(1\right),p\right)$$
$$+\int_0^1\Big(-\tilde{\lambda}_1^T\left(\tau\right)Tf_{,p}\left(T\tau,\tilde{x},p\right)+\lambda_3 Th_{,p}\left(T\tau,\tilde{x},p\right)+\eta_1 Tg_{,p}\left(T\tau,\tilde{x},p\right)\Big)d\tau\Bigg)\delta p. \quad (2.60)$$

Additional variations corresponding to $\delta\mu_1$ and $\delta\mu_2$ are given by $\left(1-\eta_1\right)\delta\mu_1$ and $\eta_2\delta\mu_2$, respectively.

To ensure that the variation of the Lagrangian is zero for arbitrary variations of its arguments, the coefficient for each individual variation must equal zero. The solution approach, inspired by [22], involves first satisfying the system of equations corresponding to setting the coefficients in (2.49)-(2.60) equal to zero and then satisfying those associated with $\delta\mu_1$ and $\delta\mu_2$ through successive continuation steps.

It is worth noting that, without the $\delta\mu_i$ variation equations, the adjoint system is homogeneous and completely satisfied by the trivial solution $\lambda_i = 0$ and $\eta_i = 0$. The constraints on the η_i's are satisfied as part of a sequence of continuation steps that first locates a local extremum of the objective function along a 1-dimensional solution manifold with the trivial solution for the Lagrange multipliers. This point also marks an intersection of the solution manifold with a branch of solutions along which non-zero λ_i's and η_i's are possible. The solution method continues along this second branch of solutions until $\eta_1 = 1$, which makes the variation corresponding to $\delta\mu_1$ equal to zero. From here, the value of η_1 is held fixed at 1 and the remaining $\delta\mu_2$ coefficient equations are satisfied by performing continuation until all the elements of η_2 have been driven to zero. When the elements of η_2 equal zero, the system is at a stationary point. Additional evaluations can be conducted at this point to further characterize this point as either an extremum or a saddle point.

2.3 Numerical Continuation

Using the above analytical derivation as a guide, we seek a numerical approximation to the example discussed in Section 2.1. Making use of the software package COCO and its `'coll'` toolbox mentioned in Section 1.2.4, the system of equations (2.9)-(2.23) are added to a continuation problem. A successful initial run marks a fold point of the objective function as a branch point (these points are located only approximately during numerical continuation). From the branch point, subsequent continuation runs allow for the satisfaction of the necessary conditions $\eta_1 = 1$, $\eta_k = \eta_\phi = 0$ resulting in optimal parameter and Lagrange multiplier values that match the analytical results.

The vector fields and Jacobians of the original and adjoint differential equations are encoded in `'coll'` compatible MATLAB functions included in Appendix A. The boundary conditions for the original and adjoint system (as well as their Jacobians) and stationarity constraints (2.22) and (2.23) are encoded as COCO compatible zero functions also provided in Appendix A.

The `ode_isol2coll` constructor is used to add both the original and adjoint differential equations to the continuation problem structure, and the `coco_add_glue` function is used to enforce equality of the problem parameters in the differential equations. The `coco_add_func` utility function is used to add the boundary conditions and stationarity constraints to the continuation problem structure, and the objective function is associated to an initially inactive continuation parameter with a call to the `coco_add_pars` utility function. The following call to the COCO entry point function denotes μ_1, k, η_1, and η_ϕ (`'mu1'`, `'k'`, `'eta1'`, and `'eta_phi'`, respectively, in the MATLAB extract) as active continuation parameters and performs continuation along a 1-dimensional solution manifold for μ_1 between 0.1 and 2.

```
>> bd1 = coco(prob, 'init', [], 1, {'mu1', 'k', 'eta1', 'eta_phi'}, {[0.1 2]});
```

The result of the call is shown below. The event identifiers `FP` and `BP` in the output show that a fold point in the objective function and branch point have been approximately detected by COCO.

```
    STEP   DAMPING               NORMS              COMPUTATION TIMES
  IT SIT     GAMMA     ||d||     ||f||     ||U||   F(x)  DF(x)  SOLVE
   0                          5.43e-04  1.16e+01    0.0    0.0    0.0
   1   1  1.00e+00  9.02e-04  1.63e-08  1.16e+01    0.0    0.0    0.0
   2   1  1.00e+00  6.33e-08  9.68e-16  1.16e+01    0.0    0.1    0.0

STEP      TIME        ||U||  LABEL  TYPE          mu1           k        eta1     eta_phi
   0  00:00:00   1.1600e+01      1  EP     2.9998e-01  4.0003e+00  0.0000e+00  0.0000e+00
  10  00:00:01   1.4153e+01      2         1.8249e-01  6.2907e+00  0.0000e+00  0.0000e+00
  20  00:00:01   1.8212e+01      3         1.2077e-01  9.1579e+00  0.0000e+00  0.0000e+00
  27  00:00:02   2.0883e+01      4  EP     1.0000e-01  1.0899e+01  0.0000e+00  0.0000e+00

STEP      TIME        ||U||  LABEL  TYPE          mu1           k        eta1     eta_phi
   0  00:00:02   1.1600e+01      5  EP     2.9998e-01  4.0003e+00  0.0000e+00  0.0000e+00
  10  00:00:03   1.1020e+01      6         4.6851e-01  2.4400e+00  0.0000e+00  0.0000e+00
  15  00:00:04   1.1345e+01      7  FP     5.0000e-01  2.0000e+00  0.0000e+00  0.0000e+00
  15  00:00:04   1.1346e+01      8  BP     5.0000e-01  1.9992e+00  0.0000e+00  0.0000e+00
  20  00:00:04   1.1888e+01      9         4.4686e-01  1.6169e+00  0.0000e+00  0.0000e+00
  30  00:00:05   1.2511e+01     10          1.3116e-01   1.1335e+00   0.0000e+00   0.0000e+00
  31  00:00:05   1.2524e+01     11  EP      1.0000e-01   1.1010e+00   0.0000e+00   0.0000e+00
```

From here, the `ode_BP2coll` constructor is used to initialize a new problem structure from the solution identified as a branch point in the previous continuation run. The constructor additionally orients the initial tangent vector to facilitate the switch to the solution branch where the Lagrange multipliers can take on non-trivial values. The boundary conditions, stationarity constraints, gluing conditions, and problem parameter assignment are added in a similar fashion to the first continuation run and the following call to the COCO entry point function is used to drive η_1 from 0 to 1 while allowing μ_1, k, and η_ϕ to vary. The Lagrange multipliers which have been encoded as active continuation parameters are also displayed.

```
>> bd2 = coco(prob, 'eta1', [], 1, {'eta1', 'mu1', 'k', 'eta_phi', 'la3', 'la4'}, {[0 1]});
```

The resulting output is shown below. The columns have been split across multiple lines to show that the Lagrange multipliers are taking on non-zero values along this solution branch.

```
 STEP      TIME        ||U||  LABEL  TYPE          eta1          mu1            k
    0  00:00:00   1.1346e+01      1  EP      0.0000e+00   5.0000e-01   1.9992e+00
    1  00:00:00   1.1346e+01      2  BP      1.3216e-08   5.0000e-01   1.9992e+00
   10  00:00:01   1.2070e+01      3  EP      1.0000e+00   5.0000e-01   2.0000e+00


 STEP       eta_phi          la3          la4
    0    0.0000e+00   0.0000e+00   0.0000e+00
    1   -6.6135e-09  -1.3137e-08  -3.7283e-10
   10   -5.0000e-01  -9.9392e-01  -2.8173e-02
```

A new problem structure is initialized from the terminal point of the previous run with the `ode_coll2coll` constructor. Supporting equations (boundary conditions, stationarity, etc.) were again added to the continuation problem structure. The following call to the COCO entry point function releases η_ϕ, μ_1, k, and ϕ, and additionally displays the Lagrange multipliers $\lambda_{bc,1}$ and $\lambda_{bc,2}$ (`'la3'` and `'la4'`, respectively) for comparison to the analytical result.

```
>> bd3 = coco(prob, 'opt', [], 1, {'eta_phi', 'mu1', 'k', 'phi', 'la3', 'la4'}, {[], [0.3 1.5]});


    STEP   DAMPING               NORMS              COMPUTATION TIMES
  IT SIT     GAMMA     ||d||     ||f||     ||U||   F(x)  DF(x)  SOLVE
   0                          5.39e-06  1.20e+01    0.0    0.0    0.0
   1   1  1.00e+00  5.39e-06  7.25e-10  1.20e+01    0.0    0.0    0.0
   2   1  1.00e+00  8.02e-09  9.32e-11  1.20e+01    0.0    0.0    0.0

 STEP      TIME        ||U||  LABEL  TYPE       eta_phi          mu1            k          phi
    0  00:00:00   1.2029e+01      1  EP     -5.0000e-01   5.0000e-01   2.0000e+00  -2.5490e-08
   20  00:00:02   1.3667e+01      3  OPT     0.0000e+00   1.0000e+00   1.0000e+00   1.5708e+00
  100  00:00:09   3.8721e+01      4  EP      4.9925e-01   5.2742e-01   5.3410e-02   3.0867e+00


 STEP           la3          la4
    0   -9.9392e-01  -2.8173e-02
   20   -1.0088e+00   3.9412e-02
  100   -3.4900e+00  -2.6366e+00
```

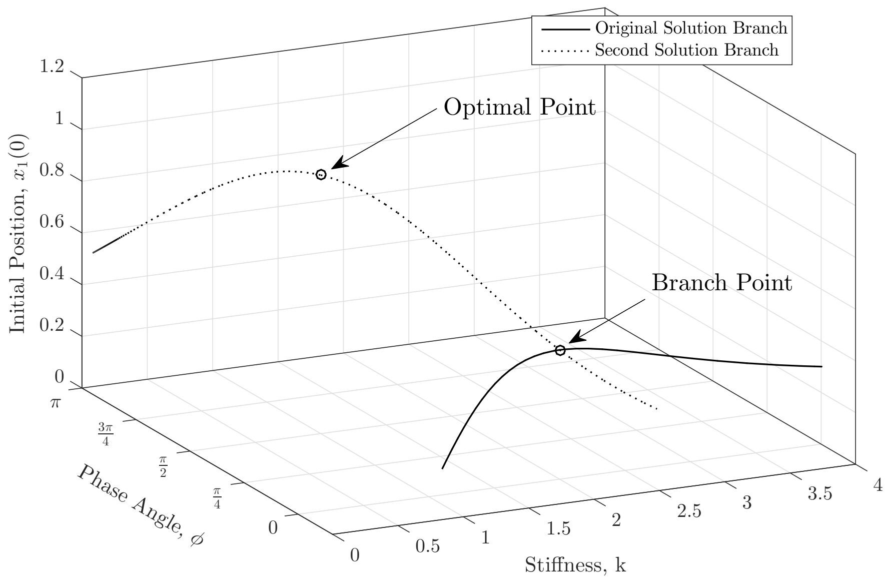

Figure 2.1: A graphical representation of the results of continuation. The point labeled 'Branch Point' denotes the approximate location of an intersection of two one-dimensional manifolds.

Continuation identifies the optimal point (where $\eta_\phi = 0$), labeled `OPT`, to be $\phi = 1.5708 \approx \frac{\pi}{2}$ as was found in the analytical solution. The columns for `la3` and `la4` in the COCO output show that their values in the run labeled `OPT` also match the analytical results for $\lambda_{bc,1}$ and $\lambda_{bc,2}$.

Figure 2.1 offers a visualization of the path taken to the optimal point during the continuation runs. The starting point for continuation is on the bottom right of the figure at $(k, \phi) = (4, 0)$. Continuation in the direction of reducing k is where COCO identified the Branch Point, and the dotted line is the tertiary solution branch along which the optimal solution is identified.

As has been mentioned, the Lagrange multipliers remain at $(\lambda_{bc,1}, \lambda_{bc,2}) = (0, 0)$ for the duration of the first continuation run. The continuation path that takes η_1 from 0 to 1 is displayed as a solid line in Figure 2.2 and during this step of the continuation process, the Lagrange multipliers take on non-zero values. The optimal point is also highlighted.

2.4 Concluding Remarks

This chapter discussed a method of identifying stationary points of constrained optimization problems through successive continuation and demonstrated its use on a linear ODE example. The example was solved both analytically and with the numerical continuation package COCO. A general set of equations and procedure were outlined for identifying stationary points of problems constrained by differential, algebraic,

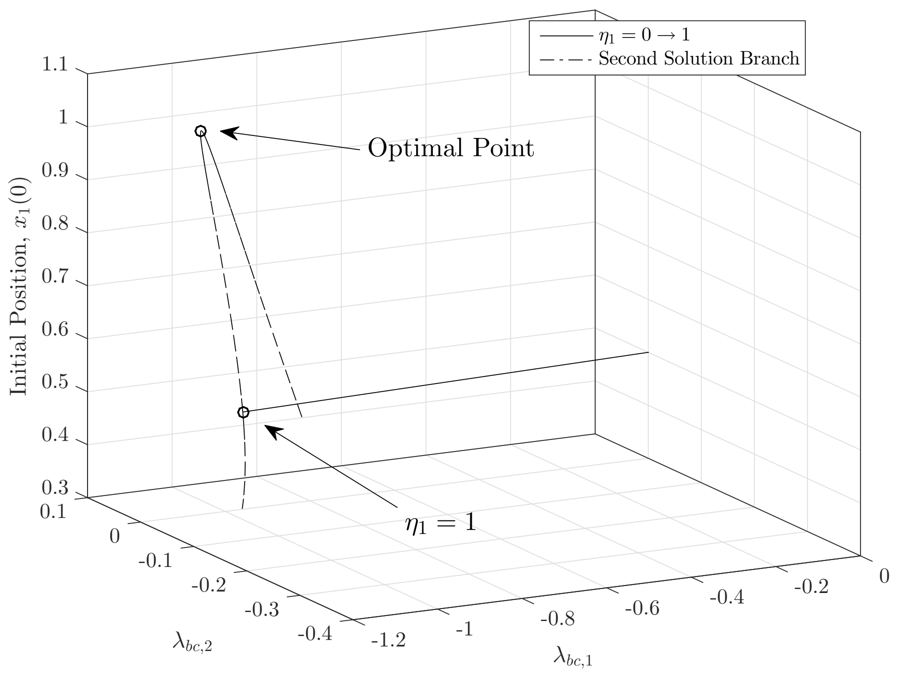

Figure 2.2: Evolution of Lagrange multipliers during continuation.

and integral constraints. The results of this derivation is that the adjoint equations for nonlinear systems need not be constructed for specific problems as has been done in this chapter. They can be derived in all generality and then specialized for a given problem [21]. In fact, a 2017 release of COCO has implemented construction of adjoint equations for optimization in the presence of particular classes of constraints. Later chapters will make use of this functionality.

Chapter 3

Polynomial Chaos Expansion

In this chapter, the focus is on calculating statistical moments of responses of systems constrained by finite dimensional boundary-value problems with parametric uncertainty using the method of Polynomial Chaos Expansion (PCE). The necessary equations for generating a PCE will be derived in a manner suitable for inclusion in a numerical continuation routine. In subsequent chapters the successive parameter continuation approach to optimization from Chapter 2 will be combined with the work in this chapter in pursuit of robust optima identified through numerical continuation.

3.1 Motivating Example

Consider the periodic boundary-value problem

$$\dot{x}_1 = x_2, \tag{3.1}$$

$$\dot{x}_2 = \cos\left(\omega t + \phi\right) - x_2 - K x_1, \tag{3.2}$$

$$x_1\left(0\right) = x_1\left(2\pi/\omega\right), \tag{3.3}$$

$$x_2\left(0\right) = x_2\left(2\pi/\omega\right), \tag{3.4}$$

in terms of the displacement and velocity of a harmonically excited linear oscillator. This system differs from (2.1)-(2.4) in two ways. First, the deterministic stiffness k is replaced by a normally distributed random variable $K \sim \mathcal{N}\left(\mu_k, \sigma_k^2\right)$. (We assume that the values of μ_k and σ_k are such that the probability of negative values is negligible.) Second, the forcing frequency is given by ω in (3.2). The maximal displacement is the system response of interest, and its mean and variance will be calculated as functions of the forcing frequency for comparison to a PCE approximation.

The steady-state solution to the ODE is given by

$$x_1\left(t, K, \omega, \phi\right) = \frac{\left(K-\omega^2\right)\cos(\omega t+\phi)+\omega\sin(\omega t+\phi)}{\left(K-\omega^2\right)^2+\omega^2}, \tag{3.5}$$

$$x_2\left(t, K, \omega, \phi\right) = \frac{-\omega\left(K-\omega^2\right)\sin(\omega t+\phi)+\omega^2\cos(\omega t+\phi)}{\left(K-\omega^2\right)^2+\omega^2}. \tag{3.6}$$

The phase variable ϕ may be constrained so that the initial position represents the sought response function. To that end, set $t = 0$ and find the value of ϕ such that $x_2(0) = 0$. The resulting phase angle is given by $\phi = \tan^{-1}\left(\frac{\omega}{K-\omega^2}\right) + \pi\mathbb{Z}$. The maximum displacement as a function of stiffness and forcing frequency is then given by the random variable

$$X_{1,0}\left(K, \omega\right) = \frac{1}{\sqrt{\left(K-\omega^2\right)^2+\omega^2}}. \tag{3.7}$$

Expressions for the mean $\mu_{X_{1,0}}$ and variance $\sigma^2_{X_{1,0}}$ of $X_{1,0}\left(K, \omega\right)$ as functions of ω can be found directly from their definitions in terms of the expectation operator $E\left[\cdot\right]$:

$$\mu_{X_{1,0}}\left(\omega\right) = E\left[X_{1,0}\left(K, \omega\right)\right] = \int_{-\infty}^{\infty} \frac{1}{\sqrt{\left(k-\omega^2\right)^2+\omega^2}} \left(\frac{e^{\frac{-\left(k-\mu_k\right)^2}{2\sigma_k^2}}}{\sqrt{2\pi\sigma_k^2}}\right) dk, \tag{3.8}$$

$$\sigma^2_{X_{1,0}}\left(\omega\right) = E\left[\left(X_{1,0}\left(K, \omega\right)\right)^2\right] - \left(E\left[X_{1,0}\left(K, \omega\right)\right]\right)^2 =$$

$$\int_{-\infty}^{\infty} \left(\frac{1}{\sqrt{\left(k-\omega^2\right)^2+\omega^2}}\right)^2 \left(\frac{e^{\frac{-\left(k-\mu_k\right)^2}{2\sigma_k^2}}}{\sqrt{2\pi\sigma_k^2}}\right) dk - \left(\int_{-\infty}^{\infty} \frac{1}{\sqrt{\left(k-\omega^2\right)^2+\omega^2}} \left(\frac{e^{\frac{-\left(k-\mu_k\right)^2}{2\sigma_k^2}}}{\sqrt{2\pi\sigma_k^2}}\right) dk\right)^2. \tag{3.9}$$

After substituting values of $\mu_k = 3$ and $\sigma_k = 0.2$, numerical integration is used to evaluate (3.8) and (3.9) at various frequencies. The resulting frequency sweeps of the mean and variance of $x_{1,0}$ are shown in Figures 3.1 and 3.2. Figure 3.3 shows the mean plus/minus three times the standard deviation as a function of frequency.

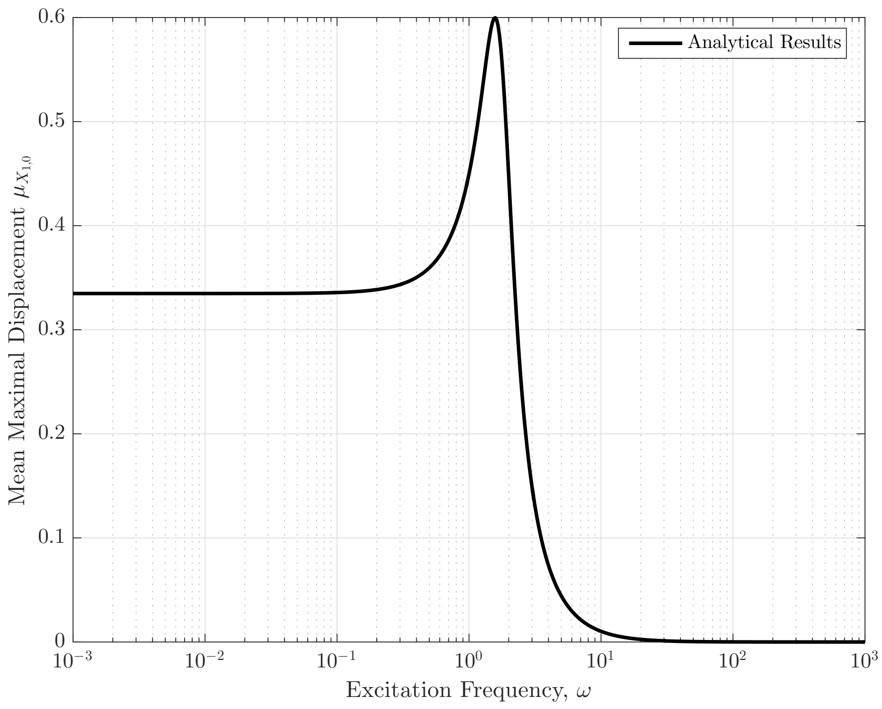

Figure 3.1: Variations in the mean of the maximal displacement $\mu_{X_{1,0}}$ as a function of excitation frequency.

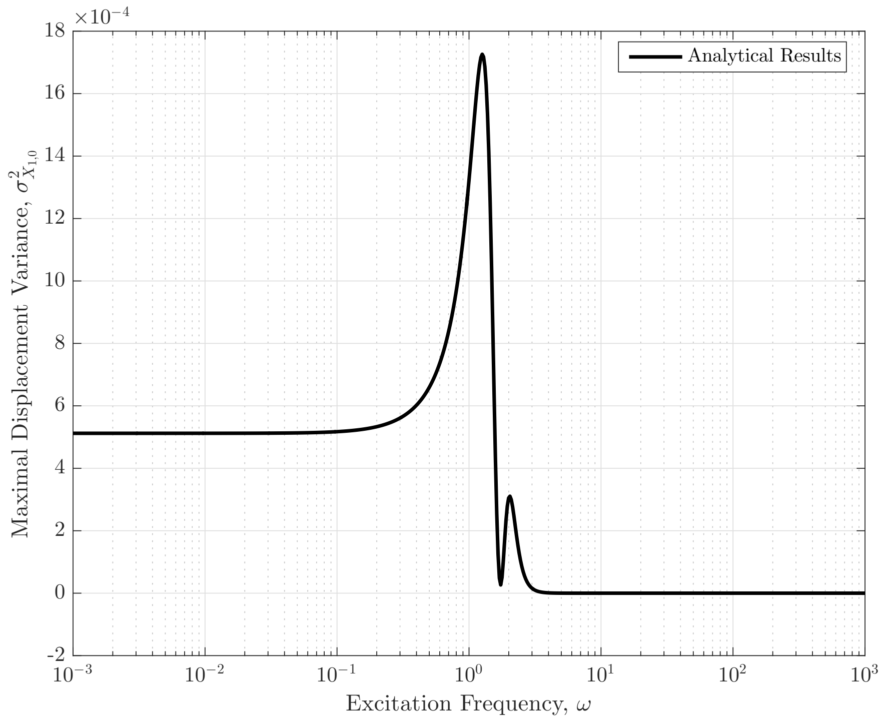

Figure 3.2: Variations in the variance of the maximal displacement $\sigma^2_{X_{1,0}}$ as a function of excitation frequency.

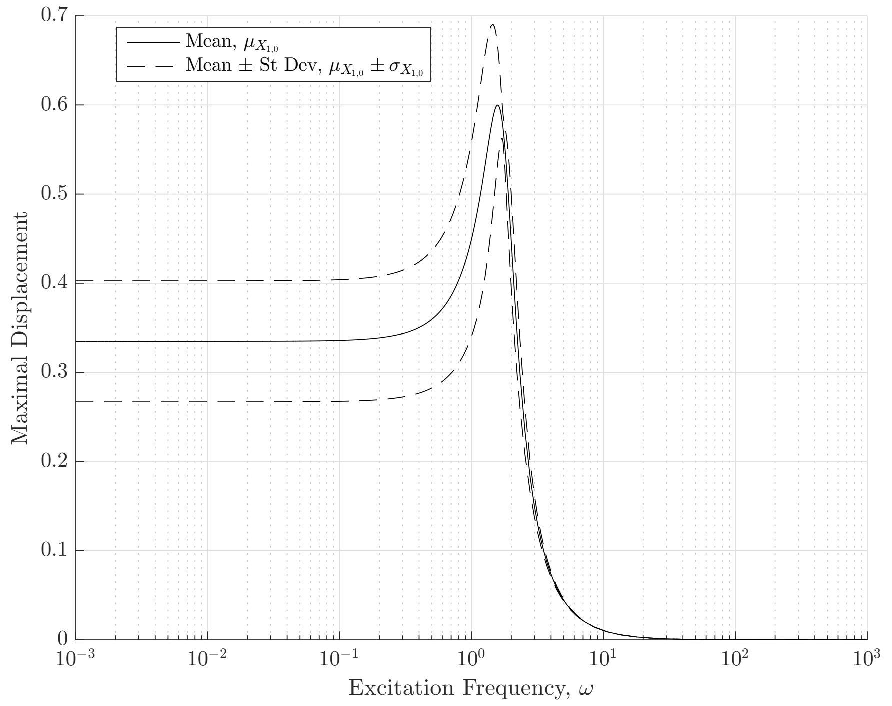

Figure 3.3: Mean and mean +/- three standard deviation curves as calculated from the analytical expressions in (3.8) and (3.9).

3.1.1 PCE Approximation

This section will work through the PCE approximation for the motivating example. The more general theory will be explained in the subsequent sections. Consider the transformation

$$K = \mu_k + \sigma_k \xi_k = 3 + 0.2\xi_k, \tag{3.10}$$

where $\xi_k \sim \mathcal{N}(0, 1)$ and K is expressed in terms of the specific numerical values for μ_k and σ_k from the previous section. The maximal displacement random variable $X_{1,0}$ in terms of ξ_k is given by substituting (3.10) into (3.7)

$$X_{1,0}(\omega, \xi_k) = \frac{1}{\sqrt{\left((3 + 0.2\xi_k) - \omega^2\right)^2 + \omega^2}}. \tag{3.11}$$

To approximate the mean and variance of (3.7) using PCE, first write (3.11) as a series expansion in the basis of polynomials Ψ_i that depend on the random variable ξ_k with coefficients α_i that depend on the deterministic variable ω:

$$X_{1,0}\left(\omega,\xi_k\right)=\sum_{i=0}^{\infty}\alpha_i\left(\omega\right)\Psi_i\left(\xi_k\right). \tag{3.12}$$

For example, truncating the expansion after four terms and choosing the probabilist's Hermite polynomials given by

$$\Psi_0\left(\xi_k\right)=1,\Psi_1\left(\xi_k\right)=\xi_k,\Psi_2\left(\xi_k\right)=\frac{1}{\sqrt{2}}\left(\xi_k^2-1\right),\Psi_3\left(\xi_k\right)=\frac{1}{\sqrt{6}}\left(\xi_k^3-3\xi_k\right) \tag{3.13}$$

as basis function yields

$$X_{1,0}\left(\omega,\xi_k\right)\approx\alpha_0(\omega)+\alpha_1\left(\omega\right)\xi_k+\frac{\alpha_2\left(\omega\right)}{\sqrt{2}}\left(\xi_k^2-1\right)+\frac{\alpha_3\left(\omega\right)}{\sqrt{6}}\left(\xi_k^3-3\xi_k\right). \tag{3.14}$$

The Hermite polynomials are an orthonormal basis under the inner product

$$\left\langle\Psi_i\left(\xi_k\right),\Psi_j\left(\xi_k\right)\right\rangle=\int_{-\infty}^{\infty}\Psi_i\left(\xi_k\right)\Psi_j\left(\xi_k\right)\frac{e^{\frac{-\xi_k^2}{2}}}{\sqrt{2\pi}}d\xi_k. \tag{3.15}$$

Taking the inner product of both sides of (3.12) with each Ψ_i gives the following system of equations

$$\int_{-\infty}^{\infty}X_{1,0}\left(\omega,\xi_k\right)\Psi_i\left(\xi_k\right)\frac{e^{\frac{-\xi_k^2}{2}}}{\sqrt{2\pi}}d\xi_k=\alpha_i\left(\omega\right),i\geq 0. \tag{3.16}$$

This remains true for $i\leq 3$ if the right-hand side of 3.14 is substituted for $x_{1,0}\left(\omega,\xi_k\right)$. Each such integral can be approximated numerically using a Gauss-Hermite numerical integration scheme. The Matlab function for determining the weights and nodes for this quadrature rule up to order m is as follows (which uses the eigenvalue algorithm given in [14]).

```matlab
function [nds, wts] = gauss_hermite_nodes(m)
  num = (1:m-1)';
  g = sqrt(num);
  J = diag(g,1)+diag(g,-1);
  [w, x] = eig(J);
  [nds, idx] = sort(diag(x));
  nds = nds';
  wts = sqrt(2*pi)*w(1,:).^2;
  wts = wts(idx)/(sqrt(2*pi));
end
```

To four decimal places, the nodes and weights for a 4th-order Gauss-Hermite integration rule are

$$(\xi_{k,1}, \xi_{k,2}, \xi_{k,3}, \xi_{k,4}) = \begin{pmatrix} -2.3344 & -0.7420 & 0.7420 & 2.3344 \end{pmatrix}, \tag{3.17}$$

$$(w_{k,1}, w_{k,2}, w_{k,3}, w_{k,4}) = \begin{pmatrix} 0.0459 & 0.4541 & 0.4541 & 0.0459 \end{pmatrix}. \tag{3.18}$$

For $i = 0, \ldots, 3$, the integral in 3.16 is then approximated by

$$\begin{pmatrix} \Psi_0(\xi_{k,1}) & \Psi_0(\xi_{k,2}) & \Psi_0(\xi_{k,3}) & \Psi_0(\xi_{k,4}) \\ \Psi_1(\xi_{k,1}) & \Psi_1(\xi_{k,2}) & \Psi_1(\xi_{k,3}) & \Psi_1(\xi_{k,4}) \\ \Psi_2(\xi_{k,1}) & \Psi_2(\xi_{k,2}) & \Psi_2(\xi_{k,3}) & \Psi_2(\xi_{k,4}) \\ \Psi_3(\xi_{k,1}) & \Psi_3(\xi_{k,2}) & \Psi_3(\xi_{k,3}) & \Psi_3(\xi_{k,4}) \end{pmatrix} \begin{pmatrix} w_1 & 0 & 0 & 0 \\ 0 & w_2 & 0 & 0 \\ 0 & 0 & w_3 & 0 \\ 0 & 0 & 0 & w_4 \end{pmatrix} \begin{pmatrix} x_{1,0}(\omega, \xi_{k,1}) \\ x_{1,0}(\omega, \xi_{k,2}) \\ x_{1,0}(\omega, \xi_{k,3}) \\ x_{1,0}(\omega, \xi_{k,4}) \end{pmatrix} = \begin{pmatrix} \alpha_0(\omega) \\ \alpha_1(\omega) \\ \alpha_2(\omega) \\ \alpha_3(\omega) \end{pmatrix}, \tag{3.19}$$

which, after substituting values for the nodes and weights, becomes:

$$\begin{pmatrix} \alpha_0(\omega) \\ \alpha_1(\omega) \\ \alpha_2(\omega) \\ \alpha_3(\omega) \end{pmatrix} = \begin{pmatrix} 0.0459 & 0.4541 & 0.4541 & 0.0459 \\ -0.1071 & -0.3369 & 0.3369 & 0.1071 \\ 0.1443 & -0.1443 & -0.1443 & 0.1443 \\ -0.1071 & 0.3369 & -0.3369 & 0.1071 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{(2.5331-\omega^2)^2+\omega^2}} \\ \frac{1}{\sqrt{(2.8516-\omega^2)^2+\omega^2}} \\ \frac{1}{\sqrt{(3.1484-\omega^2)^2+\omega^2}} \\ \frac{1}{\sqrt{(3.4669-\omega^2)^2+\omega^2}} \end{pmatrix}. \tag{3.20}$$

As shown later in this chapter, the mean and variance of the truncated expansion are given by $\alpha_0(\omega)$ and $\sum_{i=1}^{3} \alpha_i^2(\omega)$. Figures 3.4 and 3.5 show the comparison with the analytical results obtained previously. The solid line in each figure is the previously calculated analytical result and the circles are the approximations at selected frequency values. Figure 3.6 shows the mean +/- 3 standard deviation plot with samples used in the approximation at selected forcing frequencies.

