Spectral Methods for Uncertainty Quantification

Christian Brams

Kongens Lyngby 2013
DTU Compute - B.Sc.Eng.- 2013- 3

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253031
compute@compute.dtu.dk
www.compute.dtu.dk DTU Compute - B.Sc.Eng.- 2013- 3

Summary (English)

The goal of the thesis is to apply uncertainty quantification by generalized polynomial chaos with spectral methods on systems of partial differential equations, and implement the methods in the `Python` programming language.

We start off by introducing the mathematical basis for spectral methods for numerical computations. Deriving standard forms of differential operators enable us to implement the spectral collocation method on most partial differential equations. We implement the spectral methods in `Python`, creating a standardized method for solving a numerical problem spectrally using the spectral collocation method.

Afterwards we derive the stochastic collocation and Galerkin methods, allowing us to combine the spectral methods with the generalized polynomial chaos methods in order to achieve exponential convergence in both the numerical solution as well as the quantification of uncertainty.

Using `Python` as the medium, we implement these combined methods on a lid driven cavity problem as well as a two dimensional tank with nonlinear free surface movement, in order to examine the impact uncertainty on the input can have on a system of partial differential equations, and how to efficiently quantify the impact. It is discovered that using uncertainty quantification, we can describe the actual effect of the variables, by looking at what changes when the variable is subject to uncertainty, even for nonlinear systems.

The methods derived in this thesis combine excellently, and are easy to implement on most partial differential equations, allowing great versatility in implementing these methods of uncertainty quantification on different differential systems.

Summary (Danish)

Formålet med denne opgave er at anvende kvantificering af usikkerhed ved metoden *generalized polynomial chaos* sammen med spektrale metoder til løsning af partielle differentialligninger, og at implmenetere dette i programmeringssproget `Python`.

Vi starter med at indtroducere den matematiske basis for spektrale metoder til numeriske beregninger. Ved at udlede standardiserede udtryk for differential operatorer, kan vi implementere spektral kollokations metoden på de fleste partielle differential systemer. Vi implementerer de spektrale metoder i `Python`, og opsætter en standardiseret metode til at løse numeriske problemer med den spektrale kollokations metode.

Herefter udledes stokastisk kollokations og Galerkin metoder, hvilket tillader os at kombinere spektral metoder med stokastise *generalized polynomial chaos* metoder, til at opnå eksponentiel konvergens både på den numeriske løsning og i kvantificeringen af usikkerhed i et system.

Ved at anvende `Python`, implementerer vi disse metoder sammen på et *lid driven cavity* problem og en simuleret to-dimensionel vandtank med ikke-lineær fri overfladebevægelse. Med udgangspunkt i disse to problemstillinger vil vi undersøge betydningen af usikkerhed på input i et system af partielle differentialligninger, og hvordan man effektivt kvantificerer dette. Det viser sig at man kan sige meget om en enkelt konstants indflydelse på et system, ved at analysere hvordan usikkerhed på denne konstant påvirker systemet.

Metoderne der er udledt i denne opgave kan med fordel kombineres og nemt anvendes på det fleste partielle differentialligninger, hvilket giver stor mulighed for anvendelse til kvantificering af usikkerhed på forskellige differentialligningssystemer.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring an B.Sc.Eng. in Mathematics and Technology.

The thesis deals with Spectral Methods, and the application regarding Uncertainty Qualification.

The thesis consists of a study of spectral methods, their application and implementation in the `Python` programming language. It contains a chapter exploring the method of generalized polynomial chaos (GPC) for use with uncertainty quantification. The goal of this thesis is to be able to combine the spectral methods with the GPC methods for uncertainty quantification, in order to create an efficient way to quantify the propagation of uncertainty through partial differential equation models.

In order to fully comprehend this thesis, it is assumed that the reader has basic knowledge of numerical methods for solving differential equations, as well as an understanding of linear algebra and programming on a basic level.

Lyngby, 31-January-2013

Christian Brams

Acknowledgements

I would like to thank my supervisor Allan P. Engsig-Karup and Ph.D. student Daniele Bigoni for their invaluable help during this project. Without their counseling and guidance, I would likely never have finished the project.

Additionally, I would like to thank Charlotte Frausing for having an near inexhaustible reservoir of smiles and help whenever I needed it.