The agreement in the results provides confidence that the expansion has sufficiently converged with the chosen polynomial degree and integration order. However, experimentation with these orders could show acceptable convergence to the analytical results already at lower order. One must take care to choose an integration scheme of high enough order to accurately integrate the products of orthogonal polynomials. An n-th order Gaussian quadrature rule is able to provide exact results for polynomials up to degree $2n-1$. The highest order polynomial product in the above expansion is 6 (the square of Ψ_3). The 4^{th} order integration scheme is accurate up to 7^{th} degree polynomials, so accuracy of the numerical integration is sufficient for this example. It is possible that additional terms in the polynomial expansion are needed to accurately approximate the statistical moments.

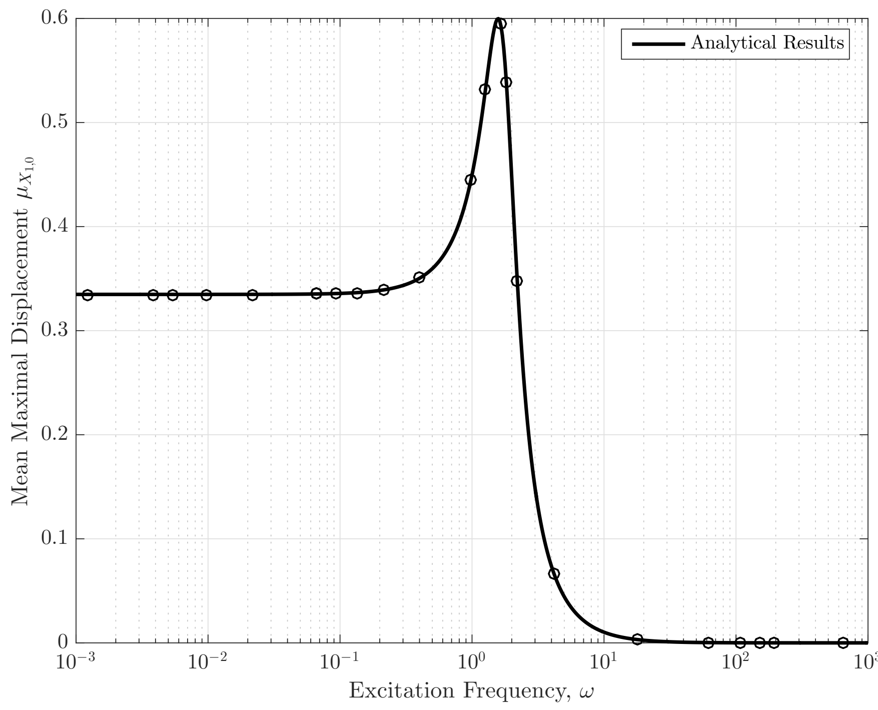

Figure 3.4: Frequency sweep of the mean value of the maximal displacement for the linear ODE with a normally distributed stiffness, $K \sim \mathcal{N}(3, 0.2^2)$. Mean values approximated by a third-order polynomial chaos expansion are shown as circles at various frequency values.

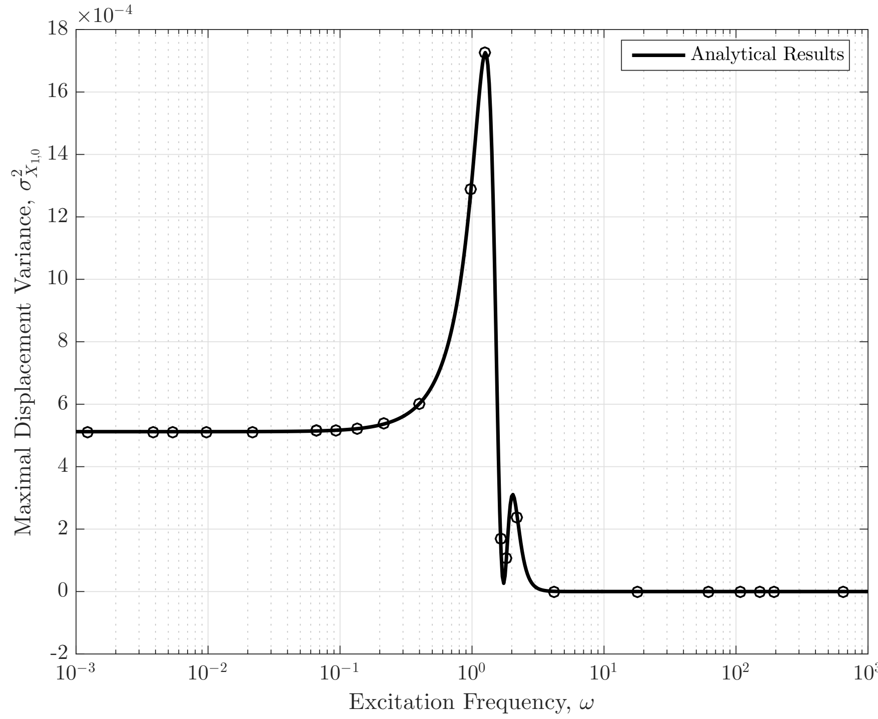

Figure 3.5: Frequency sweep of the variance of the maximal displacement for the linear ODE with a normally distributed stiffness, $K \sim \mathcal{N}(3, 0.2^2)$. Variance values approximated by a third-order polynomial chaos expansion are shown as circles at various frequency values.

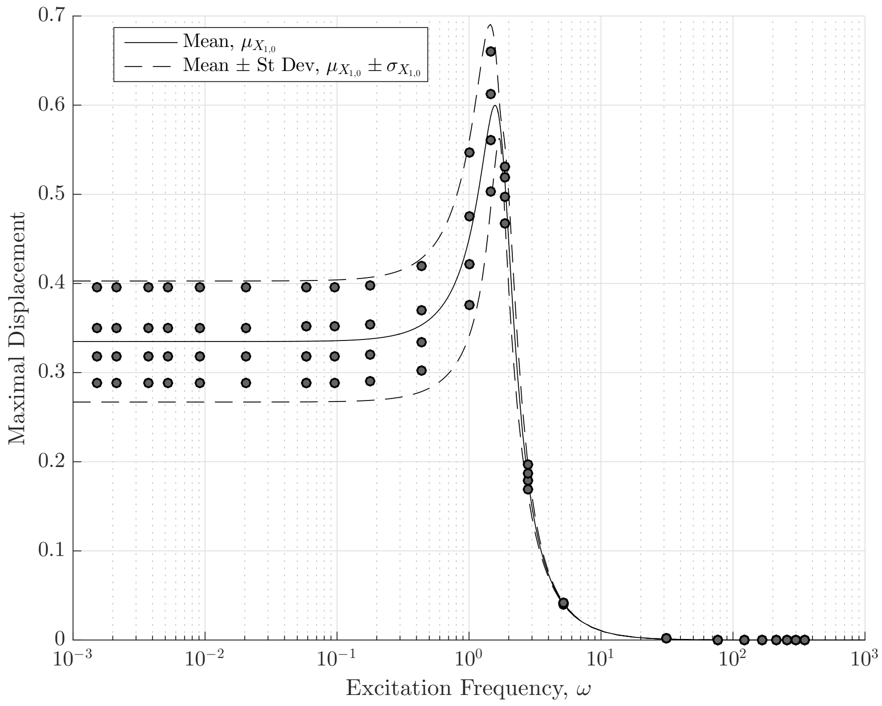

Figure 3.6: Frequency sweep of mean and mean +/- 3 standard deviation curves with samples used to calculate the polynomial chaos expansion coefficients overlaid.

3.2 PCE Background

In general, the choice of basis polynomials Ψ_i in (1.6) depends on the underlying distribution of the random variables ξ. Xiu and Karniadakis [8] generalized the method of homogeneous chaos of Wiener [23] and found optimally convergent polynomial chaos basis functions for several well-known probability distributions. They additionally discussed methods for handling random variables of arbitrary distributions. Table 3.1 outlines the optimally convergent basis functions for a few common continuous distribution types. Weight functions for additional density functions can be found in [24] or [8].

Distribution Type	Density Function	Polynomial Basis	Weight function	Support
Normal	$\frac{1}{\sqrt{2\pi}}e^{\frac{-\xi^2}{2}}$	Hermite	$\frac{1}{\sqrt{2\pi}}e^{\frac{-\xi^2}{2}}$	$[-\infty, \infty]$
Uniform	$\frac{1}{2}$	Legendre	$\frac{1}{2}$	$[-1, 1]$
Exponential	$e^{-\xi}$	Laguerre	$e^{-\xi}$	$[0, \infty]$

Table 3.1: Correspondence between probability density function and the orthogonal weight function for selected distributions and orthogonal polynomials [24]

It can be seen in Table 3.1 that the weight function is identical in form to the density function. The choice of polynomial basis functions influences the convergence of the error of the expansion. The optimal basis is one whose weight function corresponds to the density function of the random variable. With this choice, error convergence is exponential [8].

The general theory of Xiu and Karniadakis [8] concerns basis functions that are orthogonal under the inner product defined by

$$\langle f, g \rangle = \int_{\Omega} f(\xi)\, g(\xi)\, \rho_{\xi}(\xi)\, d\xi \tag{3.21}$$

with ρ_{ξ} representing the distribution-dependent weight function (from Table 3.1 or [24, 8]) and Ω determined by the support of the density function. Specifically

$$\langle \Psi_i, \Psi_j \rangle = c\, \delta_{ij} \tag{3.22}$$

where $c \in \mathbb{R}$ and δ_{ij} is the Kronecker delta function. It is always possible to choose $c = 1$ by a suitable normalization.

Polynomial chaos expansions are also available for higher-dimensional stochastic parameter spaces where $\xi = (\xi_1, \xi_2, \ldots \xi_s)$. The corresponding theory reduces to the one-dimensional case when the random variables are independent. The form of the expansion (1.6) is unchanged, but the basis polynomials are given by

$$\Psi_i(\xi) = \psi_1^{k_1}(\xi_1)\, \psi_2^{k_2}(\xi_2) \ldots \psi_s^{k_s}(\xi_s), \tag{3.23}$$

where ψ_j^k represents the k'th-degree univariate polynomial function of ξ_j and there exists some mapping $(k_1, k_2, \ldots k_s) \mapsto i$ [13]. The expansion coefficients are then determined by evaluating the multiple integral

$$\alpha_i(p) = \int_{\Omega_s} \cdots \int_{\Omega_1} r(p, \xi)\, \Psi_i(\xi_1, \xi_2, \ldots \xi_s)\, \rho_{\xi_1}(\xi_1)\, d\xi_1 \cdots \rho_{\xi_s}(\xi_s)\, d\xi_s, \tag{3.24}$$

where $\rho_{\xi_1} \rho_{\xi_2} \ldots \rho_{\xi_s}$ is the multivariate probability density function ρ_{ξ}.

The random variables in a PCE are typically understood to be *standardized* random variables. The meaning of standardized for the random variable depends on the distribution. For a normal random variable, it assumes a mean value of 0 and standard deviation of 1. For a uniform distribution it assumes a lower limit of -1 and upper limit of 1. If the random variables are not independent or standardized at the outset, then a (possibly nonlinear) transformation to a new set of random variables must be sought to get them into independent, standardized form [25].

3.3 PCE Construction

The computational task in constructing a PCE is the determination of the expansion coefficients α_i. As a first step, the expansion is truncated to a finite number of terms N_t and the inner product (3.21) is taken of both sides with each basis function in the truncated expansion

$$\langle r(p,\xi), \Psi_j(\xi)\rangle = \left\langle \sum_{i=0}^{N_t-1} \alpha_i(p)\,\Psi_i(\xi), \Psi_j(\xi) \right\rangle, \; j = 0 \dots N_t - 1. \tag{3.25}$$

Due to the orthogonality of the basis functions, the inner product on the right hand side of (3.25) is zero for all but the j'th term of the expansion which allows for the coefficient α_j to be isolated

$$\alpha_j = \frac{\langle r(p,\xi), \Psi_j(\xi)\rangle}{\langle \Psi_j^2(\xi)\rangle}. \tag{3.26}$$

If, in addition to being orthogonal, the basis functions are also normalized, then the denominator of (3.26) equals 1. In the univariate case, the expansion coefficient can now be approximated using the following numerical quadrature

$$\alpha_j = \int_\Omega r(p,\xi)\,\Psi_j(\xi)\,\rho_\xi(\xi)\,d\xi \approx \sum_{i=1}^{M} w_i\, r(p,\xi_i)\,\Psi_j\left(\xi^{(i)}\right), \tag{3.27}$$

where orthonormality of the basis functions is assumed here and going forward. Quadrature in the multivariate case will be discussed in more detail in later sections.

For the case when $\xi \in \mathbb{R}$, the basis polynomials Ψ_i are univariate and the expansion order N_t represents the maximum polynomial degree in the expansion. In the multivariate case, the so-called *total-order expansion* involves products of univariate polynomials such that the sum of the degree of individual polynomials does not exceed a defined total order P_t. As an example, consider the case where $\xi \in \mathbb{R}^3$ and $P_t = 2$, and let ψ_j^k be the k'th-degree univariate polynomial function of ξ_j as in (3.23). The multivariate basis polynomials are then given by

$$\begin{array}{lllll}\Psi_0 = \psi_1^0\psi_2^0\psi_3^0\,, & \Psi_1 = \psi_1^1\psi_2^0\psi_3^0\,, & \Psi_2 = \psi_1^0\psi_2^1\psi_3^0\,, & \Psi_3 = \psi_1^0\psi_2^0\psi_3^1\,, & \Psi_4 = \psi_1^1\psi_2^1\psi_3^0\,, \\ \Psi_5 = \psi_1^1\psi_2^0\psi_3^1\,, & \Psi_6 = \psi_1^0\psi_2^1\psi_3^1\,, & \Psi_7 = \psi_1^2\psi_2^0\psi_3^0\,, & \Psi_8 = \psi_1^0\psi_2^2\psi_3^0\,, & \Psi_9 = \psi_1^0\psi_2^0\psi_3^2.\end{array}$$

The number of terms in a total-order expansion is given by

$$N_t = \frac{(s+P_t)!}{s!P_t!}. \tag{3.28}$$

Other methods of expansion exist (like a tensor product of univariate polynomial orders [24]), but total-order expansions will be used in this thesis.

3.3.1 Numerical Quadrature

In one stochastic dimension, $\xi \in \mathbb{R}$, the quadrature weights w_i and node locations ξ_i in (3.27) are determined by Gaussian quadrature rules for the corresponding weight functions. For example, for a uniform distribution, the Gauss-Legendre (weight function $\rho_\xi = 1/2$) quadrature weights and nodes are used. Similarly, for a normal distribution, the probabilist's Gauss-Hermite weights and nodes are used.

For higher dimension, $\xi \in \mathbb{R}^s$, the integration is performed over the support of each stochastic variable as shown in (3.24). The most straightforward method for approximating the multivariate inner product in (3.24) is to use a tensor product of one-dimension quadrature rules [24]. To that end, let M_j be the integration order for the j'th stochastic parameter. The total number of terms in the numerical integration is then

$$M = \prod_{j=1}^{s} M_i. \tag{3.29}$$

Further, define $\left\{\xi_j^{(k_j)}\right\}_{k_j=1}^{M_j}$ to be the quadrature nodes for the j'th stochastic parameter. The Cartesian product given by

$$\left\{\xi_1^{(k_1)}\right\}_{k_1=1}^{M_1} \times \cdots \times \left\{\xi_s^{(k_s)}\right\}_{k_s=1}^{M_s} \tag{3.30}$$

contains every point in the stochastic input space where the response function r and basis functions Ψ must be evaluated for the numerical quadrature. The quadrature weights are created through a tensor product of the weights for one-dimensional quadrature rules

$$w_1 \otimes \cdots \otimes w_s \tag{3.31}$$

where w_j is the $1 \times M_j$ vector of quadrature weights for the j'th stochastic parameter.

Let $\Psi_j^{\Xi} = \left(\Psi_j^{(1)}, \ldots, \Psi_j^{(M)}\right)$, where $\Psi_j^{(k)}$ represents the evaluation of the j'th basis function at the k'th point in the Cartesian product (3.30). Similarly, define the $M \times 1$ column vector of response function evaluations r_Ξ as follows:

$$r_\Xi = \begin{pmatrix} r_1, \\ r_2, \\ \vdots \\ r_M \end{pmatrix}. \tag{3.32}$$

The k'th entry of r_Ξ represents the response function evaluated at the k'th point in the Cartesian product (3.30). Finally, define the $M \times M$ matrix W by applying the $\mathfrak{vec}$ operator followed by the $\mathfrak{diag}$ operator (as defined in [16]) to the weights in (3.31):

$$W = \mathfrak{diag}\left(\mathfrak{vec}\left(w_1 \otimes \cdots \otimes w_s\right)\right) = \begin{pmatrix} W_1 & 0 & \dots & 0 \\ 0 & W_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & W_M \end{pmatrix}. \tag{3.33}$$

For $j <= M$, the coefficient α_j is then approximated by the product

$$\Psi_j^\Xi W r_\Xi. \tag{3.34}$$

All of the expansion coefficients can be approximated simultaneously by constructing the $N_t \times M$ matrix $\boldsymbol{\Psi}$ whose rows are Ψ_j^Ξ for $j = 0 \dots N_t - 1$:

$$\alpha \approx \boldsymbol{\Psi} W r_\Xi. \tag{3.35}$$

A drawback of this scheme is that for high stochastic dimension, the required response function evaluations (M in total) may be large in number. It is for this reason that the straightforward application of PCE outlined here is ideally suited to problems with low stochastic dimension.

3.3.2 Construction Summary

The steps of constructing the PCE can now be summarized as follows:

1. Choose a polynomial order P_t and integration orders M_j for $j = 1 \dots s$. For accuracy of the quadrature rule ensure that $2P_t \leq 2M - 1$.

2. Choose polynomial basis functions appropriate to the random variable distributions.

3. Determine the quadrature nodes and weights for each stochastic dimension.

4. If the stochastic dimension is greater than one, construct a Cartesian product of quadrature nodes and tensor product of integration nodes and compute the matrices $\boldsymbol{\Psi}$ and W.

5. If necessary, transform the standardized quadrature nodes to their locations in the true stochastic parameter space.

6. Evaluate the response function at each transformed quadrature node and construct the response function vector r_Ξ.

7. Determine the values of the expansion coefficients using (3.35).

In the absence of some adaptive algorithm, numerical experiments must be conducted that vary P_t and the M_j's until an acceptable level of convergence is observed in the coefficients of the PCE.

3.4 Statistical Moments of the PCE

In this section the closed forms of the first two statistical moments of a PCE will be derived. The expected value of the response function is given by

$$\begin{aligned}\mu_r =&E\left[r\right] = \int_\Omega r\,\rho_\xi d\xi \approx \int_\Omega \left(\sum_{i=0}^{N_t-1} \alpha_i \Psi_i\right)\rho_\xi d\xi = \sum_{i=0}^{N_t-1}\left(\int_\Omega \alpha_i\Psi_i\rho_\xi d\xi\right)\\ =&\alpha_0\int_\Omega \Psi_0\rho_\xi d\xi + \sum_{i=1}^{N_t-1}\left(\alpha_i\int_\Omega \Psi_i\rho_\xi d\xi\right) = \alpha_0\left\langle 1,\Psi_0\right\rangle + \sum_{i=1}^{N_t-1}\left(\alpha_i\left\langle 1,\Psi_i\right\rangle\right) = \alpha_0,\end{aligned} \tag{3.36}$$

where $\Psi_0 = 1$ and the orthonormality of the basis functions have been exploited. The variance can also be calculated directly using

$$\begin{aligned}\sigma_r^2 =&E\left[r^2\right] - \left(E\left[r\right]\right)^2 = \int_\Omega r^2\,\rho_\xi d\xi - \left(\int_\Omega r\,\rho_\xi d\xi\right)^2\\ \approx&\int_\Omega\left(\sum_{i=0}^{N_t-1}\alpha_i\Psi_i\right)^2\rho_\xi d\xi - \left(\int_\Omega\left(\sum_{i=0}^{N_t-1}\alpha_i\Psi_i\right)\rho_\xi d\xi\right)^2 = \left(\sum_{i=0}^{N_t-1}\alpha_i^2\right) - \alpha_0^2 = \sum_{i=1}^{N_t-1}\alpha_i^2,\end{aligned} \tag{3.37}$$

where, again, orthonormality of the basis functions has been exploited to eliminate integrals of cross products that occur when squaring the truncated expansion.

3.5 PCE Boundary Value Continuation Problem

Consider a response function r similar in form to the objective function F in the optimization formulation of Section 2.2:

$$r\left(T,x\left(t\right),p\right) = r_{\text{bc}}\left(T,x\left(0\right),x\left(T\right),p\right) + \int_0^T r_{\text{int}}\left(x\left(t\right),p\right)dt \tag{3.38}$$

In this thesis, $x\left(t\right)$ will be restricted to functions that satisfy the boundary value problem

$$\dot{x} - f\left(t,x,p\right) = 0,\quad f_{bc}\left(T,x(0),x(T),p\right) = 0. \tag{3.39}$$

In order to construct a PCE of the response function, first split the parameters p into deterministic parameters $p_d \in \mathbb{R}^d$ and stochastic parameters $P \in \mathbb{R}^s$:

$$r\left(T, x, p_d, P\right) = r_{\mathrm{bc}}\left(T, x\left(0\right), x\left(T\right), p_d, P\right) + \int_0^T r_{\mathrm{int}}\left(x, p_d, P\right) dt, \tag{3.40}$$

$$\dot{x} - f\left(t, x, p_d, P\right) = 0, \quad f_{bc}\left(T, x(0), x(T), p_d, P\right) = 0. \tag{3.41}$$

Assuming that the distributions of the individual P_j are known, independent, and characterized by distribution parameters θ_j, it follows that:

$$\left\{P_j - V_i\left(\theta_j, \xi_j\right) = 0\right\}_{j=1}^{s} \tag{3.42}$$

for some transformation functions V_j and standardized random variables ξ_j. As a result, the response function r in (3.40) is an implicitly defined random variable. An example of (3.42) can be seen in (3.10) where $\theta_k = (\mu_k, \sigma_k)$.

As in Section 3.3.1, let M_j, $\left\{\xi_j^{(k_i)}\right\}_{k_j=1}^{M_j}$, and w_j be the integration order, standardized random variable quadrature nodes, and quadrature weights, respectively, for the stochastic parameter P_j. Applying the transformation (3.42) to each quadrature node results in $S = \sum_{j=1}^{s} M_j$ total equations:

$$\left\{\left\{P_j^{(k_j)} - V_j\left(\theta_j, \xi_j^{(k_j)}\right) = 0\right\}_{k_j=1}^{M_j}\right\}_{j=1}^{s}. \tag{3.43}$$

The Cartesian product of transformed quadrature node locations

$$\left\{P_1^{(k_1)}\right\}_{k_1=1}^{M_1} \times \cdots \times \left\{P_s^{(k_s)}\right\}_{k_s=1}^{M_s} \tag{3.44}$$

represent points in the parameter space where the response function must be evaluated and, consequently, the two-point boundary value problem (3.39) must be solved to construct the PCE. This can be structured into a single continuation problem by seeking M smooth curves that satisfy

$$\left\{\dot{x}^{(i)} - f\left(t, x^{(i)}, p_d, P^{(i)}\right) = 0\right\}_{i=1}^{M}, \tag{3.45}$$

$$\left\{f_{bc}\left(T, x^{(i)}(0), x^{(i)}(T), p_d, P^{(i)}\right) = 0\right\}_{i=1}^{M}. \tag{3.46}$$

The $P^{(i)}$ in (3.45) and (3.46) represent points in the Cartesian product (3.44).

The evaluation of the response function (3.38) on each trajectory segment is done with the following zero function

$$\left\{r_{\text{bc}}\left(T, x^{(i)}(0), x^{(i)}(T), p_d, P^{(i)}\right) + \int_0^T r_{\text{int}}\left(x^{(i)}, p_d, P^{(i)}\right) - r_i = 0\right\}_{i=1}^{M} \tag{3.47}$$

where the r_i are continuation variables that make up the entries of r_Ξ in (3.32). The equation

$$\alpha - \mathbf{\Psi} W r_\Xi = 0 \tag{3.48}$$

in the $N_t \times 1$ vector of continuation variables α imposes a constraint on the PCE coefficients. The equations

$$\alpha_0 - \mu_r = 0, \tag{3.49}$$

$$\left(\sum_{i=1}^{N_t - 1} \alpha_i^2\right) - \sigma_r^2 = 0 \tag{3.50}$$

associate the PCE coefficients with the mean and approximate variance of the response function. The equations

$$\mu_r - \rho_\mu = 0, \tag{3.51}$$

$$\sigma_r^2 - \rho_\sigma = 0 \tag{3.52}$$

imply that the continuation parameters ρ_μ and ρ_σ track the value of the mean and approximate variance of the response function.

3.6 The ‘uq’ toolbox

This section will discuss the `'uq'` toolbox implemented as part of this thesis. The toolbox provides constructors for two-point boundary value problems with parametric uncertainty that perform the following tasks:

1. Generate quadrature nodes and weights for user-specified values of maximum polynomial degree P_t and integration orders M_j, $j = 1..s$ for arbitrary stochastic dimension s.
2. For $s > 1$, construct the Cartesian product of (3.30) and tensor product of integration weights (3.31).
3. Construct the weighted matrix of basis polynomials $\mathbf{\Psi} W$ and store it for use during continuation.
4. Generate initial guesses for each sample trajectory through successive continuation runs.
5. Create gluing conditions for deterministic parameters.

6. Link stochastic parameters in each trajectory to the transformed standardized random variables.

7. Provide capability for automating the evaluation of a response function on each segment.

8. Construct a PCE for a given response function and determine its first two statistical moments.

3.6.1 A Generalized Constructor

The example problems in this thesis take advantage of two general purpose constructors, `uq_isol2bvp_sample` and `uq_BP2bvp`. The `uq_isol2bvp_sample` constructor takes an initial solution guess for a boundary value problem, generates trajectory segment instances for each integration node of the PCE, and then enforces necessary gluing conditions for deterministic parameters. The `uq_BP2bvp` takes an identified branch point from a previous continuation run and reinitializes the sample so that the initial tangent vectors are aligned to facilitate switching to the new branch. They take advantage of existing COCO toolbox constructors like `ode_coll2coll`, `ode_isol2bvp` and `ode_BP2coll` where appropriate. The `uq_isol2bvp_sample` constructor is shown here:

```
1  function prob = uq_isol2bvp_sample(prob, oid, varargin)
2
3    tbid = coco_get_id(oid, 'uq');
4    str  = coco_stream(varargin{:});
5    temp_str = coco_stream(varargin{:});
6    temp_prob2 = ode_isol2bvp(prob, tbid, str);
7    [args, opts] = uq_parse_str(str);
8    bvp_id = coco_get_id(tbid, 'bvp');
9    bc_data = coco_get_func_data(temp_prob2, bvp_id, 'data');
10
11   data = bvp_uq_init_data(bc_data, oid);
12   data = uq_init_data(prob, data, args, opts);
13   data = uq_bvp_gen_samples(data, prob, temp_str);
14
15   [prob, data] = uq_bvp_add_samples(prob, data, bc_data, args);
16   psi_mat = uq_make_psi_mat(data.nds_grid, data.uq.Pt, data.spdists);
17   data.wtd_psi_mat = psi_mat*diag(data.wts);
18
19   prob  = uq_add_sample_nodes(prob, data, args);
20
21 end
```

The constructor operates on a continuation problem structure `prob`, a string identifier `oid`, and a variable length sequence of additional arguments. It returns a modified continuation problem structure containing a collection of linked boundary-value problem instances. The data in `varargin` is stored in two variables as stream objects through the utility function `coco_stream`. The first, `str`, is used for the main problem construction task and the other, `temp_str`, is used in the successive continuation steps that generate trajectory segments in the `uq_bvp_gen_samples` function (described further in the next section). The function `uq_parse_str` (line 7 in `uq_isol2bvp_sample`) splits the PCE specific user inputs into `args` and `opts` data structures as shown below.

```
function [uqdata, opts] = uq_parse_str(str)

  grammar   = 'SPNAMES SPDIST SPDP [DPARNAMES] [OPTS]';
  args_spec = {
       'SPNAMES', 'cell', '{str}',    'spnames',       {}, 'read', {}
        'SPDIST', 'cell', '{str}',    'spdists',       {}, 'read', {}
          'SPDP',     '', '[num]',       'spdp',       {}, 'read', {}
     'DPARNAMES', 'cell', '{str}',   'dpdtpars',       {}, 'read', {}
    };

  opts_spec = {
    '-add-adjt', 'addadjt', false, 'toggle', {}
    };

  [uqdata, opts] = coco_parse(grammar, args_spec, opts_spec, str);

end
```

The field `spnames` stores a cell array of strings that correspond to parameter names designated as stochastic. The `spdists` field stores the distribution type specified for each stochastic parameter. Normal and uniform distributions are supported for the code in this thesis and are identified by a cell array containing entries of `'Normal'` and `'Uniform'`. The field `spdp` contains a numerical array of stochastic parameter distribution parameters. For normal random variables, the expected distribution parameters are the mean and standard deviation stored in a numerical array as follows: `[mu, sigma]`. For uniform random variables, the expected distribution parameters are a lower and upper limit stored in a numerical array as follows: `[lo, up]`. If multiple parameters are identified as stochastic, then the distribution parameter arrays must be vertically stacked, e.g., `[[mu1, sigma1];[mu2,sigma2]]`.

The final field `dpdtpars` identifies deterministic problem parameters that have been identified as varying between trajectory segments. They will be allowed to differ between segments, so no gluing conditions will be enforced on these parameters. An example of this type of parameter is the phase angle ϕ in (3.2). In order to ensure that $x_1(0)$ equals the maximal displacement for different values of K, the phase angle must be different for different trajectories.

The optional argument `-add-adjt`, when present, sets the field `addadjt` to true and causes the constructor to add the adjoint equations required for optimization during problem construction. This functionality triggers the execution of existing adjoint constructors available in COCO like `adjt_isol2coll`. Additional explanation is provided in Chapter 4.

The function `bvp_uq_init_data` (line 11 in `uq_isol2bvp_sample`) transfers data generated by the `ode_isol2bvp` constructor that is also necessary for the `uq_isol2bvp_sample` constructor to the `data` structure. The function `uq_init_data` (line 12 in `uq_isol2bvp_sample`) uses the `spnames`, `spdists`, and `spdp` fields to generate indices that separate deterministic parameters from stochastic parameters. Deterministic parameters are further separated between parameters shared by all segments and parameters unique to each segment (as identified through the `dpdtpars` field).

The maximum integration orders M_j for each stochastic dimension and maximum polynomial order P_t default to values of 4 (for all j) and 3, respectively, for the toolbox. However, they can be specified by the user through use of the `coco_set` function as follows:

```
>> prob = coco_set(prob, 'uq', 'M', M, 'Pt', Pt);
```

where the variables `M` and `Pt` store the desired values. If the stochastic dimension is greater than 1 and different integration orders are desired for each dimension, the variable `M` can be set as an array of integers. If `M` is a single integer, the integration orer is assumed be identical in all stochastic dimensions. This line of code must be executed before a call to the constructor.

The values of `M` and `Pt` along with the information in the `spnames`, `spdists`, and `spdp` fields are used by the `uq_init_data` function to generate standardized random variables, $\left\{\xi_i^{(k_i)}\right\}_{k_i=1}^{M_i}$, and integration weights, w_i, for each random variable. The standardized node locations for each random variable are stored in an array in the `st_nds` field of the `data` structure. The transformed node locations, $\left\{P_i^{(k_i)}\right\}_{k_i=1}^{M_i}$, are calculated and stored in the `nds` field of the `data` structure. The Cartesian product of transformed node locations given in (3.44) is generated and stored in the `nds_grid` field of the `data` structure. The tensor product of integration weights given in (3.31) is calculated and stored in the `wts` field of the `data` structure.

3.6.2 Sample Trajectory Generation

An initial solution guess must be provided to COCO at each integration node. Given an initial solution of the deterministic problem (3.39) for some choice of parameters, a solution at each integration node can be found through a series of continuation runs along 1-dimensional branches in the stochastic parameter space.

This is illustrated for a problem with $s = 2$ in Fig. 3.7. The tuple (p_1, p_2) represent a realization of the random variables P_1 and P_2 for which a solution to (3.39) is known. Solutions for each instance of the two-point boundary value problem represented by (3.45) and (3.46) can be generated from this initial solution. The function `uq_bvp_gen_samples` (line 13 in `uq_isol2bvp_sample`) performs this sequence of continuation runs.

Starting at (p_1, p_2), the arrows extending to the points marked by diamonds represent an initial continuation run in P_2 to the sample locations $\left\{P_2^{(k_2)}\right\}_{k_2=1}^{M_2}$. From each diamond, the horizontal arrows extending to the points marked by circles represent additional continuation runs in P_1 to the values $\left\{P_1^{(k_1)}\right\}_{k_1=1}^{M_1}$. The set of circles in Figure 3.7 cover all points in the Cartesian product given by (3.44). In a problem with higher stochastic dimension, the continuation runs would continue until all integration nodes in (3.44) were reached.

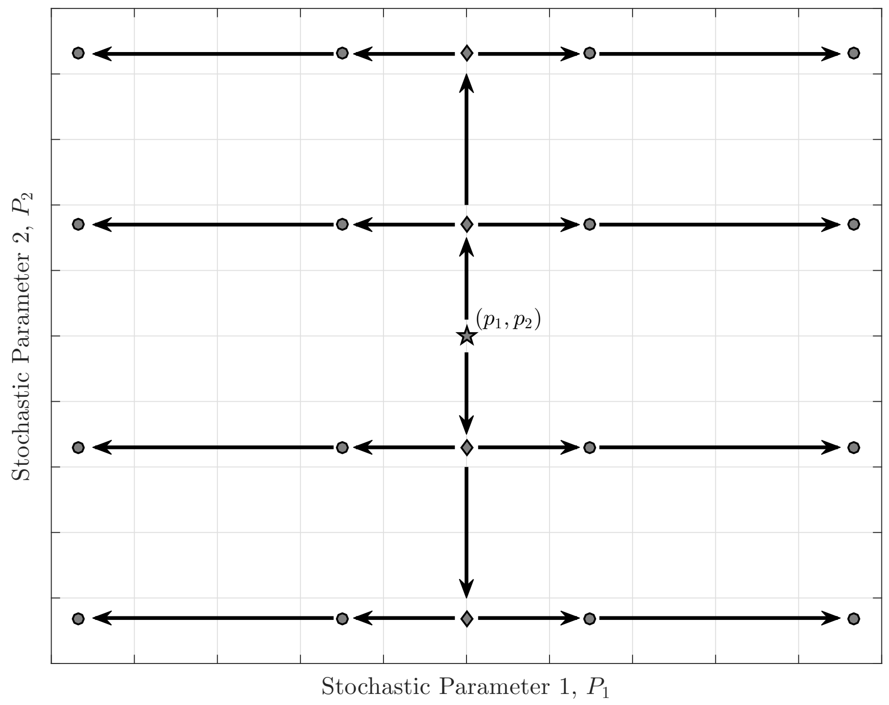

Figure 3.7: Successive continuation runs along 1-dimensional manifolds to generate initial solution guesses for the boundary-value problem associated with the integration nodes of the polynomial chaos expansion.

Because the stochastic parameters are treated as continuation variables, the sample of points in the stochastic parameter space will update (through (3.42)) if a distribution parameter θ is allowed to vary during continuation. Structuring the problem in this way allows continuation in the distribution parameters, and thus enables the successive parameter continuation optimization method in Chapter 2 to find optimal distribution parameters if desired (an example of this will be shown in the following chapter).

The function `uq_bvp_gen_samples` only generates the trajectory segments. The subsequent call to the function `uq_bvp_add_samples` (line 15 in `uq_isol2bvp_sample`) cycles through solution data created by the function `uq_bvp_gen_samples` and uses the existing `ode_coll2coll` constructor to add the sample trajectories to the problem instance. The function stores identifiers for each added collocation problem instance in the `sids` field of the `data` structure. The function `uq_bvp_add_samples` also closes each `'coll'` instance by applying the specified boundary conditions and introducing gluing conditions between the parameters for different segments.

The weighted matrix of orthogonal basis polynomial evaluations $\mathbf{\Psi}W$ is then generated and added to the `data` structure in the `wtd_psi_mat` field. This is done after the integration nodes are added because the order in which COCO reaches the integration nodes is not specified. The order of the trajectory segments is determined after the fact and the `wts` and `nds_grid` fields are reordered as necessary. This ensures proper alignment between rows in the $\mathbf{\Psi}$ matrix, the integration weights, and the response values calculated from trajectory segments. It is worth noting that for fixed integration order M and maximum polynomial order, P_t, the matrix product $\mathbf{\Psi}W$ does not change. Therefore, this calculation occurs during the problem construction and does not have to be performed again.

The final call in the constructor to `uq_add_sample_nodes` (line 19 in `uq_isol2bvp_sample`) adds the variable transformations defined in (3.42) to the continuation problem structure. The finalized data structure for the generated sample is stored with the zero function that enforces equality of the stochastic parameters in the collocation segments with the random variable locations that result from the transformation equation (3.43). Response functions that are later added to the continuation problem structure reference this data structure.

3.6.3 Adding Response Functions

The `uq` toolbox also includes the `uq_coll_add_response` function. The function takes in an existing continuation problem structure `prob`, a string identifier `oid` for a previously added collection of trajectories, a string identifier `rid` for the response function, and a variable number of additional arguments depending on the type of response function to be added. The constructor can add a response function of either of the forms r_{bc} or r_{int} in (3.47). Due to linearity of the expectation operator, the moments of a sum of the two types (like that given in (3.47)) could be evaluated by adding them together in a separate function after each has been added to the continuation problem structure.

The function returns a modified continuation problem instance that includes added zero functions for the response evaluations (3.47), polynomial chaos coefficient evaluations (3.48), and statistical moments of the response function (and corresponding inactive continuation parameters to track the values) (3.49-3.52). The source code for this function can be found in Appendix B.

3.6.4 Example Execution

This section show an example command line execution that uses the `uq` toolbox to find solutions to the example in Section 3.1 and generates frequency sweeps of the mean and variance, respectively. An initial solution guess is first generated using MATLAB's `ode45` solver. Then an empty continuation problem structure is initialized and various options are set with calls to `coco_set`.

```
>> p0 = [3;0;1];
>> [~, x0]   = ode45(@(t,x) linode(t, x, p0), [0 20*pi], [1; 0]);
>> [t0, x0]  = ode45(@(t,x) linode(t, x, p0), [0 2*pi], [x0(end,:)']);
>> prob = coco_prob();
>> prob = coco_set(prob, 'ode', 'autonomous', false);
>> prob = coco_set(prob, 'coll', 'NTST', 15, 'NCOL', 4);
>> prob = coco_set(prob, 'cont', 'PtMX', 1500, 'h', 0.5, 'h_max', 10);
```

The arguments required for initializing a boundary-value problem with the `'bvp'`-toolbox are provided in the same format already required by COCO.

```
>> coll_args = {@linode, @linode_dx, @linode_dp, @linode_dt, t0, x0, p0};
>> pnames = {'k','phi', 'om'};
>> bvp_args = {@fbc_x10, @Jbc_x10};
```

The `uq_args` variable below contains data required for the `uq_isol2bvp_sample` constructor. Note that ϕ (represented with parameter name `'phi'` in the script) is identified as a deterministic parameter whose value can differ between samples to allow for the satisfaction of a velocity constraint that ensures that the initial position corresponds to the maximal displacement for each individual sample.

```
>> uq_args = {{'k'}, {'Normal'}, [[3, 0.2]], {'phi'}};
>> prob = uq_isol2bvp_sample(prob, 'orig', coll_args{:}, pnames, bvp_args{:}, uq_args{:});
```

As shown in the screen extract below, the call to `uq_isol2bvp_sample` results in a continuation run to locate sample trajectories that will be used to construct the polynomial chaos expansion coefficients. The solutions of interest are labeled `UQ` in the output.

```
    STEP   DAMPING               NORMS              COMPUTATION TIMES
  IT SIT     GAMMA     ||d||     ||f||     ||U||   F(x)  DF(x)  SOLVE
   0                          2.00e-01  8.57e+00    0.0    0.0    0.0
   1   1  2.33e-01  2.06e+00  1.54e-01  8.58e+00    0.0    0.0    0.0
   2   1  3.27e-01  1.49e+00  1.04e-01  8.60e+00    0.0    0.0    0.0
   3   1  5.12e-01  9.66e-01  5.25e-02  8.61e+00    0.0    0.0    0.0
   4   1  1.00e+00  4.62e-01  7.22e-03  8.61e+00    0.0    0.0    0.0
   5   1  1.00e+00  2.34e-02  1.16e-07  8.60e+00    0.0    0.1    0.0
   6   1  1.00e+00  3.75e-07  1.52e-15  8.60e+00    0.0    0.1    0.0

 STEP      TIME        ||U||  LABEL  TYPE            k          phi
    0  00:00:00   8.5970e+00      1  EP      3.0000e+00   4.6365e-01
    3  00:00:00   8.6121e+00      2  UQ      2.8516e+00   4.9519e-01
    4  00:00:00   8.7387e+00      3  UQ      2.5331e+00   5.7797e-01
    4  00:00:00   8.7552e+00      4  EP      2.5078e+00   5.8562e-01

 STEP      TIME        ||U||  LABEL  TYPE            k          phi
    0  00:00:01   8.5970e+00      5  EP      3.0000e+00   4.6365e-01
    3  00:00:01   8.6046e+00      6  UQ      3.1484e+00   4.3564e-01
    4  00:00:01   8.6832e+00      7  UQ      3.4669e+00   3.8513e-01
    4  00:00:01   8.6962e+00      8  EP      3.5016e+00   3.8029e-01
```

Response functions evaluated on the generated sample are added with a call to the `uq_coll_add_response` function as shown below. This call establishes equations for evaluating PCE coefficients and the statistical moments of the expansion as well. Construction in this fashion allows for multiple response functions to be evaluated on a single sample.

```
>> prob = uq_coll_add_response(prob, 'orig', 'resp', 'bv', @x10, @x10_du, @x10_dudu);
```

The first argument to this function is the continuation problem structure, the second is the sample which the response function will be evaluated on, and the third is a name for the response function. The argument `'bv'` tells the constructor that the response function is evaluated on the boundary values of the trajectories, and the remaining arguments are the function handles for the response function and its first two derivatives.

Finally, the following call to the COCO entry point function results in a 1-dimensional manifold of solutions by releasing the forcing frequency. The inactive continuation parameters `orig.resp.mean` and `orig.resp.var` are also released and represent the mean and approximate variance of the response function added in the call to `uq_coll_add_response`.

```
>> uq_bd = coco(prob, 'freq_sweep', [], 1, {'om', 'orig.resp.mean', 'orig.resp.var'}, [0.001, 1000]);
```

The results are shown in Figs. 3.4, 3.5, and 3.6, which were created from data generated using this script.

3.7 Concluding Remarks

This chapter outlined the equations necessary for calculating approximations of the statistical moments of a response function evaluated on solutions to a finite-dimensional boundary value problem with parametric uncertainty using the method of Polynomial Chaos Expansions. The ability of the PCE to accurately approximate statistical moments was demonstrated using a linear ODE example. A toolbox capable of creating a sample of trajectory segments to construct the expansion as part of a numerical continuation scheme was also described.

Chapter 4

Optimization Under Uncertainty (OUU)

The successive parameter continuation optimization technique in Section 2.2 can be used to seek solutions to Robust Design Optimization (RDO) problems constrained by boundary-value problems by using the PCE formulation outlined in Section 3.5. This will be made apparent in this chapter through derivation of the necessary variations of a robust objective function and the PCE equations outlined in the previous chapter. An example problem seeking an optimal distribution parameter as part of a robust design optimization problem will be shown.

4.1 A Motivating Example

Consider again the harmonically forced linear oscillator with normally distributed stiffness given in (3.1)-(3.4) with fixed forcing frequency $\omega = 1$:

$$\dot{x}_1 = x_2, \tag{4.1}$$

$$\dot{x}_2 = \cos(t+\phi) - x_2 - Kx_1, \tag{4.2}$$

$$x_1(0) = x_1(2\pi), \tag{4.3}$$

$$x_2(0) = x_2(2\pi). \tag{4.4}$$

Let $K \sim \mathcal{N}\left(\mu_k, 0.2^2\right)$ and impose the condition that ϕ be chosen to ensure $x_2(0) = 0$. A value of μ_k is sought that maximizes the robust objective function

$$F = \mu_r - 3\sigma_r \tag{4.5}$$

where μ_r and σ_r are the mean and standard deviation of the response function

$$r = x_1(0). \tag{4.6}$$

As before, due to the imposed phase condition, the response function will correspond to the maximal displacement of the periodic trajectory.

From the earlier result in Chapter 3, the initial position of the trajectory is given by

$$x_{1,0}(K) = x_1(0, K) = \frac{1}{\sqrt{(K-1)^2+1}}, \tag{4.7}$$

where $\omega = 1$ has been substituted into (3.7). The objective function written in terms of the definitions of mean and variance is as follows

$$\begin{aligned} F(\mu_k) = {} & \int_{-\infty}^{\infty} x_{1,0}(k)\, \rho_K(k;\mu_k)\, dk \\ & - 3\sqrt{\left(\int_{-\infty}^{\infty} (x_{1,0}(k))^2\, \rho_K(k;\mu_k)\, dk - \left(\int_{-\infty}^{\infty} x_{1,0}(k)\, \rho_K(k;\mu_k)\, dk\right)^2\right)} \end{aligned} \tag{4.8}$$

where

$$\rho_K(k;\mu_k) = \frac{e^{\frac{-(k-\mu_k)^2}{2(0.2)^2}}}{\sqrt{2\pi(0.2)^2}} \tag{4.9}$$

is the probability density function for K. Using standard methods from calculus, a critical point of F with respect to μ_k occurs when $\frac{dF}{d\mu_k} = 0$. Using numerical quadrature the derivative of (4.8) has been evaluated several values for μ_k and the results plotted in Figure 4.1. The derivative crosses the horizontal axis at $\mu_k = 1$ with a negative slope indicating that the resulting stationary point is a maximum. Figure 4.2 shows the actual value of the robust objective function as a function of μ_k and confirms that the identified stationary point is a maximum of the robust objective function.

This results is also apparent by inspection of the integrands of the derivative $\frac{dF}{d\mu_k}$. The expression (4.7) is even about $K = 1$. The derivative of $\rho_K(k;\mu_k)$ with respect to μ_k

$$\frac{(k-\mu_k)}{(0.2)^3\sqrt{2\pi}} e^{\frac{-(k-\mu_k)^2}{2(0.2)^2}}, \tag{4.10}$$

is the product of an even exponential term and odd linear term (both about μ_k). Thus their product is odd about $k = 1$ when $\mu_k = 1$. The product of (4.7) and (4.10) is then odd when $\mu_k = 1$. As a result, the integrals

$$\int_{-\infty}^{\infty} x_{1,0}(k)\, \frac{d\rho_K}{d\mu_k} dk \tag{4.11}$$

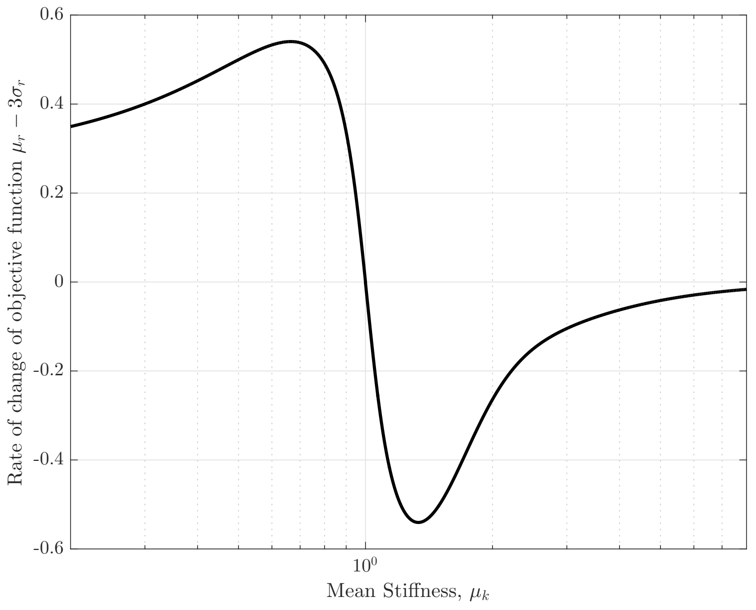

Figure 4.1: The derivative of the objective function (4.5) with respect to μ_k as a function of μ_k. The optimal mean stiffness is identified as 1.

and

$$\int_{-\infty}^{\infty} (x_{1,0}(k))^2 \frac{d\rho_K}{d\mu_k} dk \tag{4.12}$$

are zero when $\mu_k = 1$. At least one of (4.11) or (4.12) is present in every term of $\frac{dF}{d\mu_k}$ meaning that the derivative is zero when $\mu_k = 1$, confirming the result that the objective function is at a critical value when $\mu_k = 1$.

4.2 Robust Design Optimization with PCE

Consider the optimization problem

$$\begin{aligned} \text{minimize:} \quad & F\left(\mu_r, \sigma_r^2\right), \\ \text{subject to:} \quad & \dot{x}(t) - f\left(t, x(t), p, P\right) = 0, \\ & B\left(T, x(0), x(T), p, P\right) = 0, \\ & \int_0^T h\left(t, x(t), p, P\right) dt = 0, \\ & \left\{P_j - V_j\left(\theta_j, \xi_j\right) = 0\right\}_{j=1}^{s}, \end{aligned}$$

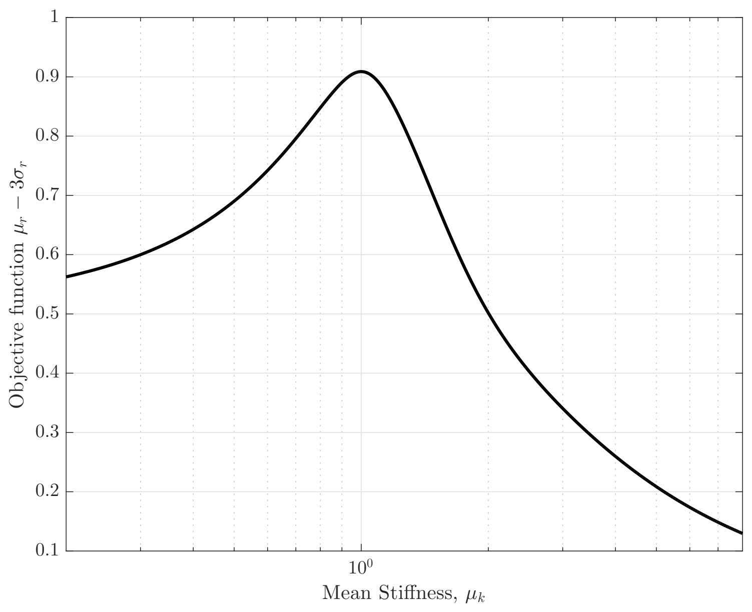

Figure 4.2: The robust objective function (4.5) value plotted against the value of μ_k.

where s is the number of stochastic variables in the problem. The robust objective function F is a function of the mean and standard deviation of the response function

$$r\left(T, x(t), p_d, P\right) = r_{\mathrm{bc}}\left(T, x\left(0\right), x\left(T\right), p_d, P\right)\right) + \int_0^T r_{\mathrm{int}}\left(x(t), p_d, P\right) dt. \tag{4.13}$$

In order to apply the PCE approximation to the above robust optimization problem, consider the expanded set of equations

$$\left\{\dot{x}^{(i)} - f\left(t, x^{(i)}(t), p_d, P^{(i)}\right) = 0\right\}_{i=1}^{M}, \tag{4.14}$$

$$\left\{B\left(T, x^{(i)}(0), x^{(i)}(T), p, P^{(i)}\right) = 0\right\}_{i=1}^{M}, \tag{4.15}$$

$$\left\{\int_0^T h\left(t, x^{(i)}(t), p, P^{(i)}\right) dt = 0\right\}_{i=1}^{M}, \tag{4.16}$$

$$\left\{\left\{P_j^{(i)} - V_j\left(\theta_j, \xi_j^{(i)}\right) = 0\right\}_{j=1}^{s}\right\}_{i=1}^{M}. \tag{4.17}$$

As before, the $P^{(i)}$ represent points in the Cartesian product (3.44). The $\xi^{(i)}$ are the corresponding points in the standardized random variable space. To simplify the representation of (4.17) in the adjoint formulation, let $\theta = (\theta_1, \ldots, \theta_s)$ and $V^*\left(\theta, \xi^{(i)}\right)$ be the vector-valued function whose j'th entry is equal to $V_j\left(\theta_j, \xi_j^{(i)}\right)$.

Equation (4.17) can then be rewritten as

$$\left\{P^{(i)} - V^*\left(\theta, \xi^{(i)}\right) = 0\right\}_{i=1}^{M} \tag{4.18}$$

Save for the introduction of new parameters θ these equations are equivalent to those for which variations were derived in the Chapter 2. The remaining equations to calculate the expansion coefficients and moment approximations are repeated here from Chapter 3:

$$\left\{r_{\text{bc}}\left(T, x^{(i)}(0), x^{(i)}(T), p_d, P^{(i)}\right) + \int_0^T r_{t,\text{int}}\left(x^{(i)}, p_d, P^{(i)}\right) - r_i = 0\right\}_{i=1}^{M}, \tag{4.19}$$

$$\alpha - \mathbf{\Psi} W r_{\Xi} = 0, \tag{4.20}$$

$$\alpha_0 - \mu_r = 0, \tag{4.21}$$

$$\left(\sum_{i=1}^{N_t-1} \alpha_i^2\right) - \sigma_r^2 = 0. \tag{4.22}$$

As in Chapter 3, the vectors r_{Ξ} and α in (4.2) are vectors whose entries are as follows

$$r_{\Xi} = \begin{pmatrix} r_1 \\ \vdots \\ r_M \end{pmatrix}, \qquad \alpha = \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_{N_t-1} \end{pmatrix}.$$

After a transformation of the time parameter ($\{t = T\tau : \tau \in [0,1]\}$) that was also done in Chapter 2 and addition of continuation variables μ_1, μ_2, μ_3, μ_4, and μ_5 to track the values of the robust objective function, problem parameters p_d, distribution parameters θ, response function mean, and response function variance, respectively, the Lagrangian corresponding to (4.14)-(4.22) equals

$$\begin{aligned}
&\mathcal{L}\left(T, \tilde{x}^{(1)}(\tau), \ldots, \tilde{x}^{(M)}(\tau), p_d, P, \theta, \mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \tilde{\lambda}_1^{(1)}(\tau), \ldots, \tilde{\lambda}_1^{(M)}(\tau),\right. \\
&\left.\lambda_2^{(1)}, \ldots \lambda_2^{(M)}, \lambda_3^{(1)}, \ldots, \lambda_3^{(M)}, \lambda_4^{(1)}, \ldots, \lambda_4^{(M)}, \lambda_5^{(1)}, \ldots, \lambda_5^{(M)}, \lambda_6, \lambda_7, \lambda_8, \eta_1, \eta_2, \eta_3, \eta_4, \eta_5\right) = \\
&\mu_1 + \sum_{i=1}^{M} \left(\int_0^1 \left(\tilde{\lambda}_1^{(i)}(\tau)\right)^T \left(\tilde{x}'^{(i)}(\tau) - T\, f\left(T\tau, \tilde{x}^{(i)}(\tau), p_d, P^{(i)}\right)\right) d\tau \right. \\
&+ \left(\lambda_2^{(i)}\right)^T B\left(T, \tilde{x}^{(i)}(0), \tilde{x}^{(i)}(1), p_d, P^{(i)}\right) + \lambda_3^{(i)} \left(T \int_0^1 h\left(T\tau, \tilde{x}^{(i)}(\tau), p_d, P^{(i)}\right) d\tau\right) \\
&+ \lambda_4^{(i)} \left(r_{\text{bc}}\left(T, \tilde{x}^{(i)}(0), \tilde{x}^{(i)}(1), p_d, P^{(i)}\right) + T \int_0^1 r_{\text{int}}\left(\tilde{x}^{(i)}(\tau), p_d, P^{(i)}\right) d\tau - r_i\right) \\
&\left. + \left(\lambda_5^{(i)}\right)^T \left(P^{(i)} - V^*\left(\theta, \xi^{(i)}\right)\right) \right) + \lambda_6^T \left(\alpha - \mathbf{\Psi} W r_{\Xi}\right) + \lambda_7 \left(\alpha_0 - \mu_r\right) + \lambda_8 \left(\sum_{i=1}^{N_t - 1} \alpha_i^2 - \sigma_r^2\right) \\
&+ \eta_1 \left(F\left(\mu_r, \sigma_r^2\right) - \mu_1\right) + \eta_2^T \left(p_d - \mu_2\right) + \eta_3^T \left(\theta - \mu_3\right) + \eta_4 \left(\mu_r - \mu_4\right) + \eta_5 \left(\sigma_r^2 - \mu_5\right).
\end{aligned} \tag{4.23}$$

Variations are taken with respect to each argument. Note that the $\xi^{(i)}$'s are fixed for a given problem. Thus they do not appear in the list of arguments for the Lagrangian and no variations need to be taken with respect to them. The variations with respect to the M $\lambda_1^{(i)}(\tau)$'s result in the following M expressions:

$$\left\{ \int_0^1 \left(\delta\lambda_1^{(i)}(\tau)\right)^T \left(\tilde{x}'^{(i)}(\tau) - T\, f\left(T\tau, \tilde{x}^{(i)}(\tau), p_d, P^{(i)}\right)\right) d\tau \right\}_{i=1}^{M}. \tag{4.24}$$

The variations with respect to the M $\lambda_2^{(i)}$'s result in the following M expressions:

$$\left\{ \left(\delta\lambda_2^{(i)}\right)^T \left(B\left(T, \tilde{x}^{(i)}(0), \tilde{x}^{(i)}(1), p_d, P^{(i)}\right)\right) \right\}_{i=1}^{M}. \tag{4.25}$$

The variations with respect to the M $\lambda_3^{(i)}$'s result in the following M expressions:

$$\left\{ \delta\lambda_3^{(i)} \left(T \int_0^1 h\left(T\tau, \tilde{x}^{(i)}(\tau), p_d, P^{(i)}\right) d\tau\right) \right\}_{i=1}^{M}. \tag{4.26}$$

The variations with respect to the M $\lambda_4^{(i)}$'s result in the following M expressions:

$$\left\{ \delta\lambda_4^{(i)} \left(r_{\text{bc}}\left(T, \tilde{x}^{(i)}(0), \tilde{x}^{(i)}(1), p_d, P^{(i)}\right) + T \int_0^1 r_{\text{int}}\left(\tilde{x}^{(i)}(\tau), p_d, P^{(i)}\right) d\tau - r^{(i)}\right) \right\}_{i=1}^{M}. \tag{4.27}$$

The variation with respect to the M $\lambda_5^{(i)}$'s result in the following M expressions:

$$\left\{ \left(\delta\lambda_5^{(i)}\right)^T \left(P^{(i)} - V^*\left(\theta, \xi^{(i)}\right)\right) \right\}_{i=1}^{M}. \tag{4.28}$$

The variation with respect to λ_6 is given by

$$\delta\lambda_6^T \left(\alpha - \mathbf{\Psi} W r\right). \tag{4.29}$$

The variation with respect to λ_7 is given by

$$\delta\lambda_7 \left(\alpha_0 - \mu_r\right). \tag{4.30}$$

The variation with respect to λ_8 is given by

$$\delta\lambda_8 \left(\sum_{i=1}^{N_t - 1} \alpha_i^2 - \sigma_r^2 \right). \tag{4.31}$$

The variation with respect to η_1 is given by

$$\delta\eta_1 \left(F\left(\mu_r, \sigma_r^2\right) - \mu_1\right). \tag{4.32}$$

The variation with respect to η_2 is given by

$$\delta\eta_2^T \left(p_d - \mu_2\right). \tag{4.33}$$

The variation with respect to η_3 is given by

$$\delta\eta_3^T \left(\theta - \mu_3\right). \tag{4.34}$$

The variation with respect to η_4 is given by

$$\delta\eta_4 \left(\mu_r - \mu_4\right). \tag{4.35}$$

The variation with respect to η_5 is given by

$$\delta\eta_5 \left(\sigma_r^2 - \mu_5\right). \tag{4.36}$$

As in Chapter 2, setting the coefficients of the variations with respect to the Lagrange multipliers given in (4.24)-(4.36) equal to zero returns the original system.