I would like to thank my family for providing understanding and support, as well as relieving me of some of the more menial tasks when the deadline approached.

Kenneth, Anna and Christine deserve a special thanks, for providing a forum with intelligent feedback, when we first endeavored into spectral methods.

Contents

Chapter 1

Introduction

Uncertainty quantification is the concept of characterizing the effects of uncertainty to a useable domain. This allows us to quantify the effects a slight change in the input will have when applied to an advanced mathematical model. This characterization is applicable to most areas of science, as it is infeasible to replicate the exact conditions for the tests which are run.

Through David A. Kopriva's "Implementing Spectral Methods for Partial Differential Equations", [Kop09], this thesis will explore the numerical spectral methods used for solving differential equations. This will allow for rapid convergence in the error for the numerical solutions to differential equations, enabling us to sample data with lesser points.

Implementing these numerical methods is essential to the process, as the vast amounts of calculations needed to be done is insurmountable to anything but computers. For this we will explore the possibilities for implementing these methods in an open and easily acceptable form, through the programming language of `Python`. Using the communally developed packages `SciPy` and `NumPy`, we can emulate the easy and optimized interface of `MATLAB`, without the need for commercial licenses, allowing everyone to utilize the ideas behind these methods.

Using Dongbin Xiu's "Numerical Methods for Stochastic Computations", uncertainty quantification will be introduced, with the main focus on generalized polynomial chaos, a method with very fast convergence in the errors of quantifying uncertainty, akin to the spectral methods described in Kopriva's book. Generalized polynomial chaos is currently "one of the most widely adopted methods, and in many cases the only feasible method, for stochastic simulations of complex systems" as Xiu explains in his preface.

Combining the spectral methods with the generalized polynomial chaos methods, we will create accurate methods for solving partial differential equations, while quantifying the uncertainty propagated through the models, by uncertainty on the boundary or initial conditions. This will allow us to effectively examine the impact of uncertainty on complex differential models.

CHAPTER 2

Problem statement

The main goal of the thesis is to

1. Be able to describe and understand relevant theory, from spectral methods to uncertainty quantification.
2. Develop routines and spectral solvers in `Python`.
3. Exemplify uncertainty quantification techniques through application with different equation solvers.
4. Formulate and express a clear set of hypotheses and project aims.
5. Conceive, design and execute appropriate experiments, analytical and/or modeling methods.
6. Communicate knowledge through well written, well presented, concise, clear and well structured reports and oral presentations. Present project results using clear tables and figures.
7. Understand the interaction between the different components of a technological issue.

Chapter 3

Spectral methods as numerical methods

Spectral methods are numerical methods designed for solving ordinary differential equations (ODEs) or partial differential equations (PDEs) as well as other related problems. The basic idea behind spectral methods can be compared to the finite element methods, where the solution is found as a function of the basis functions representing the spectrum. The key difference is that for finite element methods, the basis functions are zero on large parts of the domain, while for spectral methods the basis functions are typically nonzero. This gives the functions excellent error properties for smooth functions, as the error is minimized across the spectrum, allowing exponential convergence.

3.1 Orthogonal polynomials

Orthogonal polynomials are the basis of the spectral methods. The orthogonality property allow us to find a unique set of coefficients to describe our function, and it is central to the concept of spectral methods. The idea of the spectral methods lie in approximating the function using a finite sum of orthogonal polynomials.

Any function $\varphi(x,t)$ can be represented as a sum of a unique set of coefficients paired with the appropriate orthogonal basis from the family of basis functions $\{\Phi_n(x)\}_{n=0}^{\infty}$ such that

$$\varphi(x,t) = \sum_{k=0}^{\infty} \hat{\varphi}_k(t)\Phi_k(x)$$

This assumes that the basis functions are orthogonal on an interval $[a,b]$, with respect to a weight function w, such that

$$(\Phi_n, \Phi_m)_w = \int_a^b \Phi_n(x)\Phi_m^*(x)w(x)\,\mathrm{d}x = C_n\delta_{nm} \qquad \delta_{nm}\begin{cases}1, & n=m\\ 0, & n\neq m\end{cases} \tag{3.1}$$

Another central aspect of the polynomials we will be using, will be that they have an associated easy to evaluate quadrature, which is used to approximate the integral of the functions.

$$Q[f] = \sum_{j=0}^{N} f(x_j)w_j = \int_a^b f(x)\,\mathrm{d}x + E$$

3.1.1 Generating the polynomials

The main classes of polynomials we will be using are the *Lagrange* polynomials, for periodic functions using Fourier interpolation, and the *Legendre* polynomials, for use with non-periodic functions.