Variations with respect to the variables in the original system are now considered. Variations with respect to the M $\tilde{x}'^{(i)}(\tau)$'s result in M equations. As was done in Chapter 2, integration by parts is used on variations with respect to $\tilde{x}'^{(i)}(\tau)$ and the result is substituted into the appropriate locations:

$$\left\{\int_0^1 \left(\tilde{\lambda}_1^{(i)}(\tau)\right)^T \delta\tilde{x}'^{(i)}(\tau)\,d\tau\right\}_{i=1}^M = \\ \left\{\left(\tilde{\lambda}_1^{(i)}(1)\right)^T \delta\tilde{x}^{(i)}(1) - \left(\tilde{\lambda}_1^{(i)}(0)\right)^T \delta\tilde{x}^{(i)}(0) - \int_0^1 \left(\tilde{\lambda}_1'^{(i)}(\tau)\right)^T \delta\tilde{x}^{(i)}(\tau)\,d\tau\right\}_{i=1}^M. \quad (4.37)$$

The variations with respect to the M $\tilde{x}^{(i)}(\tau)$'s result in the following M expressions:

$$\left\{\int_0^1 \left(-\left(\tilde{\lambda}_1'^{(i)}(\tau)\right)^T - T\left((\tilde{\lambda}_1^{(i)}(\tau)\right)^T f_{,\tilde{x}^{(i)}}\left(T\tau, \tilde{x}^{(i)}(\tau), p_d, P^{(i)}\right)\right. \\ \left. + T\lambda_3^{(i)} h_{,\tilde{x}^{(i)}}\left(T\tau, \tilde{x}^{(i)}(\tau), p_d, P^{(i)}\right) + T\lambda_4^{(i)} r_{\text{int},\tilde{x}^{(i)}}\left(\tilde{x}^{(i)}(\tau), p_d, P^{(i)}\right)\delta x^{(i)}(\tau)\right)d\tau\right\}_{i=1}^M \quad (4.38)$$

where the coefficients of $\delta\tilde{x}^{(i)}(\tau)$ in (4.37) have been inserted. The result is M ODEs in the $\tilde{\lambda}_1^{(i)}(\tau)$ variables. The variations with respect to the M $\tilde{x}^{(i)}(0)$'s result in the following M expressions:

$$\left\{\left(-\left(\tilde{\lambda}_1^{(i)}(0)\right)^T + \left(\tilde{\lambda}_2^{(i)}\right)^T B_{,\tilde{x}^{(i)}(0)}\left(T, \tilde{x}^{(i)}(0), \tilde{x}^{(i)}(1), p_d, P^{(i)}\right)\right.\right. \\ \left.\left. + \lambda_4^{(i)} r_{\text{bc},\tilde{x}^{(i)}(0)}\left(T, \tilde{x}^{(i)}(0), \tilde{x}^{(i)}(1), p_d, P^{(i)}\right)\right)\delta\tilde{x}^{(i)}(0)\right\}_{i=1}^M \quad (4.39)$$

where the coefficient of $\delta\tilde{x}^{(i)}(0)$ in (4.37) has been inserted. These represent M boundary conditions for the ODEs in (4.38). The variations with respect to the M $\tilde{x}^{(i)}(1)$'s result in the following M expressions:

$$\left\{\left(\left(\tilde{\lambda}_1^{(i)}(1)\right)^T + \left(\tilde{\lambda}_2^{(i)}\right)^T B_{,\tilde{x}^{(i)}(1)}\left(T, \tilde{x}^{(i)}(0), \tilde{x}^{(i)}(1), p_d, P^{(i)}\right)\right.\right. \\ \left.\left. + \lambda_4^{(i)} r_{\text{bc},\tilde{x}^{(i)}(1)}\left(T, \tilde{x}^{(i)}(0), \tilde{x}^{(i)}(1), p_d, P^{(i)}\right)\right)\delta\tilde{x}^{(i)}(1)\right\}_{i=1}^M \quad (4.40)$$

where the coefficient of $\tilde{x}^{(i)}(1)$ in (4.37) has been inserted. These represent M additional boundary conditions for the ODEs in (4.38). The variations with respect to T gives the following equations

$$\begin{aligned}\Bigg(\sum_{i=1}^{M}\Bigg(\int_0^1\Bigg(-\left(\tilde{\lambda}_1^{(i)}(\tau)\right)^T\left(f\left(T\tau,\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)+T\tau f_{,t}\left(T\tau,\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)\right)\\ +\lambda_3^{(i)}\left(h\left(T\tau,\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)+T\tau h_{,t}\left(T\tau,\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)\right)\\ +\lambda_4^{(i)}r_{\text{int}}\left(T\tau,\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)\Bigg)d\tau\Bigg)+\left(\tilde{\lambda}_2^{(i)}\right)^T B_{,T}\left(T,\tilde{x}^{(i)}(0),\tilde{x}^{(i)}(1),p_d,P^{(i)}\right)\\ +\lambda_4^{(i)}r_{\text{bc},T}\left(T,\tilde{x}^{(i)}(0),\tilde{x}^{(i)}(1),p_d,P^{(i)}\right)\Bigg)\delta T.\end{aligned} \tag{4.41}$$

The variations with respect to p_d gives the following equations

$$\begin{aligned}\Bigg(\eta_2^T+\sum_{i=1}^{M}\Bigg(T\int_0^1\Bigg(-\left(\tilde{\lambda}_1^{(i)}(\tau)\right)^T f_{,p_d}\left(T\tau,\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)\\ +\lambda_3^{(i)}h_{,p_d}\left(T\tau,\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)+\lambda_4^{(i)}r_{\text{int},p_d}\left(\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)\Bigg)d\tau\\ +\left(\tilde{\lambda}_2^{(i)}\right)^T B_{,p_d}\left(T,\tilde{x}^{(i)}(0),\tilde{x}^{(i)}(1),p_d,P^{(i)}\right)+\lambda_4^{(i)}r_{\text{bc},p_d}\left(T,\tilde{x}^{(i)}(0),\tilde{x}^{(i)}(1),p_d,P^{(i)}\right)\Bigg)\Bigg)\delta p_d.\end{aligned} \tag{4.42}$$

The variations with respect to the M $P^{(i)}$'s gives the following M equations

$$\begin{aligned}\Bigg\{\Bigg(T\int_0^1\Bigg(-\left(\tilde{\lambda}_1^{(i)}(\tau)\right)^T f_{,P}\left(T\tau,\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)\\ +\lambda_3^{(i)}h_{,P}\left(T\tau,\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)+\lambda_4^{(i)}r_{\text{int},P}\left(\tilde{x}^{(i)}(\tau),p_d,P^{(i)}\right)\Bigg)d\tau\\ +\left(\tilde{\lambda}_2^{(i)}\right)^T B_{,P}\left(T,\tilde{x}^{(i)}(0),\tilde{x}^{(i)}(1),p_d,P^{(i)}\right)+\lambda_4^{(i)}r_{\text{bc},P}\left(T,\tilde{x}^{(i)}(0),\tilde{x}^{(i)}(1),p_d,P^{(i)}\right)+\lambda_5^{(i)}\Bigg)\delta P^{(i)}\Bigg\}_{i=1}^{M}.\end{aligned} \tag{4.43}$$

The variation with respect to θ is given by

$$\left(\eta_3^T-\sum_{i=1}^{M}V^*_{,\theta}\left(\theta,\xi^{(i)}\right)\lambda_5^{(i)}\right)\delta\theta. \tag{4.44}$$

For the variations with respect to the expansion coefficients α, first define the vector $v\in\mathbb{R}^{N_t}$ and matrix $V\in\mathbb{R}^{N_t\times N_t}$ as follows:

$$v=\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix},\qquad V=\begin{pmatrix}0&0&\dots&0\\0&1&\dots&0\\\vdots&\vdots&\ddots&\vdots\\0&0&\dots&1\end{pmatrix}. \tag{4.45}$$

The variations with respect to α can then be written as

$$\left(\lambda_6^T + \lambda_7 v + 2\lambda_8 \alpha^T V \alpha\right) \delta\alpha. \tag{4.46}$$

The variation with respect to r_Ξ is given by

$$\left(-\lambda_4^T - \lambda_6^T \mathbf{\Psi} W\right) \delta r_\Xi, \tag{4.47}$$

where $\lambda_4 \in \mathbb{R}^M$ is a column vector whose i'th entry is $\lambda_4^{(i)}$. The variation with respect to μ_r is given by

$$\left(\eta_1 F_{\mu_r} - \lambda_7 + \eta_4\right) \delta\mu_r. \tag{4.48}$$

The variation with respect to σ_r^2 is given by

$$\left(\eta_1 F_{\sigma_r^2} - \lambda_8 + \eta_5\right) \delta\sigma_r^2. \tag{4.49}$$

Additionally, the variations with respected to μ_1, μ_2, and μ_3 are given by $(1 - \eta_1)\,\delta\mu_1$, $\eta_2^T \delta\mu_2$, $\eta_3^T \delta\mu_3$, $\eta_4^T \delta\mu_4$and $\eta_5^T \delta\mu_5$. The adjoint system is constructed by setting the values of the coefficients of (4.38)-(4.49) equal to zero.

The PCE-specific equations do not change the approach to seeking stationary values. The trivial solution for the Lagrange multipliers still satisfies the adjoint equations, so starting from an initial solution of the original system of equations and the trivial solution of the adjoint system, continuation is performed in pursuit of a fold point for μ_1 (which is located approximately by COCO). The fold point will also approximately coincide with the location of a branch point identified by COCO. From the branch point, continuation is performed along a secondary branch until η_1 equals 1. Along this secondary branch, the values of the Lagrange multipliers are able to take on nontrivial values. In subsequent continuation steps, the value of η_1 remains at 1 and elements of η_2, η_3, η_4, and η_5 are successively driven to zero.

4.3 Adjoint Construction

The addition of the adjoints for the equations that make up a sample of trajectory segments in the `'uq'` toolbox is triggered by the inclusion of the `'-add-adjt'` optional argument in the constructor call. The optional argument triggers straightforward calls to the `coco_add_adjt` function and existing constructors of relevant toolboxes (namely `'coll'` and `'bvp'`). The following snippet of code is taken from a portion of the constructor code that handles addition of deterministic parameters and shows a representative example for how adjoints are added.

```
 1 if data.addadjt
 2    adj_s_idxi = adj_s_idx{i};
 3    if isfield(args, 'run') && isfield(args, 'lab')
 4       chart = coco_read_solution(args.run, args.lab, 'chart');
 5       cdata = coco_get_chart_data(chart, 'lsol');
 6       [chart, lidx] = coco_read_adjoint(sfid, args.run, args.lab, 'chart', 'lidx');
 7       l0 = chart.x;
 8       if add_tl0
 9          tl0 = cdata.v(lidx);
10       end
11    else
12       l0 = zeros([fxd_p_idx,1]);
13    end
14    if add_tl0
15       prob = coco_add_adjt(prob, sfid, 'aidx', [adj_s_idx1(fxd_p_idx); adj_s_idxi(fxd_p_idx)], ...
16          'l0', l0, 'tl0', tl0);
17    else
18       prob = coco_add_adjt(prob, sfid, 'aidx', [adj_s_idx1(fxd_p_idx); adj_s_idxi(fxd_p_idx)], ...
19          'l0', l0);
20    end
21 end
```

A few things are worth pointing out in the above excerpt. First, the optional argument `'-add-adjt'` sets the `addadjt` field of `data` to true and thus results in the execution of the above snippet. Second, if the arguments passed to the function that contains this code include a previous run name and label (stored in `args.run` and `args.label`), the constructor will use these arguments to read data from the previous run and initialize the adjoint variables (lines 3-10). If these arguments are not included, the adjoint variables are left to their default values of zero (line 12), and the candidate tangent direction is not passed to the adjoint constructor. If the previous run represents a branch point, the `add_tl0` flag is set to 1 and the tangent direction is also read and eventually passed to the adjoint constructor (lines 8-9 and 14-16). The purpose of doing so is to facilitate branch switching, a critical part of the optimization method outlined in Chapter 2. The above code makes use of the `coco_read_solution`, `coco_get_chart_data`, and `coco_read_adjoint` utility functions available in COCO for collecting the necessary data.

A call to the function `uq_coll_add_response_adjoint` adds the adjoint equations for response functions. It takes in a continuation problem structure `prob`, a string identifier `oid` for a collection of trajectory segments, and a string identifier `rid` for an existing response function. It optionally take two additional arguments: a previous continuation run name and number. If these are passed, the data is read from the previous run and the tangent directions of the Lagrange multipliers are oriented to facilitate branch switching, as necessary. If these are not passed, the values are initialized to zero. The function returns the modified continuation problem instance `prob` with adjoint equations added for the response function, its expansion coefficients, and its statistical moments.

As a final observation related to the adjoint construction, as shown in Chapter 2, the sequence of continuation runs to satisfy the first-order optimality conditions require that certain Lagrange multipliers be released during continuation. The equations added for uncertainty quantification will increase the number of Lagrange multipliers for the stochastic parameters. For stochastic parameters, there will be a Lagrange multiplier for each distribution parameter (the η_3's in the general formulation), so one must keep this in mind when releasing parameters.

4.4 Linear Example

The following numerical example shows how to use the successive continuation method for identifying an optimal distribution parameter. Consider again the harmonically forced, damped, linear oscillator with normally distributed stiffness K from earlier in this chapter.

$$\dot{x}_1 = x_2, \tag{4.50}$$

$$\dot{x}_2 = \cos(t+\phi) - x_2 - Kx_1, \tag{4.51}$$

$$x_1(0) = x_1(2\pi), \tag{4.52}$$

$$x_2(0) = x_2(2\pi). \tag{4.53}$$

Recall that while the variance is specified, the mean of the stiffness is considered variable $K \sim \mathcal{N}\left(\mu_k, 0.2^2\right)$. The phase variable ϕ will again be assigned to ensure that the initial position corresponds to the maximal displacement for each individual trajectory. The robust objective function is given by

$$F\left(\mu_r, \sigma_r^2\right) = \mu_r - 3\sigma_r, \tag{4.54}$$

where

$$r = x_1(0), \tag{4.55}$$

and seeks to maximize the lower bound of a 3σ confidence interval for the maximal displacement.

4.4.1 Numerical Simulation

The initial command line sequence is identical to the example in Section 3.6.4 and will not be shown again. The assignment shown below of the arguments for the `'uq'` toolbox, however, highlights the addition of the optional argument `'-add-adjt'` described in Section 4.3.

```
>> uq_args = {{'k'}, {'Normal'}, [[3, 0.2]], {'phi'}, '-add-adjt'};
>> prob = uq_isol2bvp_sample(prob_init, 'orig', coll_args{:}, pnames, bvp_args{:}, uq_args{:});
```

The following commands add the sample and response function as before and add the adjoint of the response function.

```
>> prob = uq_coll_add_response(prob, 'orig', 'resp', 'bv', @x10, @x10_du, @x10_dudu);
>> prob = uq_coll_add_response_adjoint(prob, 'orig', 'resp');
```

Finally, the robust objective function is added by first collecting the necessary indices for the mean and variance variables and then adding `rdo` as an inactive monitor function in line with the requirements for the adjoint optimization. The adjoint indices are then collected and the `coco_add_adjt` constructor is called to finish the problem definition.

```
>> response_id = coco_get_id('orig', 'uq', 'responses');
>> mean_idx = coco_get_func_data(prob, coco_get_id(response_id, 'resp', 'mean'), 'uidx');
>> var_idx = coco_get_func_data(prob, coco_get_id(response_id, 'resp', 'variance'), 'uidx');
>> prob = coco_add_func(prob, 'rdo', @rdo, @rdo_dU, @rdo_dUdU, [], 'inactive', 'rdo', 'uidx', ...
[mean_idx;var_idx]);
>> mean_aidx = coco_get_adjt_data(prob, coco_get_id(response_id, 'resp', 'mean'), 'axidx');
>> var_aidx = coco_get_adjt_data(prob, coco_get_id(response_id, 'resp', 'variance'), 'axidx');
>> prob = coco_add_adjt(prob, 'rdo', 'd.rdo', 'aidx', [mean_aidx;var_aidx]);
```

The call to the COCO entry point function then seeks a fold point in the objective function which will also be a branch point allowing for switching to a secondary branch with non-trivial Lagrange multipliers.

```
>> uq_bd = coco(prob, 'rdo', [], 1, {'rdo', 'mu.k', 'd.rdo', 'orig.resp.mean', 'orig.resp.var', ...
'd.om', 'd.sig.k'}, {[],[1e-3,1e3]});
```

The following partial output from the continuation sequence shows the location of the fold point and branch point. The continuation parameters `orig.resp.mean`, `orig.resp.var`, `d.om`, and `d.sig.k` are also required to be released to allow for a 1-dimensional solution manifold, but their columns are not shown.

```
    STEP   DAMPING               NORMS              COMPUTATION TIMES
  IT SIT     GAMMA     ||d||     ||f||     ||U||   F(x)  DF(x)  SOLVE
   0                           1.93e-14  1.88e+01    0.0    0.0    0.0

 STEP      TIME        ||U||  LABEL  TYPE            rdo         mu.k        d.rdo
    0  00:00:00   1.8847e+01      1  EP      3.4059e-01   3.0000e+00   0.0000e+00
   10  00:00:03   2.4552e+01      2          1.8933e-01   5.5068e+00   0.0000e+00
   20  00:00:07   8.6930e+01      3          4.4215e-02   2.2979e+01   0.0000e+00
   30  00:00:10   1.8642e+02      4          2.0275e-02   4.9705e+01   0.0000e+00
   40  00:00:12   2.8627e+02      5          1.3151e-02   7.6431e+01   0.0000e+00
   49  00:00:15   3.7438e+02      6  EP      1.0039e-02   1.0000e+02   0.0000e+00

 STEP      TIME        ||U||  LABEL  TYPE            rdo         mu.k        d.rdo
    0  00:00:15   1.8847e+01      7  EP      3.4059e-01   3.0000e+00   0.0000e+00
   10  00:00:18   2.0775e+01      8          6.4639e-01   1.5916e+00   0.0000e+00
   20  00:00:21   2.1709e+01      9          8.9996e-01   1.0704e+00   0.0000e+00
   22  00:00:23   2.1717e+01     10  BP      9.0961e-01   1.0000e+00   0.0000e+00
   22  00:00:23   2.1717e+01     11  FP      9.0961e-01   9.9995e-01   0.0000e+00
   30  00:00:25   2.1230e+01     12          7.5878e-01   6.3168e-01   0.0000e+00
   33  00:00:27   2.0870e+01     13  EP      6.9014e-01   5.0000e-01   0.0000e+00
```

A second continuation problem instance is initiated from the approximate branch point by a call to the constructor `uq_BP2bvp` which, like its counterparts `ode_BP2coll` and `ode_BP2coll`, aligns the initial predictor guess for the equations it adds to facilitate switching branches as mentioned in Section 4.3. The `'-add-adjt'` optional argument is again passed to include the adjoint equations in the problem instance. The optional argument `'-add-resp'` tells the constructor to include previously added response functions to the continuation problem structure.

```
>> prob = prob_init;
>> prob = uq_BP2bvp(prob, 'orig', 'rdo', BPLab(1), '-add-adjt', '-add-resp');
```

The robust objective function is manually added back the continuation problem structure. Its adjoint is also manually added by reading the solution data for initial conditions and tangent vectors then providing this information to the `coco_add_adjt` constructor.

```
>> mean_idx = coco_get_func_data(prob, coco_get_id(response_id, 'resp', 'mean'), 'uidx');
>> var_idx = coco_get_func_data(prob, coco_get_id(response_id, 'resp', 'variance'), 'uidx');
>> prob = coco_add_func(prob, 'rdo', @rdo, @rdo_dU, @rdo_dUdU, [], 'inactive', 'rdo', ...
'uidx', [mean_idx;var_idx]);
>> mean_aidx = coco_get_adjt_data(prob, coco_get_id(response_id, 'resp', 'mean'), 'axidx');
>> var_aidx = coco_get_adjt_data(prob, coco_get_id(response_id, 'resp', 'variance'), 'axidx');
>> chart = coco_read_solution('rdo', BPLab(1), 'chart');
>> cdata = coco_get_chart_data(chart, 'lsol');
>> [chart, lidx] = coco_read_adjoint('rdo', 'rdo', BPLab(1), 'chart', 'lidx');
>> l0 = chart.x;
>> tl0 = cdata.v(lidx);
>> prob = coco_add_adjt(prob, 'rdo', 'd.rdo', 'aidx', [mean_aidx;var_aidx], 'l0', l0, 'tl0', tl0);
```

The call shown below to the COCO entry point function results in a sequence of continuation steps whose terminal point satisfies the first-order necessary conditions. The COCO output is again truncated.

```
>> uq_bd = coco(prob, 'rdo.switch', [], 1, {'d.rdo', 'mu.k', 'rdo', 'orig.resp.mean', ...
'orig.resp.var', 'd.om', 'd.sig.k'}, [0,1]);

STEP      TIME        ||U||  LABEL  TYPE         d.rdo         mu.k          rdo
   0  00:00:00   2.1717e+01      1  EP      0.0000e+00   1.0000e+00   9.0962e-01
  10  00:00:01   4.4429e+01      2          6.1761e-01   1.0000e+00   9.0962e-01
  13  00:00:02   6.6409e+01      3  EP      1.0000e+00   1.0000e+00   9.0962e-01
```

As can be seen from the output, the satisfaction of the first-order necessary conditions results in an optimal mean stiffness of $\mu_k = 1$ as anticipated from the earlier analytical results.

4.5 Concluding Remarks

This chapter derived first-order optimality conditions for a system of equations that included a PCE approximation of the statistical moments of a response function. The conditions did not materially deviate from the format in Chapter 2, and so the successive continuation method for seeking optimal solutions was still valid. The method of constructing the adjoint equations for the `'uq'` toolbox was briefly discussed, and an example script was shown that located a stationary point of a robust objective function with respect to a distribution parameter of a stochastic input parameter. The following chapter takes the method outlined in this chapter and applies it to a nonlinear example.

Chapter 5

OUU for a Duffing Oscillator

This chapter considers a robust design optimization problem for a harmonically forced Duffing oscillator. It is well-known that for sufficiently large forcing amplitude the harmonically excited Duffing oscillator exhibits hysteresis in its steady-state frequency response as shown in Figure 5.1. As a result, the Duffing oscillator has been used as a lumped parameter model for certain classes of MEMS sensors that exhibit hysteretic frequency responses [26]. Though recent studies have sought to exploit the dynamics of nonlinear systems [27, 28], in many cases it is still generally desirable to try and mitigate the effects of nonlinearity. In [29], the transition amplitude to bistability for a beam system with hardening nonlinearity was sought by approximately locating fold points with pairs of linked orbits and driving their frequency difference to a sufficiently small value. A similar procedure will be used here to locate an approximate cusp for the Duffing oscillator. The robust design optimization method from Chapter 4 will then be used to locate a combination of mass and stiffness that maximizes a function of the mean and standard deviation of the response amplitude in the presence of an uncertain nonlinear stiffness. During the optimization procedure, the forcing amplitude will be adjusted to keep the frequency response curve at the approximate transition to bistability.

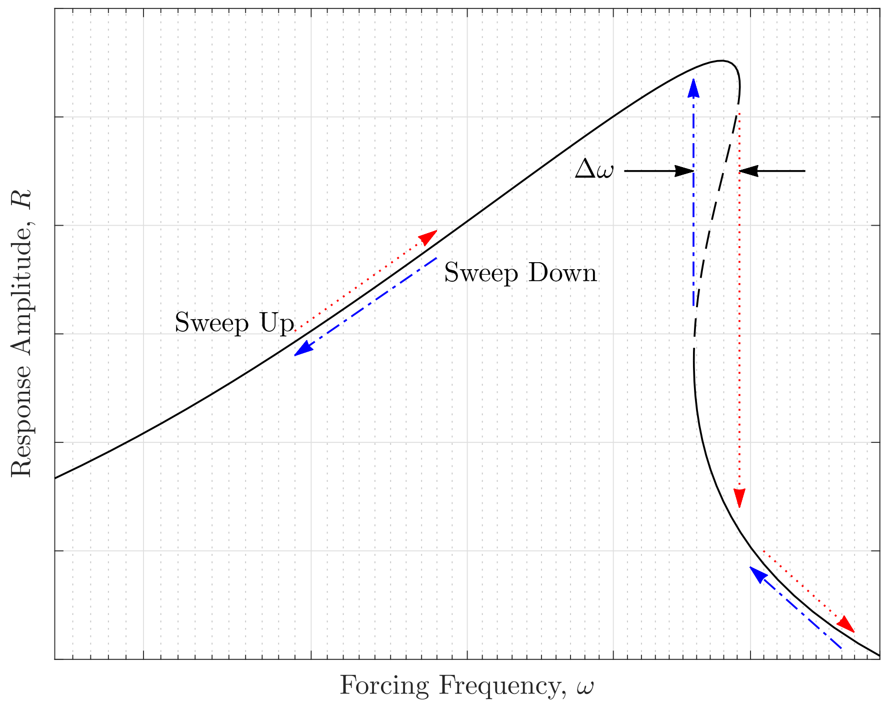

Figure 5.1: Hysteresis in the frequency response of a Duffing oscillator with a hardening nonlinearity.

5.1 Problem Formulation

The steady state dynamics of the Duffing oscillator are described by the following differential equation and periodic boundary conditions:

$$\dot{x}_1 = x_2, \tag{5.1}$$

$$\dot{x}_2 = \frac{1}{m}\left(A\cos\left(x_3\right) - cx_2 - kx_1 - \alpha x_1^3\right), \tag{5.2}$$

$$\dot{x}_3 = \omega, \tag{5.3}$$

$$x_1\left(2\pi\right) = x_1\left(0\right), \tag{5.4}$$

$$x_2\left(2\pi\right) = x_2\left(0\right), \tag{5.5}$$

$$x_3\left(2\pi\right) = x_3\left(0\right) + 2\pi. \tag{5.6}$$

The phase condition

$$x_2\left(0\right) = 0 \tag{5.7}$$

ensures that the initial position of the periodic orbit is an extremum. For sufficiently small values of the forcing amplitude, the frequency response curve will be single-valued. The objective function, will seek to maximize the resonant peak of the response amplitude R while staying (approximately) at the transition point to bistability through an appropriate choice of A, k and m. For the purpose of illustration, the linear stiffness and mass parameters are constrained to lie on a circle of radius ρ centered at (k_0, m_0)

$$(k - k_0)^2 + (m - m_0)^2 = \rho^2. \tag{5.8}$$

Due to compactness of the set of possible k and m, the Extreme Value Theorem ensures that a maximum and minimum exist somewhere along the circle.

Following the example in [29], the fold points and resonant peak of the frequency response curve are approximated using pairs of linked orbits. The expanded system

$$\dot{x}_{1,i} = x_{2,i}, \tag{5.9}$$

$$\dot{x}_{2,i} = \frac{1}{m}\left(A\cos\left(x_{3,i}\right) - cx_{2,i} - kx_{1,i} - \alpha x_{1,i}^3\right), \tag{5.10}$$

$$\dot{x}_{3,i} = \omega_i, \tag{5.11}$$

$$x_{1,i}\left(2\pi\right) = x_{1,i}\left(0\right), \tag{5.12}$$

$$x_{2,i}\left(2\pi\right) = x_{2,i}\left(0\right), \tag{5.13}$$

$$x_{3,i}\left(2\pi\right) = x_{3,i}\left(0\right) + 2\pi, \tag{5.14}$$

$$x_{2,i}\left(0\right) = 0, \tag{5.15}$$

for $i = 1 \ldots 5$ allows for two orbits to approximate the response peak and three to approximate the pair of fold points. Imposing

$$\omega_1 - \omega_2 = \epsilon_\omega, \tag{5.16}$$

$$x_1\left(0\right) - x_2\left(0\right) = 0, \tag{5.17}$$

allows for $x_{1,1}(0)$ to approximate the resonant peak for sufficiently small ϵ_ω. The constraints

$$\omega_3 - \omega_4 = 0, \tag{5.18}$$

$$\omega_4 - \omega_5 = 0, \tag{5.19}$$

$$x_3(0) - x_4(0) = \epsilon_R, \tag{5.20}$$

$$x_4(0) - x_5(0) = \epsilon_R. \tag{5.21}$$

manipulate the shape of the frequency response curve by forcing orbits 3-5 to share the same frequency but only have a small difference ϵ_R in their initial position. This condition is satisfied during continuation by allowing the forcing amplitude to vary. The intent is that this will keep the system near the transition to a bistable frequency response curve. However, reducing ϵ_R and ϵ_ω degrades the conditioning of the numerical problem, so, as a practical matter, ϵ_R and ϵ_ω cannot be made arbitrarily small. Additionally, without a constraint on the curvature of the response curve at these points, it is possible that the true fold points could still be significantly separated. No additional constraints are added to address this, rather, it will be a known deficiency and the shape of the response curve will be checked visually.

A parametric uncertainty is introduced by treating the damping coefficient c as a uniform random variable with lower limit c_l and upper limit c_u:

$$c \sim \mathcal{U}(c_l, c_u). \tag{5.22}$$

Using the PCE approximation outlined in Chapter 3, this means that there will be M samples of the system (5.9)-(5.15) each with identical parameters save for c which will instead be replaced by c_k for the k'th sample and A whose value will adjust to satisfy the approximate cusp conditions in (5.20) and (5.21).

The random variable passed to the PCE basis functions must be in a standard form $\xi_c \sim \mathcal{U}(-1,1)$. The transformation

$$c = \frac{1}{2}((c_u + c_l) + (c_u - c_l)\,\xi_c) \tag{5.23}$$

in terms of the standardized uniform variable returns the non-standard random variable c. Following the guidance from Table 3.1, Gauss-Legendre integration nodes on the interval $[-1, 1]$ will be used to construct the PCE. The deterministic objective function

$$r = x_{1,1}(0) \tag{5.24}$$

will be evaluated at each sample. Extremal values of the robust objective function

$$F = \mu_r - \sigma_r \tag{5.25}$$

are sought, where μ_r and σ_r are the mean and standard deviation of the response function, respectively. As outlined in Chapter 3, these statistical moments will be approximated through the construction of a PCE.

5.2 `'uq'`-Toolbox Limitations

The constructors used in Chapter 3 and 4 allow for samples of the five periodic orbits in (5.9)-(5.15) to be added to the same continuation problem through repeated calls to `uq_isol2bvp` or `uq_bvp2bvp`. However, a difficulty arises when imposing constraints between the multiple orbits that make up a single sample. Namely, the forcing amplitude must be different at each sample to satisfy (5.20) and (5.21). They do not fit into the class of dependent parameters discussed in Chapter 3 because the constraints that drive the difference between samples are external to the boundary-value problem definition. Addressing this deficiency in the existing toolbox is left to future development.

As a result, at each stage of the optimization process, the periodic orbits, parameter constraints, and adjoints will be added through direct calls to existing COCO functions and toolbox constructors like `ode_isol2bvp`, `coco_add_glue`, and `adjt_isol2bvp`. Some of the utility functions embedded in the `'uq'` constructors are used in this example. Namely, the same data structure used in the constructor will be generated and used to generate integration nodes, c_k, and link these nodes to distribution parameters. Additionally, the PCE statistical moment approximations will be calculated using functions developed for the `'uq'` toolbox.

5.3 Problem Initialization

The specific parameters values chosen for this numerical experiment are summarized in Table 5.1. The values for A, m, and k are merely starting points and vary during continuation. Their values in the table represent those used to generate the initial periodic solution for the Duffing oscillator.

Figure 5.2 shows graphically the sequence of continuation runs used to achieve the starting point for optimization under uncertainty. Starting from an initial solution that satisfies the differential equation and periodic boundary constraints (Fig. 5.2a), a continuation run in the initial velocity is executed with a single orbit until the phase condition (5.7) is satisfied (Fig. 5.2b). Second, a frequency sweep is performed and, using the `coco_add_event` function, two points with equal amplitude are labeled `PK` and three points with

Parameter	Value	Parameter	Value
m	1	k	2
α	0.5	A	0.5
c_l	0.1	c_u	0.5
m_0	2	k_0	2
ρ	1	ϵ_ω	1e-2
ϵ_R	1e-2		

Table 5.1: Parameter Values

equal frequency encompassing both fold points are labeled `OM` (Fig. 5.2c). The results from this frequency sweep are used in three different continuation runs.

The first run approximates the resonant peak. Orbits labeled `PK` are added to a single continuation problem, the constraint (5.17) is added as a zero function, and (5.16) is added as a monitor function with an associated inactive continuation parameter ϵ_ω. Gluing conditions are applied to enforce equality of the parameters in the two orbits (except forcing frequency). Continuation is performed in ϵ_ω until its value matches that given in Table 5.1 (Fig. 5.2d).

The second run approximately locates the top fold point. The orbits labeled `OM` are added to a continuation problem, constraints (5.18) are added as zero functions, and a monitor function is added for (5.20) with the associated inactive continuation parameter $\epsilon_{R,1}$. Gluing conditions are applied to enforce equality of the parameters in the three orbits. Continuation is performed in $\epsilon_{R,1}$ (also releasing the shared forcing frequency) until the value of $\epsilon_{R,1}$ matches the value of ϵ_R given in Table 5.1 (Fig. 5.2e).

The third run collects all five orbits (`PK` and `OM`) into the same continuation problem, adding (5.16)-(5.20) (the values of ϵ_ω and $\epsilon_{R,1}$ are fixed in this run). A monitor function with inactive continuation parameter $\epsilon_{R,2}$, and gluing conditions are added to enforce equality of the parameters in the five orbits (ω_1, ω_2, and ω_3 are not glued together). Continuation is performed in $\epsilon_{R,2}$ and the parameters A, ω_1, and ω_3 are released to allow for the satisfaction of the applied constraints on the problem (Fig. 5.2f). Figure 5.2g shows the resulting five orbits (solid gray markers) that will act as approximations to the true peak and fold points. The figure also shows that while the curvature of the frequency response curve was not constrained, the separation of the true fold points appears sufficiently small.

5.4 Robust Design Optimization

After the completion of the runs in the previous section, all five orbits satisfying (5.8)-(5.21) are collected into a single problem instance for the robust design optimization. The information required for uncertainty quantification is generated by sorting arguments that were stored in the `uq_args` variable in Section 3.6.4

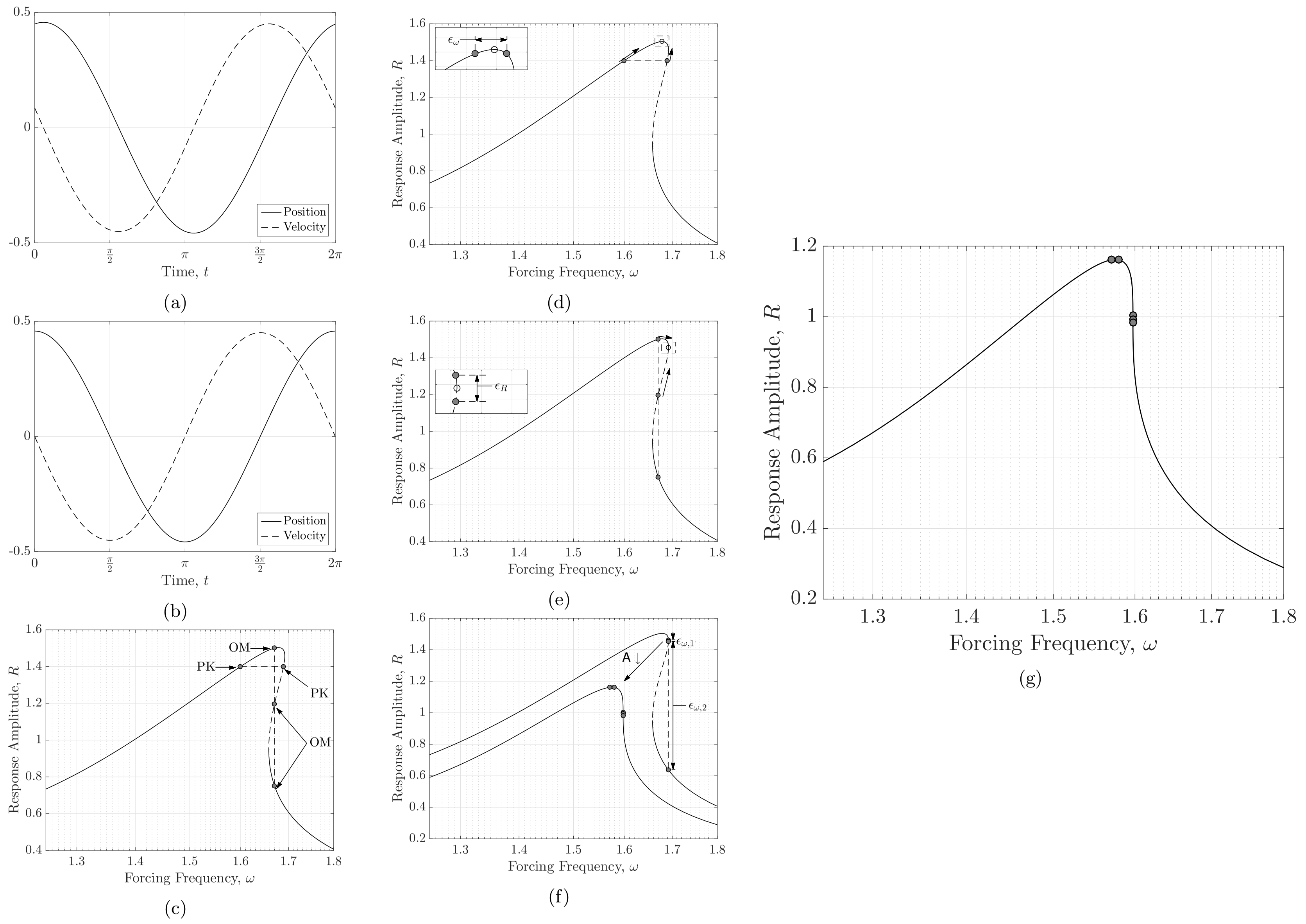

Figure 5.2: Graphical sequence of continuation runs used to generate the the starting point for the Robust Design Optimization. See Section 5.3 for a description of the sequence.

between an arguments data structures, `args`, and options data structures `opts`.

```
>> args = struct();
>> args.spdists = {'Uniform'};
>> args.spnames = {'c'};
>> args.spdp = [0.1, 0.5];
>> args.dpdtpars = {'A'};
>> opts=struct();
>> opts.addadjt = 1;
```

The data structure for the PCE creation (in particular the integration nodes for the damping coefficient) is generated by the `uq_init_data` function in the following segment of code.

```
>> bvp_id = coco_get_id('PK1', 'bvp');
>> bc_data = coco_get_func_data(prob, bvp_id, 'data');
>> uq_data = bvp_uq_init_data(bc_data, 'PK1');
>> uq_data = uq_init_data(prob, uq_data, args, opts);
```

A one dimensional continuation run is executed in c and the solution points corresponding to the integration nodes (stored in `nds` field of the `uq_data` structure) are marked with the label `UQ` by the `coco_add_event` function.

```
>> prob = coco_add_event(prob, 'UQ', 'c', uq_data.nds);
>> bd = coco(prob, run_name, [], 1, {'c', 'om', 'A'},[0.1,0.5]);
```

The below extract shows the result of starting the run from a nominal value of 0.2 for c with A and ω_1 released.

```
STEP      TIME        ||U||  LABEL  TYPE            c           om            A
   0  00:00:00   2.7071e+03      1  EP     2.0000e-01   1.5701e+00   3.6408e-01
  10  00:00:02   2.7072e+03      2         1.6561e-01   1.5412e+00   2.6609e-01
  20  00:00:04   2.7073e+03      3  UQ     1.2777e-01   1.5100e+00   1.7438e-01
  20  00:00:04   2.7073e+03      4         1.2610e-01   1.5086e+00   1.7072e-01
  28  00:00:06   2.7074e+03      5  EP     1.0000e-01   1.4873e+00   1.1781e-01

STEP      TIME        ||U||  LABEL  TYPE            c           om            A
   0  00:00:06   2.7071e+03      6  EP     2.0000e-01   1.5701e+00   3.6408e-01
  10  00:00:07   2.7071e+03      7  UQ     2.3200e-01   1.5973e+00   4.6796e-01
  10  00:00:07   2.7071e+03      8         2.3526e-01   1.6001e+00   4.7922e-01
  20  00:00:09   2.7070e+03      9         2.7829e-01   1.6373e+00   6.4048e-01
  30  00:00:10   2.7070e+03     10         3.2142e-01   1.6753e+00   8.2590e-01
  40  00:00:11   2.7069e+03     11         3.6426e-01   1.7138e+00   1.0348e+00
  41  00:00:12   2.7069e+03     12  UQ     3.6800e-01   1.7171e+00   1.0542e+00
  50  00:00:13   2.7069e+03     13         4.0656e-01   1.7524e+00   1.2663e+00
  60  00:00:14   2.7069e+03     14         4.4813e-01   1.7910e+00   1.5196e+00
  66  00:00:15   2.7069e+03     15  UQ     4.7223e-01   1.8138e+00   1.6787e+00
  70  00:00:16   2.7069e+03     16         4.8886e-01   1.8295e+00   1.7939e+00
  73  00:00:16   2.7069e+03     17  EP     5.0000e-01   1.8402e+00   1.8736e+00
```

Next, the labeled solutions are extracted from the previous run and added to a new continuation problem instance (again with the required parameter gluing conditions and constraints). During problem construction, the indices for the $x_1(0)$ are collected in variables `r_idx` for passing to the response function. Additionally, at this stage of the problem construction, the adjoint equations are added to the system with calls to either `coco_add_adjt` or through an appropriate adjoint constructor for the `'bvp'` toolbox.

After the orbits are added, the following sequence of code uses information stored in the previously constructed data structure `uq_data` to build the weighted matrix of basis polynomial evaluations, $\mathbf{\Psi}W$, shown in (3.35) (the full source code for the `uq_make_psi_mat` is available in Appendix B).

In the following code snippet, the variable indices for the parameters in one of the peak approximating orbits (which have been stored during problem construction) are added to the `uq_data` structure in the `s_idx` field. The parameter adjoint indices were similarly collected and are stored in the `adj_s_idx` field. The `uq_add_sample_nodes` function then uses these indices to handle the addition of the parameters gluing conditions and the linking of PCE integration nodes to the distribution parameters α_l and α_u described in (3.42).

```
>> psi_mat = uq_make_psi_mat(uq_data.nds_grid, uq_data.uq.Pt, uq_data.spdists);
>> uq_data.wtd_psi_mat = psi_mat*diag(uq_data.wts);
>> uq_data.s_idx = pk1_par_idx;
>> uq_data.adj_s_idx = pk1_par_aidx;
>> uq_data.addadjt=1;
```

The response function (5.24) is added to the problem by first generating a vector of initial response function values and a vector of indices of the response peak approximant. The initial response function values are pre-multiplied by $\mathbf{\Psi}W$ to generate an initial guess for the PCE coefficients. The adjoint is added after collecting appropriate adjoint indices. Initial guesses for the Lagrange multipliers are zero.

```
>> igs = zeros(uq_data.nsamples, 1);
>> idx = zeros(1, uq_data.nsamples);
>> for i=1:uq_data.nsamples
       igs(i) = r_igs{i};
       idx(i) = [r_idx{i}];
   end
>> alpha_ig = uq_data.wtd_psi_mat*igs;
>> response_id = 'resp';
>> prob = coco_add_func(prob, response_id, @resp_pce, @resp_pce_dU, @resp_pce_dUdU, ...
   uq_data, 'zero', 'uidx', idx(:), 'u0', alpha_ig);
>> aidx = zeros(1, uq_data.nsamples);
>> for i=1:uq_data.nsamples
       aidx(i) = [r_aidx{i}];
   end
>> prob = coco_add_adjt(prob, response_id, 'aidx', aidx(:), 'l0', zeros(size(alpha_ig)));
```

The initial guesses for the PCE coefficients along with the indices for the coefficient variables are also used when adding the mean and variance functions below. The source code for the mean and variance functions `uq_pce_mean` and `uq_pce_var` and their respective derivatives are provided in Appendix B. The adjoints use the corresponding adjoint indices and again have zeros for the initial guesses.

```
>> mean_id = coco_get_id(response_id, 'pce_mean');
>> prob = coco_add_func(prob, mean_id, @uq_pce_mean, @uq_pce_mean_dU, ...
   @uq_pce_mean_dUdU, uq_data, 'zero', 'uidx', alpha_idx, 'u0', alpha_ig(1));
>> prob = coco_add_adjt(prob, mean_id, 'aidx', alpha_aidx, 'l0', 0);
>> var_id = coco_get_id(response_id, 'pce_variance');
>> prob = coco_add_func(prob, var_id, @uq_pce_variance, @uq_pce_variance_dU, ...
   @uq_pce_variance_dUdU, uq_data, 'zero', 'uidx', alpha_idx, 'u0', sum(alpha_ig(2:end).^2));
>> prob = coco_add_adjt(prob, var_id, 'aidx', alpha_aidx, 'l0', 0);
```

The objective function (5.25) and its adjoint are added in the following extract.

```
>> obj_id = 'obj';
>> mean_idx = coco_get_func_data(prob, mean_id, 'uidx');
>> var_idx = coco_get_func_data(prob, var_id, 'uidx');
>> prob = coco_add_func(prob, obj_id, @obj, @obj_du, @obj_dudu, [], ...
>> 'inactive', obj_id, 'uidx', [mean_idx(end); var_idx(end)]);
>> dobj_id = coco_get_id('d', obj_id);
>> mean_aidx = coco_get_adjt_data(prob, mean_id, 'axidx');
>> var_aidx = coco_get_adjt_data(prob, var_id, 'axidx');
>> prob = coco_add_adjt(prob, obj_id, dobj_id, 'aidx', [mean_aidx(end); var_aidx(end)]);
```

In the initial phase of the optimization, a fold point is sought in the robust objective function value along the k-m constraint circle. To that end, a 1-dimensional continuation run is initiated by releasing the inactive continuation parameter corresponding to the robust objective function and its adjoint and the parameters k and m (to allow for satisfaction of constraint 5.8). Additionally, the forcing amplitudes $A^{(k)}$ (the forcing amplitude of the k'th sample in the PCE) and the forcing frequency for the first and third periodic orbits ($\omega_1^{(k)}$ and $\omega_3^{(k)}$, the approximate peak and cusp frequency in the k'th sample, respectively) are also released to allow for satisfaction of the constraints on the frequency response curve. The Lagrange multipliers for the inactive parameters (except for k and m) are also released. The result of the continuation run is shown graphically in Figure 5.3 and in the below screen output from COCO.

```
    STEP   DAMPING               NORMS              COMPUTATION TIMES
  IT SIT     GAMMA     ||d||     ||f||     ||U||   F(x)  DF(x)  SOLVE
   0                          8.08e-07  5.41e+03    0.1    0.0    0.0

 STEP      TIME        ||U||  LABEL  TYPE            obj            k            m        d.obj
    0  00:00:49   5.4141e+03      1  EP      1.1025e+00   2.0000e+00   1.0000e+00   0.0000e+00
   10  00:02:49   5.4135e+03      2          9.3722e-01   1.3063e+00   1.2797e+00   0.0000e+00
   20  00:05:12   5.4137e+03      3          7.4766e-01   1.0972e+00   2.4300e+00   0.0000e+00
```

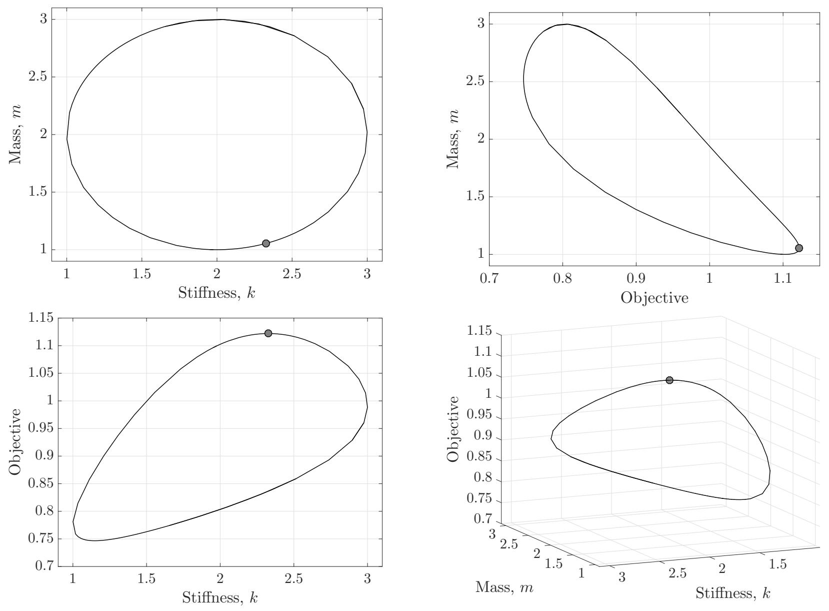

Figure 5.3: The result of the 1-D continuation run in the objective function. The fold point corresponding to the maximum value is marked by the gray filled circle in each figure.

```
  26  00:07:16   5.4138e+03      4  BP      7.4668e-01   1.1487e+00   2.5247e+00   0.0000e+00
  26  00:07:45   5.4138e+03      5  FP      7.4668e-01   1.1487e+00   2.5247e+00   0.0000e+00
  30  00:08:48   5.4138e+03      6          7.4712e-01   1.1903e+00   2.5869e+00   0.0000e+00
  40  00:10:53   5.4140e+03      7          7.5584e-01   1.3893e+00   2.7919e+00   0.0000e+00
  50  00:13:00   5.4145e+03      8  EP      8.5908e-01   2.5151e+00   2.8572e+00   0.0000e+00

STEP      TIME        ||U||  LABEL  TYPE           obj            k            m        d.obj
   0  00:13:35   5.4141e+03      9  EP      1.1025e+00   2.0000e+00   1.0000e+00   0.0000e+00
  10  00:15:41   5.4143e+03     10          1.1209e+00   2.2508e+00   1.0320e+00   0.0000e+00
  20  00:17:29   5.4144e+03     11          1.1219e+00   2.3104e+00   1.0494e+00   0.0000e+00
  24  00:19:13   5.4144e+03     12  BP      1.1219e+00   2.3280e+00   1.0553e+00   0.0000e+00
  24  00:19:42   5.4144e+03     13  FP      1.1219e+00   2.3280e+00   1.0553e+00   0.0000e+00
  30  00:21:18   5.4144e+03     14          1.1212e+00   2.3936e+00   1.0807e+00   0.0000e+00
  40  00:23:37   5.4146e+03     15          1.0395e+00   2.9416e+00   1.6633e+00   0.0000e+00
  50  00:26:01   5.4142e+03     16  EP      7.7428e-01   1.6551e+00   2.9387e+00   0.0000e+00
```

Two approximate fold points are marked `FP` along the sweep. The marked points also correspond with approximate branch points marked `BP`. The marked points coincide with the objective function minimum and maximum along the closed k-m circle.

From the fold point that corresponds with the maximum, a new continuation run is initiated to drive the value of the Lagrange multiplier associated with objective function (the η_1 variable from Chapter 2) from 0 to 1. The problem construction is nearly identical to the setup for the initial sweep in the fold parameters. The key difference is that the tangent direction for the adjoint functions are collected from the previous run and the new run uses these values as the initial direction for continuation. The goal is to converge to a secondary manifold where the Lagrange multipliers are able to take on nonzero values. The screen output below shows the result of this run. The Lagrange multiplier associated with the objective function is driven

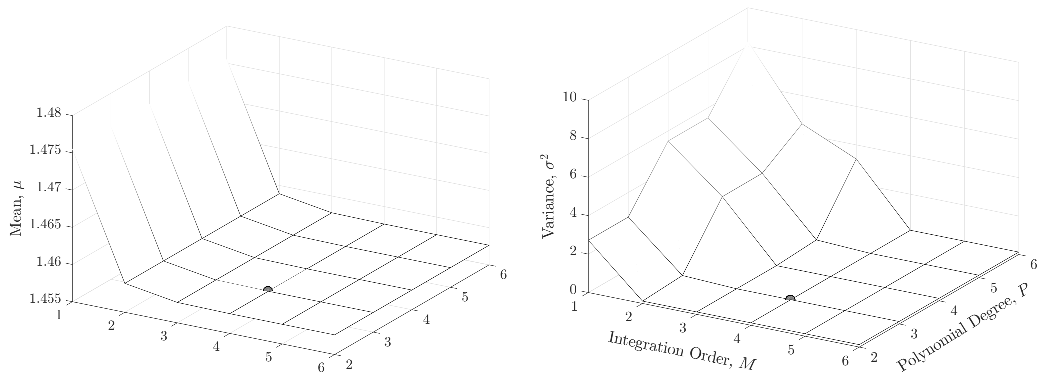

Figure 5.4: Convergence of the mean and variance of the response function for varying integration order and polynomial degree

to one while the Lagrange multiplier corresponding to the nonlinear stiffness `d.alpha` changes to nontrivial values during continuation.

```
STEP       TIME        ||U||  LABEL  TYPE         d.obj            k            m          obj
   0  00:01:36   5.4144e+03      1  EP      0.0000e+00   2.3280e+00   1.0553e+00   1.1219e+00
  10  00:03:49   5.4148e+03      2          7.2560e-02   2.3280e+00   1.0553e+00   1.1219e+00
  15  00:05:42   5.4949e+03      3  EP      1.0000e+00   2.3280e+00   1.0553e+00   1.1219e+00


STEP     d.alpha
   0  0.0000e+00
  10  8.1372e-02
  15  1.1214e+00
```

Since the nonlinear stiffness α is not allowed to vary in this example, its corresponding Lagrange multiplier need not be driven to zero. Therefore, with the value of Lagrange multipliers for k and m at zero (they were never released, and therefore remain at zero) and `d.obj` at 1, the first order optimality conditions are satisfied and the value of the objective function is thus at a locally optimal value.

5.5 Convergence of the PCE

The results in the preceding sections were arrived at using an integration order M of 4 and polynomial degree P_t of 3 (see Section 3.2). It is good practice to confirm that these values result in sufficient convergence of the statistical moments. To check this, the initial value of the mean and variance will be checked at all combinations of $M = 1 \ldots 6$ and $P_t = 2 \ldots 6$. $P_t = 1$ is not tested as the variance requires at least two terms in the expansion to be calculated. The results of this evaluation are shown in Figure 5.4. The mean value converges quickly and is independent of the value of P_t (the mean only use the first expansion coefficient). The variance plot shows that for $M < 4$ the numerical error associated with an insufficient quadrature rule causes poor results for $P_t > 2$. The gray dots in both sub-figures of Figure 5.4 are at $(M, P_t) = (4, 3)$.

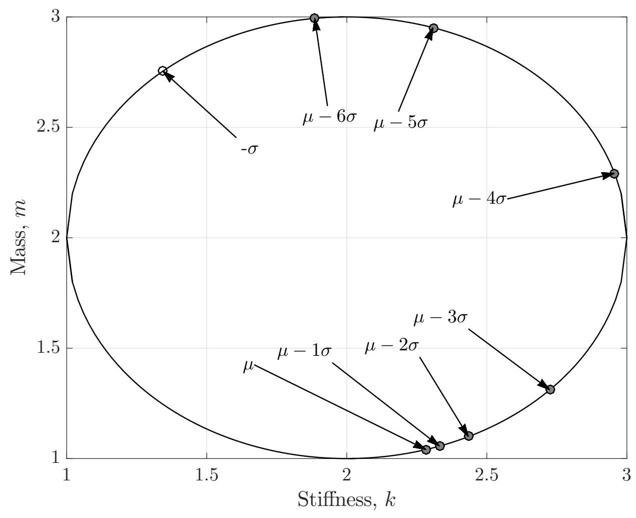

Figure 5.5: Maximum value of the objective function 5.26 for $n = 0 \ldots 6$. The point of minimum variance of the response function (i.e. the maximum of $-\sigma$) is marked with an open circle

5.6 The effect of Variance on the Robust Optimal solution

One can imagine situations where a designer is more risk averse to variation in a performance metric and may want to maximize the lower bound of a higher sigma level robust objective function (5.25). This section will briefly show the result of considering the following alternative objective function

$$F = \mu_r - n\sigma_r \tag{5.26}$$

for sigma levels of $n = 0 \ldots 6$. Figure 5.5 shows the location of the maximum value of the objective function for progressively increasing values of n. The values for the objective function at these locations are also summarized in Table 5.2. For $n = 0$ (i.e. only the mean value is considered), the maximum lies in the lower

n	0	1	2	3	4	5	6
$\mu - n\sigma$	1.48	1.12	0.77	0.42	0.12	-0.14	-0.39

Table 5.2: Maximum Values for progressively increasing n

right of the circle, and increasing the value of n shifts the maximum counterclockwise around the constraint circle. As n increases, the maximum value of the objective function approaches the location of minimum variance which is marked with an open circle in Figure 5.5.

5.7 Concluding Remarks

This chapter demonstrated the use of the optimization under uncertainty formulation outlined in Chapter 4 for a more complicated problem for which the developed toolbox constructors were not well-suited. Instead of a single boundary-value problem, each sample represents five linked periodic orbits. This gives a nod to more complicated problems that the tools developed for this thesis could be used to solve. Additionally, convergence of the statistics that were used in the robust design optimization function was demonstrated. A brief investigation into the effect of varying sigma levels for the robust objection function was also shown.

Chapter 6

Conclusions and Future Work

The work outlined in this document provides a framework for performing uncertainty quantification as part of a numerical continuation routine using Polynomial Chaos Expansion. An implementation has been created for use with the continuation package COCO in MATLAB. Additionally, the uncertainty quantification has been combined with an existing optimization technique to allow for solving Robust Design Optimization problems with dynamic constraints. Also of note is that due to the use of the PCE technique to approximate the mean and variance, those parameters can be directly manipulated by the continuation algorithm as was done in pursuit of fold points in Chapters 4 and 5. Finally, this work has shown that the developed toolbox matches analytical results in a linear case and has shown an example application to the approximate analysis of a cusp bifurcation for a nonlinear oscillator. The library (and all code used to run examples in this thesis) is available on the University of Illinois gitlab server in the following repository: `https://gitlab.engr.illinois.edu/jcandrsn/coco_uq_toolbox` (see the `'thesis'` branch of the repository).

6.1 Future Work

The code developed as part of this thesis represents a working toolbox capable of solving a variety of problems. However, there are, of course, plenty of areas for improvement and further development. A few of these areas of improvement are listed here.

6.1.1 Distribution Types

The current toolbox supports the use of random variables that follow normal and uniform distributions. The addition of other distribution types would require inclusion of an appropriate routine for calculating integration weights and nodes based on the integration rules listed in Table 3.1 and references [24] and [8]. For the more common distribution types such routines are generally available and straightforward to code (see [15] and [14]).

Additionally, the constructor code would need to be extended to handle the new distribution types including the input format of the distribution parameters and the transformations from an appropriate standardized random variable to a more general random variable. It happens to be the case that normal and uniform distributions both have two distribution parameters. If a distribution type with a different number of parameters were added, the constructor would need modified. A Matlab cell array would be well-suited to this task, and should include some way to identify the meaning of the parameter (as opposed to the current implementation which requires the numerical value of the moments to be in a particular order).

6.1.2 Sparse Integration

As noted in Chapter 3, the PCE method is ideally suited to a low stochastic dimension due to the curse of dimensionality. The implemented integration routines for this thesis use a tensor product integration scheme that is particularly affected by this issue. Future development may consider incorporating sparse grid integration for the PCE coefficients. For a given set of integration nodes, successive 1D continuation could still be used to locate initial solutions for the integration scheme.

6.1.3 Adaptivity

The choice of integration order and polynomial degree for the PCE implementation is currently determined during problem construction and not adjusted during continuation. It is up to the user to perform numerical experiments to ensure that the statistical moments have converged (as was done in Section 5.5). One could imagine adaptively determining these orders based on a defined convergence criterion. This would potentially be challenging for the integration order and would benefit greatly from a nested integration scheme so that increasing integration order does not imply throwing away all of the existing samples in favor of a new set of nodes.

In addition to integration order and polynomial degree, the convergence of the PCE approximation is dependent on the choice of basis functions. It is possible for an initially optimal set of basis functions to lose their optimality as the output distribution of the response function deviates from the distribution of the stochastic input variables. For larger problems, it would be ideal to be able to detect when this is occurring and initialize a new weighted Ψ matrix and integration nodes. The trade-off is between the computational cost of such reinitialization and the cost of carrying longer PCE expansions (greater P_t value) that are required when convergence is no longer optimal.

A more immediate modification is updating the existing code base with appropriate remesh functions. This would allow the functions in the `'uq'` toolbox to take advantage of the `'coll'` toolboxes existing adaptive remeshing routines for the temporal discretization. In this thesis, relatively high discretization orders were used to work around the lack of adaptive remeshing. Allowing the toolbox to determine the necessary discretization would lead to more efficient and accurate computation.

6.1.4 Integral Response Functions

The form of the integral response function, r_{int} in Chapters 3 and 4 did not include an explicit dependence on time. The evaluation functions for integral response functions were not encoded in such a way that they could handle explicit time dependence. Future work could modify the functions `uq_coll_add_response`, `uq_coll_add_response_adjoint`, and the relevant functions they call to recognize whether the vector field is autonomous or non-autonomous and evaluate the integral appropriately.

Additionally, none of the examples in this thesis demonstrated the functionality of the integral response function. Future work could explore relevant examples that involve such a response function (e.g. maximizing the power of periodic which is related to the integral of the signal squared).

6.1.5 Coupled Boundary-Value Problems

Each sample in the example of Chapter 5 consisted of five coupled boundary-value problems. This could not be directly handled by the toolbox discussed in Chapter 3, so these were added and coupled to one another manually. Future development could streamline construction of such coupled problems and reduce the potential for error when writing the coupling equations.

This issue could be addressed through further generalization of sample construction. If an entire continuation problem instance were passed to a constructor, it could perform 1D continuation of the entire problem and mark integration nodes as appropriate. Such a constructor would be able to handle arbitrary constraints between variables. The constructor would additionally need to return a separate continuation problem instance that included all of the previously added functions duplicated at each integration node.

6.1.6 Robust Constraints

There are two types of constraints in the robust optimization formulation of Section 1.1.3 that were not mentioned in Chapters 3 or 4. The first constraint type given by

$$G\left(\mu_g, \sigma_g\right) \leq 0$$

can be handled by the methods outlined in Chapter 3 (and the `'uq'` toolbox) if they are strictly equality constraints. Such constraints place restrictions on the statistical moments of a function g. In this case a separate PCE can be constructed for the function g to approximate μ_g and σ_g using `uq_coll_add_response` and the constraint can then be applied to the resulting continuation variables for the mean and variance of the constraint function. Like the response functions used in the objective function, the form of the response function for the constraint must be of the form shown in (3.38).

The second constraint type

$$Pr\left[p_{s,min} \leq p_s \leq p_{s,max}\right] \geq P$$

imposes probabilistic restrictions on the values of the statistical parameters. Constraints of this type were not included for two reasons. First, inequality constraints were not considered as part of the optimization formulation in Chapter 2. Second, this type of constraint requires evaluation of failure probabilities and not just statistical moments. Inequality constraints of this form could potentially be handled by the methods outlined in [30] or [31]. Additionally, [32] outlines a method for approximating failure probabilities and their derivatives with respect to problem parameters. Implementing code to handle inequality constraints on failure probabilities, however, is considered outside the scope of the present work.

6.2 Concluding Remarks

The examples shown in this thesis represent a small class of possible applications. The `'uq'`-toolbox code is viewed to be relevant to design problems where parametric variation exists and is likely to affect design performance metrics. The ability of this code to directly vary an objective function in search of optimal statistical moments is believed to be particularly useful.

References

[1] N. Léger, L. Rizzian, and M. Marchi, "Reliability-based design optimization of reinforced concrete structures with elastomeric isolators," *Procedia Engineering*, vol. 199, pp. 1193–1198, 2017.

[2] G. Kewlani, J. Crawford, and K. Iagnemma, "A polynomial chaos approach to the analysis of vehicle dynamics under uncertainty," *Vehicle System Dynamics*, vol. 50, no. 5, pp. 749–774, 2012.

[3] P. Y. Papalambros and D. J. Wilde, *Principles of Optimal Design: Modeling and Computation.* Cambridge University Press, 2000.

[4] G.-J. Park, T.-H. Lee, K. H. Lee, and K.-H. Hwang, "Robust design: An overview," *AIAA Journal*, vol. 44, no. 1, pp. 181–191, 2006.

[5] C. Zang, M. Friswell, and J. Mottershead, "A review of robust optimal design and its application in dynamics," *Computers & Structures*, vol. 83, no. 4, pp. 315–326, 2005.

[6] L. Amândio, A. Marta, F. Afonso, J. Vale, and A. Suleman, "Stochastic optimization in aircraft design," in *Engineering Optimization* (H. C. Rodrigues *et al.*, eds.), 2014.

[7] M. T. Heath, *Scientific Computing: An Introductory Survey.* McGraw-Hill New York, 2002.

[8] D. Xiu and G. E. Karniadakis, "The Wiener–Askey polynomial chaos for stochastic differential equations," *SIAM Journal on Scientific Computing*, vol. 24, no. 2, pp. 619–644, 2002.

[9] X. Wan and G. E. Karniadakis, "Long-term behavior of polynomial chaos in stochastic flow simulations," *Computer Methods in Applied Mechanics and Engineering*, vol. 195, no. 41, pp. 5582–5596, 2006.

[10] V. Heuveline and M. Schick, "A hybrid generalized polynomial chaos method for stochastic dynamical systems," *International Journal for Uncertainty Quantification*, vol. 4, no. 1, 2014.

[11] D. Xiu and J. S. Hesthaven, "High-order collocation methods for differential equations with random inputs," *SIAM Journal on Scientific Computing*, vol. 27, no. 3, pp. 1118–1139, 2005.

[12] S. Hosder, R. Walters, and M. Balch, "Efficient sampling for non-intrusive polynomial chaos applications with multiple uncertain input variables," in *48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Structures, Structural Dynamics, and Materials and Co-located Conferences*, 2007.

[13] M. Gerritsma, J.-B. Van der Steen, P. Vos, and G. E. Karniadakis, "Time-dependent generalized polynomial chaos," *Journal of Computational Physics*, vol. 229, no. 22, pp. 8333–8363, 2010.

[14] J. A. Gubner, "Gaussian quadrature and the eigenvalue problem," tech. rep., University of Wisconsin, 2014.

[15] G. H. Golub and J. H. Welsch, "Calculation of gauss quadrature rules," *Mathematics of Computation*, vol. 23, no. 106, pp. 221–230, 1969.

[16] H. Dankowicz and F. Schilder, *Recipes for Continuation.* Philadelphia, PA: SIAM, 2013.

[17] E. J. Doedel, "AUTO: A program for the automatic bifurcation analysis of autonomous systems," *Congr. Numer*, vol. 30, pp. 265–284, 1981.

[18] A. Dhooge, W. Govaerts, and Y. A. Kuznetsov, "MATCONT: A MATLAB package for numerical bifurcation analysis of ODEs," *ACM Transactions on Mathematical Software (TOMS)*, vol. 29, no. 2, pp. 141–164, 2003.

[19] H. Dankowicz, F. Schilder, M. Li, M. Henderson, and E. Fotsch, "Continuation Core and Toolboxes (COCO)," 2018. [Online]. `http://sourceforge.net/projects/cocotools` Accessed: 2018-09-12.

[20] H. Dankowicz and F. Schilder, "The Trajectory Collocation Toolbox," 2017. [Online]. `https://sourceforge.net/p/cocotools/code/HEAD/tree/cocotools/coco2/help/COLL-Tutorial.pdf` Accessed: 2018-09-12.

[21] M. Li and H. Dankowicz, "Staged construction of adjoints for constrained optimization of integro-differential boundary-value problems," *SIAM Journal on Applied Dynamical Systems*, vol. 17, no. 2, pp. 1117–1151, 2018.

[22] J. Kernévez and E. J. Doedel, *Optimization in Bifurcation Problems using a Continuation Method*, pp. 153–160. Basel: Birkhäuser Basel, 1987.

[23] N. Wiener, "The homogeneous chaos," *American Journal of Mathematics*, vol. 60, no. 4, pp. 897–936, 1938.

[24] B. M. Adams, M. S. Ebeida, M. S. Eldred, G. Geraci, J. D. Jakeman, K. A. Maupin, J. A. Monschke, L. P. Swiler, J. A. Stephens, D. M. Vigil, T. M. Wildey, W. J. Bohnhoff, K. R. Dalbey, J. P. Eddy, J. R. Frye, R. W. Hooper, K. T. Hu, P. D. Hough, M. Khalil, E. M. Ridgway, and A. Rushdi, *Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.6 Theory Manual.* July 2014, Updated May 2017.

[25] M. Eldred and J. Burkardt, "Comparison of non-intrusive polynomial chaos and stochastic collocation methods for uncertainty quantification," in *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, 2009.

[26] R. Lifshitz and M. Cross, "Nonlinear dynamics of nanomechanical and micromechanical resonators," *Review of Nonlinear Dynamics and Complexity*, vol. 1, pp. 1–52, 2008.

[27] H. Cho, B. Jeong, M.-F. Yu, A. F. Vakakis, D. M. McFarland, and L. A. Bergman, "Nonlinear hardening and softening resonances in micromechanical cantilever-nanotube systems originated from nanoscale geometric nonlinearities," *International Journal of Solids and Structures*, vol. 49, no. 15, pp. 2059–2065, 2012.

[28] S. Dou and J. S. Jensen, "Optimization of hardening/softening behavior of plane frame structures using nonlinear normal modes," *Computers & Structures*, vol. 164, pp. 63–74, 2016.

[29] M. Saghafi, H. Dankowicz, and W. Lacarbonara, "Nonlinear tuning of microresonators for dynamic range enhancement," *Proc. R. Soc. A*, vol. 471, no. 2179, p. 20140969, 2015.

[30] B. C. Fabien, "Numerical solution of constrained optimal control problems with parameters," *Applied Mathematics and Computation*, vol. 80, no. 1, pp. 43–62, 1996.

[31] D. Jacobson and M. Lele, "A transformation technique for optimal control problems with a state variable inequality constraint," *IEEE Transactions on Automatic Control*, vol. 14, no. 5, pp. 457–464, 1969.

[32] A. Torii, R. Lopez, and L. Miguel, "A gradient-based polynomial chaos approach for risk and reliability-based design optimization," *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 39, no. 7, pp. 2905–2915, 2017.

Appendix A

Optimization Code

This appendix contains functions necessary for the numerical investigation conducted in Chapter 2.

Vectorized, `'coll'` compatible MATLAB vector field and Jacobians for the original and adjoint systems:

```
function y = linode(t, x, p)

x1 = x(1,:);
x2 = x(2,:);
k = p(1,:);
phi = p(2,:);

y(1,:) = x2;
y(2,:) = cos(t + phi) - x2 - k.*x1;

end
```

```
function J = linode_dx(t, x, p)

x1 = x(1,:);
k = p(1,:);

J = zeros(2,2,numel(x1));
J(1,2,:) = 1;
J(2,1,:) = -k;
J(2,2,:) = -1;

end
```

```
function J = linode_dp(t, x, p)

x1 = x(1,:);
phi = p(2,:);

J = zeros(2,2,numel(x1));
J(2,1,:) = -x1;
J(2,2,:) = -sin(t + phi);

end
```

```
function J = linode_dt(t, x, p)

x1 = x(1,:);
phi = p(2,:);

J = zeros(2,numel(x1));
J(2,:) = -sin(t + phi);

end
```

```
function y = adj_F(t, la, p)
```

```

la1 = la(1,:);
la2 = la(2,:);

k = p(1,:);

y(1,:) = k.*la2;
y(2,:) = la2 - la1;

end
```

```
function J = adj_F_dU(t, la, p)

k = p(1,:);

J = zeros(2,2,numel(t));
J(1,2,:) = k;
J(2,1,:) = -1;
J(2,2,:) = 1;

end
```

```
function J = adj_F_dp(t, la, p)

la2 = la(2,:);

J = zeros(2,2,numel(la2));

J(1,1,:) = la2;

end
```

```
function J = adj_F_dt(t, la, p)

la1 = la(1,:);

J = zeros(2,numel(la1));

end
```

Functions for the boundary conditions and stationarity constraints in a COCO compatible zero function format.

```
function [data, y] = linode_bc(prob, data, u)

x0 = u(1:2);
x1 = u(3:4);
T0 = u(5);
T  = u(6);

y = [x0 - x1; T0; T-2*pi];

end
```

```
function [data, J] = linode_bc_du(prob, data, u)

J = [  1,  0, -1,  0,  0,  0;
       0,  1,  0, -1,  0,  0;
       0,  0,  0,  0,  1,  0;
       0,  0,  0,  0,  0,  1];

end
```

```
function [data, y] = adj_F_bc(prob, data, u)

la0 = u(1:2);
la1 = u(3:4);
T0 = u(5);
T  = u(6);
la3 = u(7);
la4 = u(8);
eta1 = u(9);

y = [-la0(1) + la3 + eta1; ...
     -la0(2) + la4; ...
      la1(1) - la3; ...
      la1(2) - la4; ...
      T0; T-2*pi];

end
```

```
function [data, J] = adj_F_bc_du(prob, data, u)

J = [-1,  0,  0,  0,  0,  0,  1,  0,  1;
      0, -1,  0,  0,  0,  0,  0,  1,  0;
      0,  0,  1,  0,  0,  0, -1,  0,  0;
      0,  0,  0,  1,  0,  0,  0, -1,  0;
      0,  0,  0,  0,  1,  0,  0,  0,  0;
      0,  0,  0,  0,  0,  1,  0,  0,  0];

end
```

```
function [data, y] = stationarity(prob, data, u)

dim = data.coll_seg.maps.xbp_shp(2);

x1bp = u(1:dim);
la2bp = u(dim+1:2*dim);
T = u(end-4);
k = u(end-3);
phi = u(end-2);
eta_k = u(end-1);
eta_phi = u(end);

% Numerical Integration
dim = data.coll_seg.int.dim;
W = data.coll_seg.maps.W(1:dim:end, 1:dim:end);
ka = repmat(data.coll_seg.mesh.ka,[data.coll_seg.int.NCOL,1]);
ka = ka(:);
tcn = data.coll_seg.mesh.tcn;
w = data.coll_seg.mesh.gwt';
N = data.coll_seg.maps.NTST;

y1_int = (0.5*T/N)*((W*la2bp).*(W*x1bp).*ka)'*w;
y2_int = (0.5*T/N)*(sin(T*tcn+phi).*(W*la2bp).*ka)'*w;

y = [y1_int + eta_k;
     y2_int + eta_phi];

end
```

```
function [data, J] = stationarity_du(prob, data, u)

dim = data.coll_seg.maps.xbp_shp(2);
x1bp = u(1:dim);
la2bp = u(dim+1:2*dim);
T = u(end-4);
phi = u(end-2);

dim = data.coll_seg.int.dim;
```

```
w = data.coll_seg.mesh.gwt';
W = data.coll_seg.maps.W(1:dim:end, 1:dim:end);
ka = repmat(data.coll_seg.mesh.ka,[data.coll_seg.int.NCOL,1]);
ka = ka(:);
tcn = data.coll_seg.mesh.tcn;
N = data.coll_seg.maps.NTST;

% Jacobians of portions of the Numerical Integration
J_u = [(0.5*T/N)*((W.^2)*diag(la2bp))'*(ka.*w), ...
       zeros(size(x1bp))]';

J_la = [(0.5*T/N)*((W.^2)*diag(x1bp))'*(ka.*w), ...
        (0.5*T/N)*(diag(sin(T*tcn+phi).*ka)*W)'*w]';

fds = (0.5*T/N)*diag(cos(T*tcn+phi))*diag(tcn);
sdf = diag(sin(T*tcn+phi));

J_T = [(0.5/N)*((W*x1bp).*(W*la2bp).*ka)'*w; ...
       (0.5/N)*((fds + sdf)*((W*la2bp).*ka))'*w];

J_p = [0, 0;
       0, (0.5*T/N)*((W*la2bp).*cos(T*tcn + phi).*ka)'*w];

J_eta = eye(2);

% Combining into Full Jacobian
J = [J_u, J_la, J_T, J_p, J_eta];

end
```

Appendix B

‘uq’ Toolbox Code

B.1 Sample Construction

The code in this section implements generalized constructors for equations that define a sample of linked boundary value problems (and their adjoints). Functions called by the constructors are included in the order that they are called.

B.1.1 Construction from an Initial Solution

```
function prob = uq_isol2bvp_sample(prob, oid, varargin)

  tbid = coco_get_id(oid, 'uq');
  str  = coco_stream(varargin{:});
  temp_str = coco_stream(varargin{:});
  temp_prob2 = ode_isol2bvp(prob, tbid, str);
  [args, opts] = uq_parse_str(str);
  bvp_id = coco_get_id(tbid, 'bvp');
  bc_data = coco_get_func_data(temp_prob2, bvp_id, 'data');

  data = bvp_uq_init_data(bc_data, oid);
  data = uq_init_data(prob, data, args, opts);
  data = uq_bvp_gen_samples(data, prob, temp_str);

  [prob, data] = uq_bvp_add_samples(prob, data, bc_data, args);
  psi_mat = uq_make_psi_mat(data.nds_grid, data.uq.Pt, data.spdists);
  data.wtd_psi_mat = psi_mat*diag(data.wts);

  prob  = uq_add_sample_nodes(prob, data, args);

end
```

```
function [uqdata, opts] = uq_parse_str(str)

  grammar   = 'SPNAMES SPDIST SPDP [DPARNAMES] [OPTS]';
  args_spec = {
       'SPNAMES', 'cell', '{str}',    'spnames',       {}, 'read', {}
        'SPDIST', 'cell', '{str}',    'spdists',       {}, 'read', {}
          'SPDP',     '', '[num]',       'spdp',       {}, 'read', {}
    'DPARNAMES', 'cell', '{str}',   'dpdtpars',       {}, 'read', {}
    };

  opts_spec = {
    '-add-adjt', 'addadjt', false, 'toggle', {}
    };

  [uqdata, opts] = coco_parse(grammar, args_spec, opts_spec, str);

```

```
end
```

```
function data = bvp_uq_init_data(src_data, oid, varargin)
data.oid = oid;

fields = varargin;
for i=1:numel(fields)
  field = fields{i};
  if isfield(src_data, field)
    data.(field) = src_data.(field);
  else
    data.(field) = struct();
  end
end

data.pnames     = src_data.pnames;
data.xdim       = size(src_data.bvp_bc.x0_idx, 1);
data.pdim       = size(src_data.bvp_bc.p_idx, 1);

end
```

```
function data = uq_init_data(prob, data, args, opts)
  tbid = coco_get_id(data.oid, ’uq’);
  data.uq = uq_get_settings(prob, tbid, struct());
  fields = fieldnames(args);
  for i = 1:numel(fields)
      data.(fields{i}) = args.(fields{i});
  end

  if ischar(data.spnames)
      data.spnames = cellstr(data.spnames);
      data.spdists = cellstr(data.spdists);
  end
  data.s = length(data.spnames);
  data.sp2p_idx = ismember(data.pnames, data.spnames);
  data.dpar_idx = ismember(data.pnames, data.dpdtpars);
  p_in_sp = ismember(data.spnames, data.pnames);

  if numel(data.uq.M) == 1
      data.uq.M = repelem(data.uq.M, data.s);
  end

  data.Nt = factorial(data.s + data.uq.Pt)/...
       ((factorial(data.s))*(factorial(data.uq.Pt)));

  [nds, wts, idx] = uq_gauss_nodes(data.uq.M, data.s, data.spdists);
  data.wts = wts;
  data.idx = idx;
  data.nsamples = numel(data.wts);

  if sum(p_in_sp) < data.s
      not_in_p = data.spnames(~p_in_sp);
      not_in_p_string = strtrim(sprintf(’
      err_string = sprintf([’Parameter(s) {
                            ’problem instance’], not_in_p_string(1:end-1));
      assert(sum(p_in_sp) == data.s, err_string)
  end

  data.st_nds = nds’;
  data.nds = nds’;
  data.nds_grid = data.nds(data.idx);
  if data.s == 1
    data.nds_grid = data.nds_grid’;
  end

  data.sample_par_idx = zeros(size(data.nds));
  data.normal_par_idx = strcmpi(’normal’, data.spdists);
  data.normal_nds_idx = repelem(data.normal_par_idx, data.uq.M);
  data.num_normals = nnz(data.normal_par_idx);
  data.uniform_par_idx = strcmpi('uniform', data.spdists);
  data.uniform_nds_idx = repelem(data.uniform_par_idx, data.uq.M);
  data.num_uniforms = nnz(data.uniform_par_idx);
  if opts.addadjt
    data.addadjt = 1;
    data.adjt_sample_par_idx = zeros(size(data.nds));
  else
    data.addadjt = 0;
  end

  data = uq_gen_nds(data);
end
```

```
function [uq, spec] = uq_get_settings(prob, tbid, uq)

spec = {  'M',   '[int]',   4, 'read',   {},  'Numerical Integration Order'
          'Pt',  'int',     3, 'read',   {},  'Max Polynomial Order for Polynomial Chaos Expansion'
 };

uq = coco_parse_settings(prob, spec, uq, tbid);

end
```

```
function [nds, wts, idx] = uq_gauss_nodes(m, n, type)

if ischar(type)
    type = cellstr(type);
end
assert(numel(m) == 1 || numel(m) == n, ['Number of entries for m must ' ...
       'either be 1 (equal quadrature order for all variables) ' ...
       'or equal to n (specified quadrature order for each variable)'])

assert(numel(type) == 1 || numel(type) == n, ['Number of entries for ' ...
       'type must either be 1 (equal distribution for all variables) ' ...
       'or equal to n (specified quadrature order for each variable)'])
single_integration_node = false;
if n > 1
    if numel(m)==1
      if m == 1
        single_integration_node = true;
      end
        m = repmat(m, 1, n);
    end

    if numel(type)==1
        type = repmat(type, 1, n);
    end
end

idx = cellfun(@(x) 1:x, num2cell(m), 'uniformoutput', false);
combinations = cell(1, numel(idx));
[combinations{:}] = ndgrid(idx{:});
combinations = cellfun(@(x) x(:), combinations, 'uniformoutput', false);
idx = [combinations{:}]';
os = cumsum(max(idx')');
idx = idx + repmat([0;os(1:end-1)],1,size(idx,2));
nds = cell(1, n);
wts = cell(1, n);
if single_integration_node
  nds = 0;
  wts = 1;
else
  for i=1:n
      switch lower(type{i})
          case lower({'Legendre','Le','LeN', 'Uniform', 'U'})
              [nds{i}, wts{i}] = gauss_legendre_nodes(m(i));
          case lower({'Hermite','He','HeN', 'Normal', 'N'})
```

```
            [nds{i}, wts{i}] = gauss_hermite_nodes(m(i));
        otherwise
            warning(strcat('Specified Type not supported, Providing',...
                           ' weights and nodes for Gauss-Legendre',...
                           ' quadrature'))
            [nds{i}, wts{i}] = gauss_legendre_nodes(m(i));
    end
  end
  nds = [nds{:}];
  wts = [wts{:}];
  wts = prod(wts(idx), 1);
end

end

function [nds, wts] = gauss_legendre_nodes(m)
num = (1:m-1)';
g = num.*sqrt(1./(4*num.^2-1));
J = -diag(g,1)-diag(g,-1);
[w, x] = eig(J);
nds = diag(x)';
wts = (2*w(1,:).^2)/2;
end

function [nds, wts] = gauss_hermite_nodes(m)
  num = (1:m-1)';
  g = sqrt(num);
  J = diag(g,1)+diag(g,-1);
  [w, x] = eig(J);
  [nds, idx] = sort(diag(x));
  nds = nds';
  wts = sqrt(2*pi)*w(1,:).^2;
  wts = wts(idx)/(sqrt(2*pi));
end
```

```
function data = uq_gen_nds(data)

if data.num_normals > 0
    mu = data.spdp(data.normal_par_idx,1);
    sig = data.spdp(data.normal_par_idx,2);

    mu_rep = repelem(mu, data.uq.M(data.normal_par_idx));
    sig_rep = repelem(sig, data.uq.M(data.normal_par_idx));

    if size(mu_rep,1) == 1
        mu_rep = mu_rep';
        sig_rep = sig_rep';
    end

    data.nds(data.normal_nds_idx) = ...
        mu_rep + sig_rep.*data.st_nds(data.normal_nds_idx);
end

if data.num_uniforms > 0
    lo_gt_up = data.spdp(data.uniform_par_idx,1) > ...
               data.spdp(data.uniform_par_idx,2);

    if nnz(lo_gt_up) ~= 0
        uniform_rv_names = data.spnames(data.uniform_par_idx);
        bad_data_entries = uniform_rv_names(lo_gt_up);
        bad_data_string = strtrim(sprintf('
        assert(nnz(lo_gt_up) == 0, ...
           ['First distribution parameter for uniform random variables' ...
            ' must be smaller than second parameter.  Check values for' ...
            ' parameters: {
    end
    lo = data.spdp(data.uniform_par_idx,1);
    hi = data.spdp(data.uniform_par_idx,2);
    lo_rep = repelem(lo, data.uq.M(data.uniform_par_idx));
    up_rep = repelem(hi, data.uq.M(data.uniform_par_idx));

    if size(lo_rep,1) == 1
        lo_rep = lo_rep';
        up_rep = up_rep';
    end

    data.nds(data.uniform_nds_idx) = ...
        ((up_rep - lo_rep)/2).*data.st_nds(data.uniform_nds_idx) ...
        + (up_rep + lo_rep)/2;
end

end
```

```
function data = uq_bvp_gen_samples(data, prob, str)
tbid = coco_get_id(data.oid, 'uq');
num_pars = numel(data.spnames);
runs_per_parameter = cumprod([1,data.uq.M(1:end-1)]);
total_runs = sum(runs_per_parameter);
run_names = cell(1, total_runs);

prob_init = coco_prob();
for setting={'autonomous', 'vectorized'}
  value = coco_get(prob, 'ode', setting{:});
  if ~isempty(value)
    prob_init = coco_set(prob_init, 'ode', setting{:}, value);
  end
end

for setting={'NCOL', 'NTST', 'var'}
  value = coco_get(prob, 'coll', setting{:});
  if ~isempty(value)
    prob_init = coco_set(prob_init, 'coll', setting{:}, value);
  end
end

pnum = 1;
k = 1;
while pnum <= num_pars
    temp_bds = cell(1, runs_per_parameter(pnum));
    idx = false(1, num_pars);
    idx(pnum) = true;
    idx = repelem(idx, data.uq.M);
    vals = data.nds(idx);
    low = min(vals);
    high = max(vals);
    msg = sprintf('\n
                  tbid, data.spnames{pnum});
    fprintf(msg)
    if pnum == 1
        run_names{k} = [tbid, '_samples_p', int2str(pnum), '_run', int2str(1)];
        temp_prob2 = prob_init;
        temp_prob2 = ode_isol2bvp(temp_prob2, 'sample', str);
        [~, opts] = uq_parse_str(str);
        pnames = {data.spnames{pnum}, data.pnames{data.dpar_idx}};

        if opts.addadjt
          temp_prob2 = adjt_isol2bvp(temp_prob2, 'sample');
          dnames = coco_get_id('d', data.pnames(~data.dpar_idx));
          pnames = {pnames{:}, dnames{:}};
        end
        temp_prob2 = coco_add_event(temp_prob2, 'UQ', ...
            data.spnames{pnum}, vals);
        temp_bds{1} = coco(temp_prob2, run_names{k}, [], 1, pnames, ...
            [low, high]);
        k = k+1;
    else
        run_count = 1;
        for run=1:runs_per_parameter(pnum-1)
```

```
            labs = coco_bd_labs(bds{run}, 'UQ');
            for lab = labs
                run_names{k} = [tbid, '_samples_p', int2str(pnum), ...
                    '_run', int2str(run_count)];
                previous_run_name = [tbid, '_samples_p', ...
                    int2str(pnum-1), '_run', int2str(run)];
                temp_prob2 = prob_init;
                temp_prob2 = ode_bvp2bvp(temp_prob2, 'sample', ...
                    previous_run_name, lab);
                pnames = {data.spnames{pnum}};
                pnames = {pnames{:}, data.pnames{data.dpar_idx}};
                if opts.addadjt
                  temp_prob2 = adjt_bvp2bvp(temp_prob2, 'sample', ...
                  previous_run_name, lab);
                  dnames = coco_get_id('d', data.pnames);
                  pnames = {pnames{:}, dnames{:}};
                end
                pnames = {pnames{:}, data.spnames{1:pnum-1}};
                temp_prob2 = coco_add_event(temp_prob2, 'UQ', ...
                    data.spnames{pnum}, vals);
                temp_bds{run_count} = coco(temp_prob2, run_names{k}, [], ...
                    1, pnames, ...
                    [low, ...
                     high]);
                run_count = run_count + 1;
                k = k+1;
            end

        end
    end
    bds = temp_bds;
    pnum = pnum+1;
end

for i=1:(numel(run_names) - runs_per_parameter(end))
    try
        rmdir(['data/', run_names{i}], 's')
    catch
    end
end

data.sample_run_names = run_names(end-runs_per_parameter(end)+1:end);

end
```

```
function [prob, data] = uq_bvp_add_samples(prob, data, bc_data, args)

[prob, data] = uq_coll_add_samples(prob, data);
[prob, data] = uq_bvp_close_samples(prob, data, bc_data, args);

end
```

```
function [prob, data] = uq_coll_add_samples(prob, data)


tbid = coco_get_id(data.oid, 'uq');
reorder_idx = zeros(size(data.wts));
run_count = 1;
run_names = data.sample_run_names;
data.sids = cell(1, data.nsamples);
ndvals = data.nds(data.idx);
if size(ndvals,2) == 1
    ndvals = ndvals';
end
for i=1:numel(run_names)
  bd = coco_bd_read(run_names{i});
  labs = coco_bd_labs(bd, 'UQ');
  for lab=labs
```

```
    sol = coll_read_solution('sample.bvp.seg1', run_names{i}, lab);
    p_rep = repmat(sol.p(data.sp2p_idx), 1, prod(data.uq.M, 2));
    nd_diff = sum((ndvals - p_rep).^2,1);
    reorder_idx(run_count) = find(nd_diff==min(nd_diff));
    sample_name = coco_get_id(tbid, sprintf('sample
    prob = ode_coll2coll(prob, sample_name, run_names{i}, 'sample.bvp.seg1', lab);
    if data.addadjt
      prob = adjt_coll2coll(prob, sample_name, run_names{i}, 'sample.bvp.seg1', lab);
    end

    data.sids{run_count} = sample_name;
    run_count = run_count+1;
  end
end

data.xdim = size(sol.xbp, 2);
data.pdim = size(sol.p, 1);
data.idx = data.idx(:, reorder_idx);
data.wts = data.wts(reorder_idx);
data.nds_grid = data.nds_grid(:, reorder_idx);
data.reorder_idx = reorder_idx;

for i=1:numel(run_names)
  try
    rmdir(['data/', run_names{i}], 's')
  catch
  end
end

end
```

```
function [prob, data] = uq_bvp_close_samples(prob, data, bc_data, args)

if nargin < 4
  args = {};
end

bc   = bc_data.bvp_bc;

nsamples  = data.nsamples;
s_idx  = cell(1, nsamples);

fxd_p_idx = and(~data.sp2p_idx, ~data.dpar_idx);
frd_p_idx = and(~data.sp2p_idx, data.dpar_idx);
has_unique_deterministic_parameters = sum(frd_p_idx) > 0;
if has_unique_deterministic_parameters
  unq_p_idx = cell(1, nsamples);
end

add_tl0 = 0;

if isfield(args, 'run') && isfield(args, 'lab')
  chart = coco_read_solution(args.run, args.lab, 'chart');
  if strcmpi(chart.pt_type, 'BP')
    add_tl0 = 1;
  end
end

for i=1:nsamples
  fid  = coco_get_id(data.sids{i}, 'bc');


  coll_id = coco_get_id(data.sids{i}, 'coll');
  [fdata, uidx] = coco_get_func_data(prob, coll_id, 'data', 'uidx');
  maps = fdata.coll_seg.maps;
  T0_idx = uidx(maps.T0_idx);
  T_idx  = uidx(maps.T_idx);
  x0_idx = uidx(maps.x0_idx);
  x1_idx = uidx(maps.x1_idx);
  s_idx{i} = uidx(maps.p_idx);
  if has_unique_deterministic_parameters
    unq_p_idx{i} = uidx(maps.p_idx(frd_p_idx));
  end

  uidx = [T0_idx; T_idx; x0_idx; x1_idx; s_idx{i}];
  prob = coco_add_func(prob, fid, @F, @DF, @DDF, bc_data, ’zero’, ...
    ’uidx’, uidx);
  prob = coco_add_slot(prob, fid, @coco_save_data, bc_data, ’save_full’);

  if data.addadjt
    if isfield(args, ’run’) && isfield(args, ’lab’)
      chart = coco_read_solution(args.run, args.lab, ’chart’);
      cdata = coco_get_chart_data(chart, ’lsol’);
      [chart, lidx] = coco_read_adjoint(fid, args.run, args.lab, ...
        ’chart’, ’lidx’);
      l0 = chart.x;
      if add_tl0
        tl0 = cdata.v(lidx);
      end
    end

    [fdata, aidx] = coco_get_adjt_data(prob, coll_id, ’data’, ’axidx’);
    opt = fdata.coll_opt;
    T0_idx = aidx(opt.T0_idx);
    T_idx  = aidx(opt.T_idx);
    x0_idx = aidx(opt.x0_idx);
    x1_idx = aidx(opt.x1_idx);
    adj_s_idx{i} = aidx(opt.p_idx);
    if has_unique_deterministic_parameters
      adj_unq_p_idx{i} = aidx(opt.p_idx(frd_p_idx));
    end

    aidx = [T0_idx; T_idx; x0_idx; x1_idx; adj_s_idx{i}];
    if isfield(args, ’run’) && isfield(args, ’lab’)
      if add_tl0
        prob = coco_add_adjt(prob, fid, ’aidx’, aidx, ’l0’, l0, ’tl0’, tl0);
      else
        prob = coco_add_adjt(prob, fid, ’aidx’, aidx, ’l0’, l0);
      end
    else
      prob = coco_add_adjt(prob, fid, ’aidx’, aidx);
    end
  end
end
data.s_idx = s_idx;
s_idx1 = s_idx{1};
if data.addadjt
  data.adj_s_idx = adj_s_idx;
  adj_s_idx1 = adj_s_idx{1};
end

if sum(fxd_p_idx) > 0
  for i=2:nsamples
    s_idxi = s_idx{i};

    sfid = coco_get_id(bc.fid, sprintf(’shared

    prob = coco_add_glue(prob, sfid, s_idx1(fxd_p_idx), s_idxi(fxd_p_idx));

    if data.addadjt
      adj_s_idxi = adj_s_idx{i};
      if isfield(args, ’run’) && isfield(args, ’lab’)
        chart = coco_read_solution(args.run, args.lab, ’chart’);
        cdata = coco_get_chart_data(chart, ’lsol’);
        [chart, lidx] = coco_read_adjoint(sfid, args.run, args.lab, ...
          ’chart’, ’lidx’);
        l0 = chart.x;
        if add_tl0
          tl0 = cdata.v(lidx);
        end
      else
        l0 = zeros([fxd_p_idx,1]);
      end
      if add_tl0
        prob = coco_add_adjt(prob, sfid, ...
          'aidx', [adj_s_idx1(fxd_p_idx); adj_s_idxi(fxd_p_idx)], ...
          'l0', l0, ...
          'tl0', tl0);
      else
        prob = coco_add_adjt(prob, sfid, ...
          'aidx', [adj_s_idx1(fxd_p_idx); adj_s_idxi(fxd_p_idx)], ...
          'l0', l0);
      end
     end
   end
  end

if ~isempty(data.pnames) && sum(fxd_p_idx) > 0
  pfid  = coco_get_id(bc.fid, 'shared_pars');
  prob = coco_add_pars(prob, pfid, s_idx1(fxd_p_idx), ...
    data.pnames(fxd_p_idx));
  if data.addadjt
    if isfield(args, 'run') && isfield(args, 'lab')
      chart = coco_read_solution(args.run, args.lab, 'chart');
      cdata = coco_get_chart_data(chart, 'lsol');
      [chart, lidx] = coco_read_adjoint(pfid, args.run, args.lab, ...
        'chart', 'lidx');
      l0 = chart.x;
      if add_tl0
        tl0 = cdata.v(lidx);
        prob = coco_add_adjt(prob, pfid, ...
          coco_get_id('d', data.pnames(fxd_p_idx)), ...
          'aidx', adj_s_idx1(fxd_p_idx), ...
          'l0', l0, ...
          'tl0', tl0);
      else
        prob = coco_add_adjt(prob, pfid, ...
          coco_get_id('d', data.pnames(fxd_p_idx)), ...
          'aidx', adj_s_idx1(fxd_p_idx), ...
          'l0', l0);
      end
    else
      prob = coco_add_adjt(prob, pfid, ...
        coco_get_id('d', data.pnames(fxd_p_idx)), 'aidx', ...
        adj_s_idx1(fxd_p_idx));
    end
  end
end

if ~isempty(data.pnames) && sum(frd_p_idx) > 0
  unq_p_idx = cell2mat(unq_p_idx)';
  pnames = uq_get_sample_par_names(data.pnames(frd_p_idx), 1:nsamples);
  pfid  = coco_get_id(bc.fid, 'unique_pars');
  prob = coco_add_pars(prob, pfid, unq_p_idx, pnames, 'active');
  if data.addadjt
    adj_unq_p_idx = cell2mat(adj_unq_p_idx);
    if isfield(args, 'run') && isfield(args, 'lab')
      chart = coco_read_solution(args.run, args.lab, 'chart');
      cdata = coco_get_chart_data(chart, 'lsol');
      [chart, lidx] = coco_read_adjoint(pfid, args.run, args.lab, ...
        'chart', 'lidx');
      l0 = chart.x;
      if add_tl0
        tl0 = cdata.v(lidx);
        prob = coco_add_adjt(prob, pfid, ...
          coco_get_id('d', pnames), ...
          'aidx', adj_unq_p_idx, ...
          'l0', l0, ...
          'tl0', tl0);
      else
        prob = coco_add_adjt(prob, pfid, ...
          coco_get_id('d', pnames), ...
          'aidx', adj_unq_p_idx, ...
          'l0', l0);
      end
    else
      prob = coco_add_adjt(prob, pfid, coco_get_id('d', pnames), ...
        'aidx', adj_unq_p_idx);
    end
  end
end

end

function [data, y] = F(prob, data, u)

bc = data.bvp_bc;

T0 = u(bc.T0_idx);
T  = u(bc.T_idx);
x0 = u(bc.x0_idx);
x1 = u(bc.x1_idx);
p  = u(bc.p_idx);

args = {T0, T, x0, x1, p};
y = data.fhan(data, args{bc.nargs+1:end});

end

function [data, J] = DF(prob, data, u)

bc = data.bvp_bc;

T0 = u(bc.T0_idx);
T  = u(bc.T_idx);
x0 = u(bc.x0_idx);
x1 = u(bc.x1_idx);
p  = u(bc.p_idx);

args = {T0, T, x0, x1, p};

J = data.dfdxhan(data, args{bc.nargs+1:end});
J = [zeros(size(J,1), bc.nargs*numel(T0)), J];

end

function [data, dJ] = DDF(prob, data, u)

bc = data.bvp_bc;

T0 = u(bc.T0_idx);
T  = u(bc.T_idx);
x0 = u(bc.x0_idx);
x1 = u(bc.x1_idx);
p  = u(bc.p_idx);

args = {T0, T, x0, x1, p};

dJ = data.dfdxdxhan(data, args{bc.nargs+1:end});

if bc.nargs
  dJ_tmp = zeros(size(dJ,1), numel(u), numel(u));
  dJ_tmp(:,numel(T0)+1:end, numel(T0)+1:end) = dJ;
  dJ = dJ_tmp;
end


end
```

```
function [names_out] = uq_get_sample_par_names(names, idx)

  names_out = {};

  if ischar(names)
    names = cellstr(names);
  end

  for i=1:numel(idx)
    for name=names
      n = sprintf('
      names_out = {names_out{:} n};
    end
  end
end
```

```
function psi = uq_make_psi_mat(sample, max_order, poly_type)

sample = sample';

if nargin < 3
    poly_type = 'Legendre';
end

n = size(sample,2);
ps = cell(1,n);
for k = 1:n
    ps{k} = 0:max_order;
end
grid_cells = cell(size(ps));

[grid_cells{:}] = ndgrid(ps{:});

grids = zeros(n, numel(grid_cells{1}));

for i=1:n
    grids(i,:) = reshape(grid_cells{i},1,[]);
end

idx = sum(grids,1) <= max_order;
grids = grids(:, idx)';
[~,idx] = sort(sum(grids,2));
grids = grids(idx,:);

idx = max_order*(0:(n-1)) + (1:n);
idx = grids + repmat(idx,size(grids,1),1);

[d1, d2] = size(sample);
[d3, d4] = size(idx);

psi = uq_orthogonal_poly_vals(sample, poly_type, max_order, 1);
psi = permute(psi, [1,3,2]);
psi = reshape(psi, [], d2*(max_order+1));
psi = psi(:,idx)';
psi = reshape(psi,d3,d4,d1);
psi = prod(psi,2);
psi = permute(psi,[3,1,2]);
psi = psi';
```

```
function psi = uq_orthogonal_poly_vals(x, types, max_order, norm)

[M, s] = size(x);
psi = zeros(M, s, max_order+1);
psi(:,:,1) = 1;
psi(:,:,2) = x;

if nargin < 4
    norm = 1;
```

```
end

if ischar(types)
  types = repmat(cellstr(types), 1, s);
end

wts = ones(M, s, max_order+1);

for j=1:s
  type = types{j};
  switch type
    case {'Legendre', 'Le', 'Uniform'}
      if norm
        wts(:,j,:) = repmat(sqrt(2*(0:max_order)+1), M, 1);
      else

      end
      if max_order==0
        psi = psi(:,:,1);
      else
        for i=3:(max_order+1)
          n = i-2;
          psi(:,j,i) = ((2*n+1)*x(:,j).*psi(:,j,i-1) - n*psi(:,j,i-2))/(n+1);
        end
      end

    case {'Hermite', 'He', 'Normal'}
      if norm
        wts(:,j,:) = repmat(sqrt(factorial(0:max_order)).^-1, M, 1);
      else

      end
      if max_order==0
        psi = psi(:,:,1);
      else
        for i=3:(max_order+1)
          psi(:,j,i) = x(:,j).*psi(:,j,i-1) - (i-2)*psi(:,j,i-2);
        end
      end
  end
end
psi = wts.*psi;
end
```

```
function prob = uq_add_sample_nodes(prob, data, args)


if nargin < 3
  args = {};
end

add_tl0 = 0;

if isfield(args, 'run') && isfield(args, 'lab')
  chart = coco_read_solution(args.run, args.lab, 'chart');
  if strcmpi(chart.pt_type, 'BP')
    add_tl0 = 1;
  end
end

if data.num_normals > 0
    muid = coco_get_id(data.oid, 'mus');
    sigid = coco_get_id(data.oid, 'sigs');
    mu = data.spdp(data.normal_par_idx,1);
    sig = data.spdp(data.normal_par_idx,2);

    normal_names = data.spnames(data.normal_par_idx);
    munames = strcat('mu.', normal_names);
    signames = strcat('sig.', normal_names);
    prob = coco_add_pars(prob, muid, munames, mu);
    prob = coco_add_pars(prob, sigid, signames, sig);

    muidx = coco_get_func_data(prob, muid, 'uidx');
    sigidx = coco_get_func_data(prob, sigid, 'uidx');
    normal_sample_id = coco_get_id(data.oid, 'normal_sample');

    prob = coco_add_func(prob, normal_sample_id, @uq_normal_sample, ...
      @uq_normal_sample_dU, @uq_normal_sample_dUdU, data, 'zero', ...
      'uidx', [muidx;sigidx], 'u0', data.nds(data.normal_nds_idx));

    if data.addadjt
      if isfield(args, 'run')
        chart = coco_read_solution(args.run, args.lab, 'chart');
        cdata = coco_get_chart_data(chart, 'lsol');
        [mu_chart, mu_lidx] = coco_read_adjoint(muid, args.run, args.lab, ...
          'chart', 'lidx');
        mu_l0 = mu_chart.x;

        [sig_chart, sig_lidx] = coco_read_adjoint(sigid, args.run, args.lab, ...
          'chart', 'lidx');
        sig_l0 = sig_chart.x;

        [norm_chart, normal_lidx] = coco_read_adjoint(normal_sample_id, args.run, args.lab, ...
          'chart', 'lidx');
        norm_l0 = norm_chart.x;
        if add_tl0
          mu_tl0 = cdata.v(mu_lidx);
          sig_tl0 = cdata.v(sig_lidx);
          norm_tl0 = cdata.v(normal_lidx);
        end
      else
        mu_l0 = zeros(size(mu));
        sig_l0 = zeros(size(sig));
        norm_l0 = zeros(size(data.normal_nds_idx'));
      end
      if add_tl0
        prob = coco_add_adjt(prob, muid, coco_get_id('d', munames), ...
          'l0', mu_l0, 'tl0', mu_tl0);
        prob = coco_add_adjt(prob, sigid, coco_get_id('d', signames), ...
          'l0', sig_l0, 'tl0', sig_tl0);
        amuidx = coco_get_adjt_data(prob, muid, 'axidx');
        asigidx = coco_get_adjt_data(prob, sigid, 'axidx');
        prob = coco_add_adjt(prob, normal_sample_id, 'aidx', ...
          [amuidx; asigidx], 'l0', norm_l0, 'tl0', norm_tl0);
      else
        prob = coco_add_adjt(prob, muid, coco_get_id('d', munames), ...
          'l0', mu_l0);
        prob = coco_add_adjt(prob, sigid, coco_get_id('d', signames), ...
          'l0', sig_l0);
        amuidx = coco_get_adjt_data(prob, muid, 'axidx');
        asigidx = coco_get_adjt_data(prob, sigid, 'axidx');
        prob = coco_add_adjt(prob, normal_sample_id, 'aidx', ...
          [amuidx; asigidx], 'l0', norm_l0);
      end
    end
    uidx = coco_get_func_data(prob, normal_sample_id, 'uidx');
    data.sample_par_idx(data.normal_nds_idx) = ...
        uidx((2*data.num_normals+1):end);
    if data.addadjt
      aidx = coco_get_adjt_data(prob, normal_sample_id, 'axidx');
      data.adjt_sample_par_idx(data.normal_nds_idx) = ...
        aidx((2*data.num_normals+1):end);
    end
end

if data.num_uniforms > 0
  upid = coco_get_id(data.oid, 'up');
  loid = coco_get_id(data.oid, 'lo');

```

```
lo = data.spdp(data.uniform_par_idx,1);
up = data.spdp(data.uniform_par_idx,2);

uniform_names = data.spnames(data.uniform_par_idx);
lonames = strcat('lo.', uniform_names);
upnames = strcat('up.', uniform_names);

prob = coco_add_pars(prob, loid, lonames, lo);
prob = coco_add_pars(prob, upid, upnames, up);

loidx = coco_get_func_data(prob, loid, 'uidx');
upidx = coco_get_func_data(prob, upid, 'uidx');
uniform_sample_id = coco_get_id(data.oid, 'uniform_sample');

prob = coco_add_func(prob, uniform_sample_id, @uq_uniform_sample, ...
  @uq_uniform_sample_dU, @uq_uniform_sample_dUdU, data, 'zero', ...
  'uidx', [loidx;upidx], 'u0', data.nds(data.uniform_nds_idx));

if data.addadjt
  if isfield(args, 'run')
    chart = coco_read_solution(args.run, args.lab, 'chart');
    cdata = coco_get_chart_data(chart, 'lsol');
    [lo_chart, lo_lidx] = coco_read_adjoint(loid, args.run, ...
      args.lab, 'chart', 'lidx');
    lo_l0 = lo_chart.x;

    [up_chart, up_lidx] = coco_read_adjoint(upid, args.run, ...
      args.lab, 'chart', 'lidx');
    up_l0 = up_chart.x;

    [unif_chart, unif_lidx] = coco_read_adjoint(uniform_sample_id, ...
      args.run, args.lab, 'chart', 'lidx');
    unif_l0 = unif_chart.x;
    if add_tl0
      lo_tl0 = cdata.v(lo_lidx);
      up_tl0 = cdata.v(up_lidx);
      unif_tl0 = cdata.v(unif_lidx);
    end
  else
    lo_l0 = zeros(size(lo));
    up_l0 = zeros(size(up));
    unif_l0 = zeros(size(data.uniform_nds_idx'));
  end
  if add_tl0
    prob = coco_add_adjt(prob, loid, coco_get_id('d', lonames), ...
      'l0', lo_l0, 'tl0', lo_tl0);
    prob = coco_add_adjt(prob, upid, coco_get_id('d', upnames), ...
      'l0', up_l0, 'tl0', up_tl0);
    aloidx = coco_get_adjt_data(prob, loid, 'axidx');
    aupidx = coco_get_adjt_data(prob, upid, 'axidx');
    prob = coco_add_adjt(prob, uniform_sample_id, 'aidx', ...
      [aloidx; aupidx], 'l0', unif_l0, 'tl0', unif_tl0);
  else
    prob = coco_add_adjt(prob, loid, coco_get_id('d', lonames), ...
      'l0', lo_l0);
    prob = coco_add_adjt(prob, upid, coco_get_id('d', upnames), ...
      'l0', up_l0);
    aloidx = coco_get_adjt_data(prob, loid, 'axidx');
    aupidx = coco_get_adjt_data(prob, upid, 'axidx');
    prob = coco_add_adjt(prob, uniform_sample_id, 'aidx', ...
      [aloidx; aupidx], 'l0', unif_l0);
  end
end
uidx = coco_get_func_data(prob, uniform_sample_id, 'uidx');
data.sample_par_idx(data.uniform_nds_idx) = ...
    uidx((2*data.num_uniforms+1):end);
if data.addadjt
  aidx = coco_get_adjt_data(prob, uniform_sample_id, 'axidx');
  data.adjt_sample_par_idx(data.uniform_nds_idx) = ...
    aidx((2*data.num_uniforms+1):end);
  end
end

seg2ndsid = coco_get_id(data.oid, 'uq.s_par_glue');
seg_par_idx = cell2mat(data.s_idx);
seg_par_idx = seg_par_idx(data.sp2p_idx, :);
seg_par_idx = seg_par_idx(:);

seg2ndsidx = [seg_par_idx; data.sample_par_idx];
prob = coco_add_func(prob, seg2ndsid, @uq_seg_to_nds, @uq_seg_to_nds_du, ...
  @uq_seg_to_nds_dudu, data, 'zero', 'uidx', seg2ndsidx);
pnames = uq_get_sample_par_names(data.pnames(data.sp2p_idx), 1:data.nsamples);
nds2smplid = coco_get_id(data.oid, 'uq.nds2samples');

prob = coco_add_pars(prob, nds2smplid, seg_par_idx, pnames, 'active');

if data.addadjt
  adjt_seg_par_idx = cell2mat(data.adj_s_idx);
  adjt_seg_par_idx = adjt_seg_par_idx(data.sp2p_idx, :);
  adjt_seg_par_idx = adjt_seg_par_idx(:);

  if isfield(args, 'run')
    chart = coco_read_solution(args.run, args.lab, 'chart');
    cdata = coco_get_chart_data(chart, 'lsol');
    [s2n_chart, s2n_lidx] = coco_read_adjoint(seg2ndsid, args.run, args.lab, ...
      'chart', 'lidx');
    s2n_l0 = s2n_chart.x;
    [n2s_chart, n2s_lidx] = coco_read_adjoint(nds2smplid, args.run, args.lab, ...
      'chart', 'lidx');
    n2s_l0 = n2s_chart.x;

    if add_tl0
      s2n_tl0 = cdata.v(s2n_lidx);
      prob = coco_add_adjt(prob, seg2ndsid, ...
        'aidx', [adjt_seg_par_idx; data.adjt_sample_par_idx], ...
        'l0', s2n_l0, ...
        'tl0', s2n_tl0);

      n2s_tl0 = cdata.v(n2s_lidx);
      prob = coco_add_adjt(prob, nds2smplid, coco_get_id('d',pnames), ...
        'aidx', adjt_seg_par_idx, ...
        'l0', n2s_l0, ...
        'tl0', n2s_tl0);
    else
      prob = coco_add_adjt(prob, seg2ndsid, ...
        'aidx', [adjt_seg_par_idx; data.adjt_sample_par_idx], ...
        'l0', s2n_l0);

      prob = coco_add_adjt(prob, nds2smplid, coco_get_id('d',pnames), ...
        'aidx', adjt_seg_par_idx, ...
        'l0', n2s_l0);
    end
  else
    prob = coco_add_adjt(prob, seg2ndsid, 'aidx', ...
      [adjt_seg_par_idx; data.adjt_sample_par_idx]);
    prob = coco_add_adjt(prob, nds2smplid, coco_get_id('d',pnames), ...
      'aidx', adjt_seg_par_idx);
  end
end

end
```

```
function [data, y] = uq_normal_sample(prob, data, u)

mu = u(1:data.num_normals);
sig = u(data.num_normals+1:2*data.num_normals);
transformed_node_locations = u(2*data.num_normals+1:end);

mu_rep = repelem(mu, data.uq.M(data.normal_par_idx));
```

```
sig_rep = repelem(sig, data.uq.M(data.normal_par_idx));

if size(mu_rep,1) == 1
    mu_rep = mu_rep';
    sig_rep = sig_rep';
end

st_nds = data.st_nds(data.normal_nds_idx);
vals = mu_rep + sig_rep.*st_nds;

y = vals - transformed_node_locations;

end
```

```
function [data, J] = uq_normal_sample_dU(prob, data, u)

numnodes = numel(u) - 2*data.num_normals;
num_us = size(u, 1);
J = zeros(numnodes, num_us);
J_spdp = ones(numnodes/data.num_normals,2);
J_spdp(:,2) = data.st_nds(1:numnodes/data.num_normals);
J_spdp(:,1) = 1;
start_nodes = 2*data.num_normals + 1;
J(:, 1:data.num_normals) = kron(eye(data.num_normals), J_spdp(:,1));
J(:, data.num_normals+1:2*data.num_normals) = kron(eye(data.num_normals), J_spdp(:,2));
J(:, start_nodes:end) = -eye(numnodes);

end
```

```
function [data, dJ] = uq_normal_sample_dUdU(prob, data, u)

numnodes = numel(u) - 2*data.num_normals;
num_us = size(u, 1);
dJ = zeros(numnodes, num_us, num_us);

end
```

```
function [data, y] = uq_uniform_sample(prob, data, u)

lo = u(1:data.num_uniforms);
hi = u(data.num_uniforms+1:2*data.num_uniforms);

transformed_node_locations = u(2*data.num_uniforms+1:end);

lo_rep = repelem(lo, data.uq.M(data.uniform_par_idx));
up_rep = repelem(hi, data.uq.M(data.uniform_par_idx));

if size(lo_rep,1) == 1
    lo_rep = lo_rep';
    up_rep = up_rep';
end

st_nds = data.st_nds(data.uniform_nds_idx);

vals = (up_rep.*(st_nds+1)+lo_rep.*(1-st_nds))/2;

y = vals - transformed_node_locations;

end
```

```
function [data, J] = uq_uniform_sample_dU(prob, data, u)

numnodes = numel(u) - 2*data.num_uniforms;
num_us = size(u, 1);
J = zeros(numnodes, num_us);
J_spdp = ones(numnodes/data.num_uniforms,2);
```

```
J_spdp(:,1) = (1-data.st_nds(1:numnodes/data.num_uniforms))/2;
J_spdp(:,2) = (1+data.st_nds(1:numnodes/data.num_uniforms))/2;

start_nodes = 2*data.num_uniforms + 1;
J(:, 1:data.num_uniforms) = kron(eye(data.num_uniforms), J_spdp(:,1));
J(:, data.num_uniforms+1:2*data.num_uniforms) = kron(eye(data.num_uniforms), J_spdp(:,2));
J(:, start_nodes:end) = -eye(numnodes);

end
```

```
function [data, dJ] = uq_uniform_sample_dUdU(prob, data, u)

numnodes = numel(u) - 2*data.num_uniforms;
num_us = size(u, 1);
dJ = zeros(numnodes, num_us, num_us);


end
```

```
function [data, y] = uq_seg_to_nds(prob, data, u)

seg_pars  = u(1:numel(data.idx));
nds_start = numel(data.idx) + 1;
node_pars = u(nds_start:end);
node_pars = node_pars(data.idx);
node_pars = node_pars(:);

y = seg_pars - node_pars;

end
```

```
function [data, J] = uq_seg_to_nds_du(prob, data, u)

negs = data.idx+numel(data.idx);
negs = negs(:);
i = repmat(1:numel(data.idx),[1,2]);
j = [1:numel(data.idx), negs'];
v = [ones(1,numel(data.idx)), -ones(1,numel(data.idx))];
J = sparse(i, j, v);

end
```

```
function [data, dJ] = uq_seg_to_nds_dudu(prob, data, u)

dJ = zeros(numel(data.idx), numel(u), numel(u));

end
```

B.1.2 Construction from a Stored Branch Point solution

```
function prob = uq_BP2bvp(prob, sid, varargin)

grammar   = 'RUN [SOID] LAB [OPTS]';
args_spec = {
     'RUN', 'cell', '{str}',  'run',  {}, 'read', {}
    'SOID',     '',   'str', 'soid', sid, 'read', {}
     'LAB',     '',   'num',  'lab',  [], 'read', {}
  };
opts_spec = {
  '-add-adjt', 'addadjt', false, 'toggle', {}
  '-add-resp', 'addresp', false, 'toggle', {}
  };

[args, opts] = coco_parse(grammar, args_spec, ...
  opts_spec, varargin{:});

responses = uq_get_responses(args.run);

data = coco_read_solution(responses{1}, args.run, ...
  args.lab, 'data');

switch data.response_type
  case 'bv'
    fields = {'response_type', 'rhan', 'drduhan', ...
      'drduduhan', 'resp'};
  case 'int'
    fields = {'response_type', 'rhan', 'drdxhan', ...
      'drdphan', 'drdxdxhan', 'drdpdphan', ...
      'drdxdphan', 'resp'};
end
data = rmfield(data, fields);

bc_data = coco_read_solution(coco_get_id(sid, 'uq.sample1.bc'), ...
  args.run, args.lab, 'data');

if data.num_normals > 0
  muid = coco_get_id(sid, 'mus');
  mu_chart = coco_read_solution(muid, args.run, args.lab, 'chart');
  mus = mu_chart.x;
  sigid = coco_get_id(sid, 'sigs');
  sig_chart = coco_read_solution(sigid, args.run, args.lab, 'chart');
  sigs = sig_chart.x;
  data.spdp(data.normal_par_idx,:) = [mus, sigs];
end

if data.num_uniforms > 0
  lsid = coco_get_id(sid, 'lo');
  lo_chart = coco_read_solution(lsid, args.run, args.lab, 'chart');
  los = lo_chart.x;
  upid = coco_get_id(sid, 'up');
  up_chart = coco_read_solution(upid, args.run, args.lab, 'chart');
  ups = up_chart.x;
  data.spdp(data.uniform_par_idx,:) = [los, ups];
end

data = uq_gen_nds(data);
[prob, data] = uq_BP2bvpsamples(prob, data, bc_data, args, opts);
psi_mat = uq_make_psi_mat(data.nds_grid, data.uq.Pt, data.spdists);
data.wtd_psi_mat = psi_mat*diag(data.wts);

prob = uq_add_sample_nodes(prob, data, args);

if opts.addresp
  for i=1:numel(responses)
      data = coco_read_solution(responses{i}, ...
        args.run, args.lab, 'data');

      switch data.response_type
        case 'bv'
          r = data.rhan;
          dr = data.drduhan;
          ddr = data.drduduhan;
          resp_args = {data.response_type, r, dr, ddr};
        case 'int'
          r = data.rhan;
          drdx = data.drdxhan;
          drdp = data.drdphan;
          drdxdx = data.drdxdxhan;
          drdpdp = data.drdxdphan;
          drdxdp = data.drdpdphan;
          resp_args = {data.response_type, r,...
            drdx, drdp, drdxdx, drdpdp, drdxpdp};
      end
      r = strsplit(responses{i}, '.uq.responses.');
      rid = strsplit(r{2}, '.');
      rid = rid{1};
      prob = uq_coll_add_response(prob, sid, rid, resp_args{:});
    if opts.addadjt
      prob = uq_coll_add_response_adjoint(prob, sid, rid, args);
    end
  end
end

end
```

```
function responses = uq_get_responses(varargin)

switch numel(varargin)
  case 1
    sid = '';
    runid = varargin{1};
  case 2
    sid = varargin{1};
    runid = varargin{2};
  otherwise
   error('Expected 1 or 2 input arguments: Sample ID (optional), Run ID (required)')
end

data = coco_read_solution(runid, 1, 'data');
resp_idx = strfind(data(:,1), coco_get_id(sid, 'uq.responses'));

responses = {};

for i=1:numel(resp_idx)
  if ~isempty(resp_idx{i})
    responses = {responses{:}, data{i,1}};
  end
end

end
```

```
function [prob, data] = uq_BP2bvpsamples(prob, data, bc_data, args, opts)

[prob, data] = uq_BP2collsamples(prob, data, args, opts);
[prob, data] = uq_bvp_close_samples(prob, data, bc_data, args);

end
```

```
function [prob, data] = uq_BP2collsamples(prob, data, args, opts)

if nargin < 4
  opts = {};
  opts.addadjt=0;
end

reorder_idx = zeros(size(data.wts));

for sample_count = 1:numel(data.sids)
  prob = ode_BP2coll(prob, data.sids{sample_count}, args.run, args.lab);
  if opts.addadjt
    prob = adjt_BP2coll(prob, data.sids{sample_count}, args.run, args.lab);
  end

  sol = coll_read_solution(data.sids{sample_count}, args.run, args.lab);

  ndvals = data.nds(data.idx);
  if size(ndvals,2) == 1
      ndvals = ndvals';
  end
  p_rep = repmat(sol.p(data.sp2p_idx), 1, prod(data.uq.M, 2));
  nd_diff = sum((ndvals - p_rep).^2,1);
  reorder_idx(sample_count) = find(nd_diff==min(nd_diff));
end

data.idx = data.idx(:, reorder_idx);
data.wts = data.wts(reorder_idx);
data.nds_grid = data.nds_grid(:, reorder_idx);
data.reorder_idx = reorder_idx;

end
```

B.2 Response Functions

The functions in this section support addition of response functions and their adjoints. Functions called by the constructors are included in the order that they are called.

B.2.1 Response Function Construction

```
function prob = uq_coll_add_response(prob, oid, rid, varargin)


err_string = ['Expected either ''bv''', ...
              ' or ''int'' for fourth argument entry'];

if ischar(varargin{1})
  if strcmp(varargin{1}, 'bv') || strcmp(varargin{1}, 'int')
    response_type = varargin{1};
  else
    assert(strcmp(varargin{1}, 'bv') || strcmp(varargin{1}, 'int'), err_string);
  end
else
  assert(ischar(varargin{1}), err_string);
end

if strcmp(response_type,'int')
  grammar   = 'R [DRDX [DRDP [DRDXDX [DRDXDP [DRDPDP]]]]] [OPTS]';
  args_spec = {
           'R',     '',     '@',       'rhan',       [], 'read', {}
        'DRDX',     '',  '@|[]',    'drdxhan',       [], 'read', {}
        'DRDP',     '',  '@|[]',    'drdphan',       [], 'read', {}
      'DRDXDX',     '',  '@|[]',  'drdxdxhan',       [], 'read', {}
      'DRDXDP',     '',  '@|[]',  'drdxdphan',       [], 'read', {}
      'DRDPDP',     '',  '@|[]',  'drdpdphan',       [], 'read', {}
    };
elseif strcmp(response_type,'bv')
  grammar   = 'R [DRDU [DRDUDU]] [OPTS]';
  args_spec = {
         'R',     '',     '@',       'rhan',       [], 'read', {}
      'DRDU',     '',  '@|[]',    'drduhan',       [], 'read', {}
    'DRDUDU',     '',  '@|[]',  'drduduhan',       [], 'read', {}
    };
end

[args, ~] = coco_parse(grammar, args_spec, {}, varargin{2:end});

seg2ndsid = coco_get_id(oid, 'uq.s_par_glue');
uq_data = coco_get_func_data(prob, seg2ndsid, 'data');
uq_data.response_type = response_type;

names = fieldnames(args);
for i=1:numel(names)
  uq_data.(names{i}) = args.(names{i});
end

```

```
uq_data = uq_coll_response_get_idx(prob, uq_data);

u = prob.efunc.x0(uq_data.resp.uidx);
[~, r] = uq_response_evaluation(prob, uq_data, u);
alpha_ig = uq_data.wtd_psi_mat*r;

response_id = coco_get_id(oid, 'uq', 'responses');
pce_id = coco_get_id(response_id, rid, 'pce');

if strcmp(uq_data.response_type, 'bv')
  prob = coco_add_func(prob, pce_id, ...
      @uq_pce_coefficients, @uq_pce_coefficients_dU, ...
      @uq_pce_coefficients_dUdU, ...
      uq_data, 'zero', 'uidx', ...
      uq_data.resp.uidx, 'u0', alpha_ig);
elseif strcmp(uq_data.response_type, 'int')
  prob = coco_add_func(prob, pce_id, ...
      @uq_pce_coefficients, @uq_pce_coefficients_dU, ...
      uq_data, 'zero', 'uidx', ...
      uq_data.resp.uidx, 'u0', alpha_ig);
end
prob = coco_add_slot(prob, pce_id, @coco_save_data, uq_data, 'save_full');

alpha_idx = coco_get_func_data(prob, pce_id, 'uidx');

alpha_idx = alpha_idx(end-uq_data.Nt+1:end);


mean_id = coco_get_id(response_id, rid, 'pce_mean');
prob = coco_add_func(prob, mean_id, ...
  @uq_pce_mean, @uq_pce_mean_dU, @uq_pce_mean_dUdU, ...
  uq_data, 'zero', 'uidx', alpha_idx, 'u0', alpha_ig(1));

mean_par_id = coco_get_id(response_id, rid, 'mean');
mean_idx = coco_get_func_data(prob, mean_id, 'uidx');
mean_name = coco_get_id(uq_data.oid, rid, 'mean');
prob = coco_add_pars(prob, mean_par_id, mean_idx(end), mean_name);

var_id = coco_get_id(response_id, rid, 'pce_variance');
prob = coco_add_func(prob, var_id, ...
  @uq_pce_variance, @uq_pce_variance_dU, @uq_pce_variance_dUdU,...
  uq_data, 'zero', 'uidx', alpha_idx, 'u0', sum(alpha_ig(2:end).^2));

var_par_id = coco_get_id(response_id, rid, 'variance');
var_idx = coco_get_func_data(prob, var_id, 'uidx');
var_name = coco_get_id(uq_data.oid, rid, 'var');
prob = coco_add_pars(prob, var_par_id, var_idx(end), var_name);

end
```

```
function data = uq_coll_response_get_idx(prob, data)

  if strcmp(data.response_type,'int')
    data = uq_int_coll_response_get_idx(prob, data);
  elseif strcmp(data.response_type,'bv')
    data = uq_bv_coll_response_get_idx(prob, data);
  end
end

function data = uq_bv_coll_response_get_idx(prob, data)

nsamples = data.nsamples;

fdata = coco_get_func_data(prob, coco_get_id(data.sids{1}, 'coll'), ...
  'data');

data.resp.uidx = zeros(nsamples, 2 + 2*fdata.xdim + fdata.pdim);

for i=1:nsamples
```

```
  [uidx, fdata] = coco_get_func_data(prob, ...
    coco_get_id(data.sids{i}, 'coll'), 'uidx', 'data');
  maps = fdata.coll_seg.maps;

  data.resp.uidx(i,1) = uidx(maps.T0_idx);
  data.resp.uidx(i,2) = uidx(maps.T_idx);
  data.resp.uidx(i,3:2+fdata.xdim) = uidx(maps.x0_idx);
  data.resp.uidx(i,2+fdata.xdim+1:2+2*fdata.xdim) = uidx(maps.x1_idx);
  data.resp.uidx(i,2+2*fdata.xdim+1:2+2*fdata.xdim+fdata.pdim) = uidx(maps.p_idx);

end
data.resp.uidx = data.resp.uidx';
data.resp.uidx = data.resp.uidx(:);

data.resp.nargs  = (nargin(data.rhan)==5);

end


function data = uq_int_coll_response_get_idx(prob, data)

nsamples = data.nsamples;

T_idx = cell(1, nsamples);
xbp_idx = cell(1, nsamples);
p_idx = cell(1, nsamples);

data.resp.T_idx = ones(nsamples,1);
data.resp.xcn_idx = ones(nsamples,2);
data.resp.xbp_idx = ones(nsamples,2);
data.resp.p_idx = ones(nsamples, 2);
adjt_size = 0;
for i=1:nsamples
  [uidx, fdata] = coco_get_func_data(prob, ...
    coco_get_id(data.sids{i}, 'coll'), 'uidx', 'data');
  maps = fdata.coll_seg.maps;
  data.resp.T_idx(i) = i;
  data.resp.xcn_idx(i,2) = size(maps.W,1);
  data.resp.xbp_idx(i,2) = size(maps.W,2);
  data.resp.p_idx(i,2) = size(maps.p_idx,1);
  adjt_size = adjt_size + 1 + size(maps.W,1) + size(maps.p_idx,1);

  T_idx{i} = uidx(maps.T_idx);
  xbp_idx{i} = uidx(maps.xbp_idx);
  p_idx{i} = uidx(maps.p_idx);

end

data.resp.xcn_idx(:,2) = cumsum(data.resp.xcn_idx(:,2));
data.resp.xcn_idx(2:end,1) = data.resp.xcn_idx(2:end,1) + data.resp.xcn_idx(1:end-1,2);
data.resp.xcn_idx = data.resp.xcn_idx + data.resp.T_idx(end);
data.resp.adjt_size = adjt_size;

data.resp.xbp_idx(:,2) = cumsum(data.resp.xbp_idx(:,2));
data.resp.xbp_idx(2:end,1) = data.resp.xbp_idx(2:end,1) + data.resp.xbp_idx(1:end-1,2);
data.resp.xbp_idx = data.resp.xbp_idx + data.resp.T_idx(end);

data.resp.p_idx(:,2) = cumsum(data.resp.p_idx(:,2));
data.resp.p_idx(2:end,1) = data.resp.p_idx(2:end,1) + data.resp.p_idx(1:end-1,2);
data.resp.p_adjt_idx = data.resp.p_idx + data.resp.xcn_idx(end);
data.resp.p_idx = data.resp.p_idx + data.resp.xbp_idx(end);

data.resp.uidx = vertcat(T_idx{:}, xbp_idx{:}, p_idx{:});

end
```

```
function [data, r] = uq_response_evaluation(prob, data, u)


```

```
if strcmp(data.response_type, 'int')
  for i=1:data.nsamples
    T = u(data.resp.T_idx(i,1));
    x = u(data.resp.xbp_idx(i,1):data.resp.xbp_idx(i,end));
    p = u(data.resp.p_idx(i,1):data.resp.p_idx(i,end));
    fdata = coco_get_func_data(prob, ...
      coco_get_id(data.sids{i}, 'coll'), 'data');

    if i == 1
      r1 = int_resp(data, fdata, T, x, p);
      r = zeros(size(r1,2), data.nsamples);
      r(:,i) = r1;
    else
      r(:,i) = int_resp(data, fdata, T, x, p);
    end
  end
elseif strcmp(data.response_type, 'bv')

  xdim = data.xdim;
  pdim = data.pdim;
  ns = data.nsamples;
  nu = 2 + 2*xdim + pdim;

  idx = reshape(1:nu*ns, [], ns);
  T0  = u(idx(1,:));
  T   = u(idx(2,:));
  x0  = u(idx(3:2+xdim,:));
  x1  = u(idx(3+xdim:2*(1+xdim),:));
  p   = u(idx(3+2*xdim:end,:));

  args = {T0, T, x0, x1, p};

  r = data.rhan(data, args{data.resp.nargs + 1:end});

end

r = r';
end

function r = int_resp(uq_data, fdata, T, x, p)


maps = fdata.coll_seg.maps;
int  = fdata.coll_seg.int;
xx   = reshape(maps.W*x, maps.x_shp);
pp   = repmat(p, maps.p_rep);

resp = uq_data.rhan(xx, pp);
wts1 = repmat(int.wt',[maps.NTST,1]);

r = T/(2*maps.NTST)*resp*wts1;

end

function r = int_resp_nu(uq_data, fdata, T, x, p)
maps = fdata.coll_seg.maps;
mesh = fdata.coll_seg.mesh;

xcn = reshape(maps.W*x, maps.x_shp);
pp  = repmat(p, maps.p_rep);

rcn = uq_data.rhan(xcn, pp);
rcn = mesh.gka.*rcn;


r = (0.5*T/maps.NTST)*mesh.gwt*rcn';

end
```

```
function [data, y] = uq_pce_coefficients(prob, data, u)

[~, r] = uq_response_evaluation(prob, data, u);

alphas = u(end-data.Nt+1:end);
R = data.wtd_psi_mat*r;

y = alphas - R;

end
```

```
function [data, J] = uq_pce_coefficients_dU(prob, data, u)


[~, dr] = uq_response_Jacobian(prob, data, u);

J = [dr, ...
     speye(data.Nt)];

end
```

```
function [data, J] = uq_response_Jacobian(prob, data, u)

  if strcmp(data.response_type, 'int')

    J = zeros(prod(data.uq.M), data.resp.p_idx(end));

    for i=data.nsamples:-1:1
      T = u(data.resp.T_idx(i,1));
      x = u(data.resp.xbp_idx(i,1):data.resp.xbp_idx(i,end));
      p = u(data.resp.p_idx(i,1):data.resp.p_idx(i,end));
      fdata = coco_get_func_data(prob, ...
        coco_get_id(data.sids{i}, 'coll'), 'data');
      [J_T, J_xbp, J_p] = int_resp_du(data, fdata, T, x, p);

      J(i, i) = J_T;
      J(i, data.resp.xbp_idx(i,1):data.resp.xbp_idx(i,2)) = J_xbp;
      J(i, data.resp.p_idx(i,1):data.resp.p_idx(i,2)) = J_p;
    end

    J = -1*data.wtd_psi_mat*J;

  elseif strcmp(data.response_type, 'bv')

    xdim = data.xdim;
    pdim = data.pdim;
    ns = data.nsamples;
    nu = 2 + 2*xdim + pdim;

    idx = reshape(1:nu*ns, [], ns);
    T0  = u(idx(1,:));
    T   = u(idx(2,:));
    x0  = u(idx(3:2+xdim,:));
    x1  = u(idx(3+xdim:2*(1+xdim),:));
    p   = u(idx(3+2*xdim:end,:));

    args = {T0, T, x0, x1, p};
    J = data.drduhan(data, args{data.resp.nargs + 1:end});
    rows = repelem(1:ns, 1, nu);
    cols = 1:ns*nu;
    J = sparse(rows, cols, J);
    J = -1*data.wtd_psi_mat*J;

  end
end

function [J_T, J_xbp, J_p] = int_resp_du(uq_data, fdata, T, x, p)

```

```
maps = fdata.coll_seg.maps;
int  = fdata.coll_seg.int;

xx  = reshape(maps.W*x, maps.x_shp);
pp  = repmat(p, maps.p_rep);

wts1    = repmat(int.wt',[maps.NTST,1]);
wts2    = diag(kron(ones(1,maps.NTST), kron(int.wt, ones(1,int.dim))));

resp  = uq_data.rhan(xx, pp);
drdx = uq_data.drdxhan(xx, pp);
drdp = uq_data.drdphan(xx, pp);

J_T   = 1/(2*maps.NTST)*resp*wts1;
J_xbp = T/(2*maps.NTST)*drdx(:)'*wts2*maps.W;
J_p   = T/(2*maps.NTST)*drdp(:)'*kron(wts1,eye(maps.pdim));

end

function [J_T, J_xbp, J_p] = int_resp_du_num(uq_data, fdata, T, x, p)
maps = fdata.coll_seg.maps;
mesh = fdata.coll_seg.mesh;

xx = reshape(maps.W*x, maps.x_shp);
pp  = repmat(p, maps.p_rep);

gcn = uq_data.rhan(xx, pp);
gcn = mesh.gka.*gcn;

gdxcn = uq_data.drdxhan(xx, pp);
gdxcn = mesh.gdxka.*gdxcn;

gdxcn = sparse(maps.gdxrows, maps.gdxcols, gdxcn(:));

gdp = uq_data.drdphan(xx, pp);
gdp = mesh.gdpka.*gdp;
gdprows = kron(ones(maps.pdim,1), 1:maps.p_rep(2));
gpdcols = kron(ones(maps.p_rep),1:maps.pdim);
gdp = sparse(gdprows, gpdcols, gdp(:));

J_T   = (0.5/maps.NTST)*mesh.gwt*gcn';
J_xbp = (0.5*T/maps.NTST)*mesh.gwt*gdxcn*maps.W;
J_p   = (0.5*T/maps.NTST)*mesh.gwt*gdp;

end
```

```
function [data, dJ] = uq_pce_coefficients_dUdU(prob, data, u)

  xdim = data.xdim;
  pdim = data.pdim;
  ns = data.nsamples;
  nu = 2 + 2*xdim + pdim;

  dJ = zeros(ns, ns*nu + data.Nt, ...
  data.nsamples*nu + data.Nt);

  idx = reshape(1:nu*ns, [], ns);
  T0  = u(idx(1,:));
  T   = u(idx(2,:));
  x0  = u(idx(3:2+xdim,:));
  x1  = u(idx(3+xdim:2*(1+xdim),:));
  p   = u(idx(3+2*xdim:end,:));

  args = {T0, T, x0, x1, p};

  drdudu = data.drduduhan(data, ...
    args{data.resp.nargs + 1:end});
  for i=1:data.nsamples
    idx1 = (i-1)*nu+1;
    idx2 = i*nu;
    dJ(i,idx1:idx2,idx1:idx2) = reshape(drdudu(:,:,i), ...
      [1, nu, nu]);
  end

  dJ = -1*data.wtd_psi_mat*reshape(dJ, ns, []);
  dJ = reshape(dJ, data.Nt, ns*nu + data.Nt, ns*nu + data.Nt);

end
```

```
function [data, y] = uq_pce_mean(prob, data, u)

alpha0 = u(1);
mu = u(end);
y = mu - alpha0;

end
```

```
function [data, J] = uq_pce_mean_dU(prob, data, u)

J = zeros(size(u'));
J(1) = -1;
J(end) = 1;

end
```

```
function [data, dJ] = uq_pce_mean_dUdU(prob, data, u)

dJ = zeros(1, numel(u), numel(u));

end
```

```
function [data, y] = uq_pce_variance(prob, data, u)

alphas = u(1:end-1);
alphas = reshape(alphas, [], data.Nt);
variance = u(end);

y = variance - sum(alphas(:,2:end).^2,2);

end
```

```
function [data, J] = uq_pce_variance_dU(prob, data, u)

J = zeros(1, numel(u));
J(2:end-1) = -2*u(2:end-1);
J(end) = 1;

end
```

```
function [data, dJ] = uq_pce_variance_dUdU(prob, data, u)

dJ = zeros(1, numel(u), numel(u));
dJ(1, 2:end-1, 2:end-1) = -2*eye(data.Nt - 1);

end
```

B.2.2 Response Function Adjoint Construction

```
function prob = uq_coll_add_response_adjoint(prob, oid, rid, varargin)

if nargin < 4
```

```
  args = struct();
else
  args = varargin{1};
end

response_id = coco_get_id(oid, 'uq', 'responses');
pce_id = coco_get_id(response_id, rid, 'pce');
uq_data = coco_get_func_data(prob, pce_id, 'data');

add_tl0 = 0;

if isfield(args, 'run')
  chart = coco_read_solution(args.run, args.lab, 'chart');

    cdata = coco_get_chart_data(chart, 'lsol');
    [chart, lidx] = coco_read_adjoint(pce_id, args.run, args.lab, ...
      'chart', 'lidx');
    l0 = chart.x;
  if strcmpi(chart.pt_type,'BP')
    tl0 = cdata.v(lidx);
    add_tl0 = 1;
  end
else
  l0 = zeros(uq_data.Nt,1);
end

if strcmp(uq_data.response_type, 'int')

  aTidx = cell(uq_data.nsamples,1);
  axidx = cell(uq_data.nsamples,1);
  apidx = cell(uq_data.nsamples,1);
  for i=1:uq_data.nsamples
    coll_id = coco_get_id(uq_data.sids{i}, 'coll');
    [adata, aidx] = coco_get_adjt_data(prob, coll_id, 'data', 'axidx');
    aTidx{i} = aidx(adata.coll_opt.T_idx);
    axidx{i} = aidx(adata.coll_opt.xcn_idx);
    apidx{i} = aidx(adata.coll_opt.p_idx);
  end
  aTidx = cell2mat(aTidx);
  axidx = cell2mat(axidx);
  apidx = cell2mat(apidx);

  aidx = [aTidx;axidx;apidx];
  if add_tl0
    prob = coco_add_adjt(prob, pce_id, ...
      @uq_pce_coefficients_adjoint, ...
      @uq_pce_coefficients_adjoint_dU, uq_data, ...
      'aidx', aidx, 'l0', l0, 'tl0', tl0);
  else
    prob = coco_add_adjt(prob, pce_id, ...
      @uq_pce_coefficients_adjoint, ...
      @uq_pce_coefficients_adjoint_dU, uq_data, ...
      'aidx', aidx, 'l0', l0);
  end

elseif strcmp(uq_data.response_type, 'bv')

  nsamples = uq_data.nsamples;
  xdim = uq_data.xdim;
  pdim = uq_data.pdim;
  aidx = zeros(nsamples, 2 + 2*xdim + pdim);
  x0end = 2+xdim;
  x1end = 2+2*xdim;
  pend  = 2+2*xdim+pdim;

  for i=1:nsamples
    [axidx, adata] = coco_get_adjt_data(prob, ...
      coco_get_id(uq_data.sids{i}, 'coll'), 'axidx', 'data');
    maps = adata.coll_opt;

    aidx(i,1) = axidx(maps.T0_idx);
    aidx(i,2) = axidx(maps.T_idx);
    aidx(i,3:x0end) = axidx(maps.x0_idx);
    aidx(i,x0end+1:x1end) = axidx(maps.x1_idx);
    aidx(i,x1end+1:pend) = axidx(maps.p_idx);
  end

  aidx = aidx';
  aidx = aidx(:);
  if add_tl0
    prob = coco_add_adjt(prob, pce_id, 'aidx', ...
      aidx, 'l0', l0, 'tl0', tl0);
  else
    prob = coco_add_adjt(prob, pce_id, 'aidx', ...
      aidx, 'l0', l0);
  end
end

dalpha_aidx = coco_get_adjt_data(prob, pce_id, 'axidx');
dalpha_aidx = dalpha_aidx(end-uq_data.Nt+1:end);

mean_id = coco_get_id(response_id, rid, 'pce_mean');
mean_par_id = coco_get_id(response_id, rid, 'mean');

if isfield(args, 'run')
  chart = coco_read_solution(args.run, args.lab, 'chart');
  cdata = coco_get_chart_data(chart, 'lsol');

  [mean_chart, mean_lidx] = coco_read_adjoint(mean_id, args.run, ...
    args.lab, 'chart', 'lidx');
  mean_l0 = mean_chart.x;

  [mean_par_chart, mean_par_lidx] = coco_read_adjoint(mean_par_id, ...
    args.run, args.lab, 'chart', 'lidx');
  mean_par_l0 = mean_par_chart.x;

  if add_tl0
    mean_tl0 = cdata.v(mean_lidx);
    mean_par_tl0 = cdata.v(mean_par_lidx);
  end
else
  mean_l0 = 0;
  mean_par_l0 = 0;
end

if add_tl0
  prob = coco_add_adjt(prob, mean_id, 'aidx', dalpha_aidx, ...
    'l0', mean_l0, 'tl0', mean_tl0);
else
  prob = coco_add_adjt(prob, mean_id, 'aidx', dalpha_aidx, ...
    'l0', mean_l0);
end
mean_aidx = coco_get_adjt_data(prob, mean_id, 'axidx');
mean_name = coco_get_id(uq_data.oid, rid, 'mean');
dmean = coco_get_id('d', mean_name);

if add_tl0
  prob = coco_add_adjt(prob, mean_par_id, dmean, 'aidx', mean_aidx(end),...
    'l0', mean_par_l0, 'tl0', mean_par_tl0);
else
  prob = coco_add_adjt(prob, mean_par_id, dmean, 'aidx', mean_aidx(end),...
    'l0', mean_par_l0);
end

var_id = coco_get_id(response_id, rid, 'pce_variance');
var_par_id = coco_get_id(response_id, rid, 'variance');

if isfield(args, 'run')
  chart = coco_read_solution(args.run, args.lab, 'chart');
  cdata = coco_get_chart_data(chart, 'lsol');

  [var_chart, var_lidx] = coco_read_adjoint(var_id, args.run, ...
    args.lab, 'chart', 'lidx');
  var_l0 = var_chart.x;


  [var_par_chart, var_par_lidx] = coco_read_adjoint(var_par_id, ...
    args.run, args.lab, 'chart', 'lidx');
  var_par_l0 = var_par_chart.x;
  if add_tl0
    var_tl0 = cdata.v(var_lidx);
    var_par_tl0 = cdata.v(var_par_lidx);
  end
else
  var_l0 = 0;
  var_par_l0 = 0;
end

if add_tl0
  prob = coco_add_adjt(prob, var_id, 'aidx', dalpha_aidx, ...
    'l0', var_l0, 'tl0', var_tl0);
else
  prob = coco_add_adjt(prob, var_id, 'aidx', dalpha_aidx, ...
  'l0', var_l0);
end
var_aidx = coco_get_adjt_data(prob, var_id, 'axidx');
var_name = coco_get_id(uq_data.oid, rid, 'var');
dvar = coco_get_id('d', var_name);
if add_tl0
  prob = coco_add_adjt(prob, var_par_id, dvar, 'aidx', var_aidx(end), ...
    'l0', var_par_l0, 'tl0', var_par_tl0);
else
  prob = coco_add_adjt(prob, var_par_id, dvar, 'aidx', var_aidx(end), ...
    'l0', var_par_l0);
end

end
```

```
function [data, J] = uq_pce_coefficients_adjoint(prob, data, u)

[~, dr] = uq_int_response_adjoint_evaluation(prob, data, u);

J = [dr, ...
     speye(data.Nt)];

end
```

```
function [data, J] = uq_pce_coefficients_adjoint_dU(prob, data, u)

J = zeros(data.Nt, data.resp.p_adjt_idx(end) + data.Nt, ...
  data.resp.p_idx(end) + data.Nt);
[~, ddr] = uq_int_response_adjoint_Jacobian(prob, data, u);

J(:, 1:data.resp.p_adjt_idx(end), 1:data.resp.p_idx(end)) = ddr;

end
```

```
function [data, J] = uq_int_response_adjoint_evaluation(prob, data, u)

  J = zeros(data.nsamples, data.resp.adjt_size);

  for i=1:data.nsamples
    T = u(data.resp.T_idx(i,1));
    x = u(data.resp.xbp_idx(i,1):data.resp.xbp_idx(i,end));
    p = u(data.resp.p_idx(i,1):data.resp.p_idx(i,end));

    fdata = coco_get_func_data(prob, ...
    coco_get_id(data.sids{i}, 'coll'), 'data');

  [J_T, J_xcn, J_p] = int_resp_adjoint(data, fdata, T, x, p);

  J(i, i) = J_T;
  J(i, data.resp.xcn_idx(i,1):data.resp.xcn_idx(i,2)) = J_xcn;
  J(i, data.resp.p_adjt_idx(i,1):data.resp.p_adjt_idx(i,2)) = J_p;
 end
 J = -1*data.wtd_psi_mat*J;
end

function [J_T, J_xcn, J_p] = int_resp_adjoint(uq_data, fdata, T, x, p)

maps = fdata.coll_seg.maps;
int  = fdata.coll_seg.int;

xx  = reshape(maps.W*x, maps.x_shp);
pp  = repmat(p, maps.p_rep);

wts1 = repmat(int.wt',[maps.NTST,1]);

resp  = uq_data.rhan(xx, pp);
drdx = uq_data.drdxhan(xx, pp);
drdp = uq_data.drdphan(xx, pp);

J_T   = 1/(2*maps.NTST)*resp*wts1;
J_xcn = T/(2*maps.NTST)*drdx(:)';
J_p   = T/(2*maps.NTST)*drdp(:)'*kron(wts1,eye(maps.pdim));

end

function [J_T, J_xcn, J_p] = int_resp_adjoint_nu(uq_data, fdata, T, x, p)
maps = fdata.coll_seg.maps;
mesh = fdata.coll_seg.mesh;

xx = reshape(maps.W*x, maps.x_shp);
pp  = repmat(p, maps.p_rep);

gcn = uq_data.rhan(xx, pp);
gcn = mesh.gka.*gcn;

gdxcn = uq_data.drdxhan(xx, pp);
gdxcn = mesh.gdxka.*gdxcn;

gdp = uq_data.drdphan(xx, pp);
gdp = mesh.gdpka.*gdp;
gdprows = kron(ones(maps.pdim,1), 1:maps.p_rep(2));
gpdcols = kron(ones(maps.p_rep),1:maps.pdim);
gdp = sparse(gdprows, gpdcols, gdp(:));

J_T   = (0.5/maps.NTST)*mesh.gwt*gcn';
J_xcn = (0.5*T/maps.NTST)*gdxcn(:);
J_p   = (0.5*T/maps.NTST)*mesh.gwt*gdp;

end
```

```
function [data, dJ] = uq_int_response_adjoint_Jacobian(prob, data, u)
 dJ = zeros(prod(data.uq.M), data.resp.p_adjt_idx(end), data.resp.p_idx(end));

 for i=data.nsamples:-1:1
  T = u(data.resp.T_idx(i,1));
  x = u(data.resp.xbp_idx(i,1):data.resp.xbp_idx(i,end));
  p = u(data.resp.p_idx(i,1):data.resp.p_idx(i,end));
  fdata = coco_get_adjt_data(prob, ...
    coco_get_id(data.sids{i}, 'coll'), 'data');

  [dJ_T, dJ_xcn, dJ_p] = adj_int_obj_du(data, fdata, T, x, p);

  dJ(i, i, i) = dJ_T(1,1,1);
```

```
    J_T_xbp_idx = 1 + (1:size(fdata.coll_seg.maps.W, 2));
    dJ(i, i, data.resp.xbp_idx(i,1):data.resp.xbp_idx(i,2)) = dJ_T(1,1,J_T_xbp_idx);
    dJ(i, i, data.resp.p_idx(i,1):data.resp.p_idx(i,2)) = dJ_T(1,1,(J_T_xbp_idx(end) + 1):end);

    dJ(i, data.resp.xcn_idx(i,1):data.resp.xcn_idx(i,2), i) = dJ_xcn(1, :, 1);
    dJ(i, data.resp.xcn_idx(i,1):data.resp.xcn_idx(i,2), ...
      data.resp.xbp_idx(i,1):data.resp.xbp_idx(i,2)) = ...
      dJ_xcn(1, :, J_T_xbp_idx);
    dJ(i, data.resp.xcn_idx(i,1):data.resp.xcn_idx(i,2), ...
      data.resp.p_idx(i,1):data.resp.p_idx(i,2)) = ...
      dJ_xcn(1, :, (J_T_xbp_idx(end) + 1):end);
    dJ(i, data.resp.p_adjt_idx(i,1):data.resp.p_adjt_idx(i,2), i) = dJ_p(1, :, 1);
    dJ(i, data.resp.p_adjt_idx(i,1):data.resp.p_adjt_idx(i,2), ...
      data.resp.xbp_idx(i,1):data.resp.xbp_idx(i,2)) = ...
      dJ_p(1, :, J_T_xbp_idx);
    dJ(i, data.resp.p_adjt_idx(i,1):data.resp.p_adjt_idx(i,2), ...
      data.resp.p_idx(i,1):data.resp.p_idx(i,2)) = ...
      dJ_p(1, :, (J_T_xbp_idx(end) + 1):end);
  end
  sz = size(dJ);
  dJ = data.wtd_psi_mat*reshape(dJ, data.nsamples, []);
  dJ = -1*reshape(dJ, [size(data.wtd_psi_mat, 1), sz(2:end)]);
end

function [dJ_T, dJ_xcn, dJ_p] = adj_int_obj_du(uq_data, fdata, T, x, p)


maps = fdata.coll_seg.maps;
int  = fdata.coll_seg.int;

NCOL = int.NCOL;
NTST = maps.NTST;
xdim = int.dim;
pdim = maps.pdim;

wts1 = repmat(int.wt',[maps.NTST,1]);

xx  = reshape(maps.W*x, maps.x_shp);
pp  = repmat(p, maps.p_rep);

dcn  = size(maps.W,1);
dbp  = size(maps.W,2);

ddrdxdx = uq_data.drdxdxhan(xx, pp);
ddrdxdp = uq_data.drdxdphan(xx, pp);
drdx    = uq_data.drdxhan(xx, pp);

J_xcn_T   = 1/(2*maps.NTST)*drdx(:);
J_xcn_xbp = sparse(maps.fdxrows, maps.fdxcols, ddrdxdx(:))*maps.W;
J_xcn_xbp = T/(2*maps.NTST)*J_xcn_xbp;
J_xcn_p = (T/(2*maps.NTST))*sparse(maps.fdprows, maps.fdpcols, ddrdxdp(:));

dJ_xcn = [J_xcn_T, J_xcn_xbp, J_xcn_p];
dJ_xcn = reshape(full(dJ_xcn), [1, size(dJ_xcn,1),size(dJ_xcn,2)]);

dim    = maps.x_shp(1);
xcnnum = maps.x_shp(2);
xcndim = maps.x_shp(1)*maps.x_shp(2);
drdx_rows = repmat(reshape(1:xcnnum, [1 xcnnum]), [dim 1]);
drdx_cols = repmat(1:xcndim, [1 1]);
drdx_2 = sparse(drdx_rows, drdx_cols, drdx(:))*maps.W;
drdp = uq_data.drdphan(xx, pp);

J_T_T   = 0;
J_T_xbp = 1/(2*maps.NTST)*drdx_2'*wts1;
J_T_xbp = J_T_xbp';
J_T_p   = 1/(2*maps.NTST)*drdp(:)'*kron(wts1,eye(maps.pdim));

dJ_T = [J_T_T, J_T_xbp, J_T_p];
dJ_T = reshape(dJ_T, [1, size(dJ_T,1),size(dJ_T,2)]);
```

```

J_p_xcn = permute(ddrdxdp, [1, 3 ,2, 4]);
J_p_xcn = reshape(J_p_xcn, [maps.pdim, prod(maps.x_shp)]);
J_p_xcn = (T/(2*maps.NTST))*J_p_xcn.*repelem(wts1', maps.pdim, maps.x_shp(1));
J_p_xbp = J_p_xcn*maps.W;
J_p_p = uq_data.drdpdphan(xx, pp);
w = kron(wts1, eye(maps.pdim^2));
J_p_p = reshape((T/(2*maps.NTST))*J_p_p(:)'*w, maps.pdim, maps.pdim);

dJ_p = [J_T_p' J_p_xbp J_p_p];
dJ_p = reshape(full(dJ_p), [1, size(dJ_p,1),size(dJ_p,2)]);

end

function [dJ_T, dJ_xcn, dJ_p] = adj_int_obj_du_nu(uq_data, fdata, T, x, p)

fdata = adj_obj_init_data(fdata);
maps = fdata.coll_seg.maps;
int  = fdata.coll_seg.int;

NCOL = int.NCOL;
NTST = maps.NTST;
xdim = int.dim;
pdim = maps.pdim;

wts1 = repmat(int.wt',[maps.NTST,1]);

xx  = reshape(maps.W*x, maps.x_shp);
pp  = repmat(p, maps.p_rep);

dcn  = size(maps.W,1);
dbp  = size(maps.W,2);

ddrdxdx = uq_data.drdxdxhan(xx, pp);
ddrdxdp = uq_data.drdxdphan(xx, pp);
drdx    = uq_data.drdxhan(xx, pp);

J_xcn_T   = 1/(2*maps.NTST)*drdx(:);
J_xcn_xbp = sparse(maps.fdxrows, maps.fdxcols, ddrdxdx(:))*maps.W;
J_xcn_xbp = T/(2*maps.NTST)*J_xcn_xbp;
J_xcn_p = (T/(2*maps.NTST))*sparse(maps.fdprows, maps.fdpcols, ddrdxdp(:));

dJ_xcn = [J_xcn_T, J_xcn_xbp, J_xcn_p];
dJ_xcn = reshape(full(dJ_xcn), [1, size(dJ_xcn,1),size(dJ_xcn,2)]);

dim    = maps.x_shp(1);
xcnnum = maps.x_shp(2);
xcndim = maps.x_shp(1)*maps.x_shp(2);
drdx_rows = repmat(reshape(1:xcnnum, [1 xcnnum]), [dim 1]);
drdx_cols = repmat(1:xcndim, [1 1]);
drdx_2 = sparse(drdx_rows, drdx_cols, drdx(:))*maps.W;
drdp = uq_data.drdphan(xx, pp);

J_T_T   = 0;
J_T_xbp = 1/(2*maps.NTST)*drdx_2'*wts1;
J_T_xbp = J_T_xbp';
J_T_p   = 1/(2*maps.NTST)*drdp(:)'*kron(wts1,eye(maps.pdim));

dJ_T = [J_T_T, J_T_xbp, J_T_p];
dJ_T = reshape(dJ_T, [1, size(dJ_T,1),size(dJ_T,2)]);

J_p_xcn = permute(ddrdxdp, [1, 3 ,2, 4]);
J_p_xcn = reshape(J_p_xcn, [maps.pdim, prod(maps.x_shp)]);
J_p_xcn = (T/(2*maps.NTST))*J_p_xcn.*repelem(wts1', maps.pdim, maps.x_shp(1));
J_p_xbp = J_p_xcn*maps.W;
J_p_p = uq_data.drdpdphan(xx, pp);
w = kron(wts1, eye(maps.pdim^2));
J_p_p = reshape((T/(2*maps.NTST))*J_p_p(:)'*w, maps.pdim, maps.pdim);

dJ_p = [J_T_p' J_p_xbp J_p_p];
dJ_p = reshape(full(dJ_p), [1, size(dJ_p,1),size(dJ_p,2)]);

end

function [data, J] = adj_objhan_du(prob, data, u)

pr   = data.pr;
maps = pr.coll_seg.maps;
mesh = pr.coll_seg.mesh;
opt  = pr.coll_opt;

T = u(maps.T_idx);
x = u(maps.xbp_idx);

xcn = reshape(maps.W*x, maps.x_shp);

gdxdxcn = pr.ghan_dxdx(xcn);
gdxdxcn = mesh.gdxdxka.*gdxdxcn;
gdxdxcn = sparse(opt.gdxdxrows1, opt.gdxdxcols1, gdxdxcn(:))*maps.W;
J       = (0.5*T/maps.NTST)*sparse(opt.gdxdxrows2, opt.gdxdxcols2, ...
  gdxdxcn(opt.gdxdxidx), opt.dJrows, opt.dJcols);

gdxcn   = pr.ghan_dx(xcn);
gdxcn   = mesh.gdxka.*gdxcn;
J       = J + (0.5/maps.NTST)*sparse(opt.gdxdTrows, opt.gdxdTcols, ...
  gdxcn(:), opt.dJrows, opt.dJcols);

gdxcn   = mesh.gwt*sparse(maps.gdxrows, maps.gdxcols, gdxcn(:))*maps.W;
J       = J + (0.5/maps.NTST)*sparse(opt.gdTdxrows, opt.gdTdxcols, ...
  gdxcn(:), opt.dJrows, opt.dJcols);

J       = J + sparse(opt.gdpdprows, opt.gdpdpcols, ones(1,3)/5, ...
  opt.dJrows, opt.dJcols);

end

function data = adj_obj_init_data(fdata)

data.coll_seg  = fdata.coll_seg;
data.ghan      = @ghan;
data.ghan_dx   = @ghan_dx;
data.ghan_dxdx = @ghan_dxdx;

seg  = fdata.coll_seg;
maps = seg.maps;
int  = seg.int;

NCOL = int.NCOL;
NTST = maps.NTST;
xdim = int.dim;
pdim = maps.pdim;

rows = NCOL*NTST*kron(0:(xdim-1), ones(1,xdim));
opt.gdxdxrows1 = repmat(rows, [1 NCOL*NTST]) + ...
  kron(1:NCOL*NTST, ones(1,xdim^2));
cols = reshape(1:xdim*NCOL*NTST, [xdim NCOL*NTST]);
opt.gdxdxcols1 = repmat(cols, [xdim 1]);

step = 1+xdim*(0:NCOL-1);
step = repmat(step(:), [1 xdim]) + repmat(0:xdim-1, [NCOL 1]);
step = repmat(step(:), [1 xdim*(NCOL+1)]) + ...
   (xdim*NCOL*NTST+2+pdim)*repmat(0:xdim*(NCOL+1)-1, [xdim*NCOL 1]);
step = repmat(step(:), [1 NTST]) + ...
   (xdim*NCOL+xdim*(NCOL+1)*(xdim*NCOL*NTST+2+pdim))*...
  repmat(0:NTST-1, [xdim^2*NCOL*(NCOL+1) 1]);
opt.gdxdxcols2 = step(:);
opt.gdxdxrows2 = ones(xdim^2*NCOL*(NCOL+1)*NTST, 1);

step = 1:NCOL;
step = repmat(step(:), [1 xdim]) + NTST*NCOL*repmat(0:xdim-1, [NCOL 1]);
step = repmat(step(:), [1 xdim*(NCOL+1)]) + ...
  xdim*NTST*NCOL*repmat(0:xdim*(NCOL+1)-1, [xdim*NCOL 1]);
step = repmat(step(:), [1 NTST]) + (NCOL+xdim^2*NTST*NCOL*(NCOL+1))*...
  repmat(0:NTST-1, [xdim^2*NCOL*(NCOL+1) 1]);
opt.gdxdxidx = step(:);

opt.gdxdTrows = ones(xdim*NTST*NCOL, 1);
opt.gdxdTcols = (xdim*NCOL*NTST+2+pdim)*xdim*(NCOL+1)*NTST + ...
  (1:xdim*NTST*NCOL)';

opt.gdTdxcols = NTST*NCOL*xdim+2 + ...
  (xdim*NTST*NCOL+2+pdim)*(0:xdim*(NCOL+1)*NTST-1)';
opt.gdTdxrows = ones(xdim*(NCOL+1)*NTST, 1);


opt.gdpdprows = ones(3,1);
opt.gdpdpcols = (NTST*xdim*NCOL+2+pdim)*(xdim*(NCOL+1)*NTST+2) + ...
  xdim*NTST*NCOL+2+[1 NTST*xdim*NCOL+2+pdim+2 2*(NTST*xdim*NCOL+2+pdim)+3]';

opt.dJrows = 1;
opt.dJcols = (xdim*NTST*NCOL+2+pdim)*(xdim*NTST*(NCOL+1)+2+pdim);

data.coll_opt = opt;

data = coco_func_data(data);

end
```