3.1.1.1 Periodic functions

For periodic functions, we will be using a Fourier series to approximate our functions, since these will have an inherent periodicity – allowing us to exploit this to automatically ensure periodicity. In essence, any function can be approximated by a Fourier series, but since we want to truncate the function and not include the infinite sum, we will only use the Fourier basis for periodic functions. The function F is approximated by the infinite sum

$$f(x) = \sum_{k=-\infty}^{\infty} \hat{f}_k e^{ikx}$$

Since the complex exponentials – the Fourier basis functions – are orthogonal, (3.1) makes it easy to calculate the Fourier coefficients $\hat{f}_k$.

$$\left(F_N, e^{inx}\right) = \left(\sum_{k=-\infty}^{\infty} \hat{f}_k e^{ikx}, e^{inx}\right) = \sum_{k=-\infty}^{\infty} \hat{f}_k \left(e^{ikx}, e^{inx}\right) = C_n \hat{f}_n$$

Where the weights can be calculated to $w = 2\pi$, since the basis functions are 2π-periodic

$$C_n = \left(e^{inx}, e^{inx}\right) = \int_0^{2\pi} e^{i(n-n)x}\,\mathrm{d}x = 2\pi$$

This relies on infinite sums, and to be able to compute this, we need to truncate the function, letting us have a truncated function $P_n f(x)$

$$P_N f(x) = \sum_{k=-N/2}^{N/2} \hat{f}_k e^{ikx}$$

The error between $P_N f$ and f is shown in [Kop09, eq. 1.30] to be directly related to the size of the remaining coefficients. This allows the spectral methods to obtain exponential convergence for functions where the coefficients decrease exponentially – functions periodic on $[0, 2\pi]$ with all derivatives continuous.

A special case of this exists, called $I_N f$ the Fourier interpolant, where we compute the coefficients so they fulfill the following

$$I_N f(x_n) = f(x_n), \quad n = 0, \ldots, N-1 \wedge x_n \frac{2\pi n}{N}$$

Where the last point is not needed, since $I_N f(0) = I_N f(2\pi)$. The coefficients and approximation for this expansion is defined in [Kop09, pg. 15] as

$$\tilde{f}_k = \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-ikx_j} \qquad I_N f(x) = \sum_{k=-N/2}^{N/2} \frac{1}{c_k} \tilde{f}_k e^{ikx}$$

$$c_k = \begin{cases} 1, & k = -N/2+1, \ldots, N/2-1 \\ 2, & k = \pm N/2 \end{cases}$$

Which can be rewritten as *Lagrange* form

$$f(x) \approx \sum_{k=-N/2}^{N/2} \frac{1}{c_k} \left(\frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-ikx_j} \right) e^{ikx} \Leftrightarrow$$

$$f(x) \approx \sum_{k=-N/2}^{N/2} \sum_{j=0}^{N-1} \frac{1}{c_k} \frac{1}{N} f(x_j) e^{-ikx_j} e^{ikx} \Leftrightarrow$$

$$f(x) \approx \sum_{j=0}^{N-1} \sum_{k=-N/2}^{N/2} \frac{1}{c_k} \frac{1}{N} f(x_j) e^{-ikx_j} e^{ikx} \Leftrightarrow$$

$$f(x) \approx \sum_{j=0}^{N-1} f(x_j) \sum_{k=-N/2}^{N/2} \frac{1}{c_k} \frac{1}{N} e^{-ikx_j} e^{ikx}$$

We can then use the trigonometric sum formula for reducing the sum in $h_j(x)$ to a closed form expression, which is can be evaluated easily in `Python`. The trigonometric sum formula says

$$\sum_{k=-K}^{K} e^{iks} = \frac{\sin\left(\left(K + \frac{1}{2}\right)s\right)}{\sin\left(\frac{1}{2}s\right)}$$

Which we can use to rewrite $h_j(x)$

$$h_j(x) = \frac{1}{N} \sum_{k=-N/2}^{N/2} e^{ik(x-x_j)} \Leftrightarrow h_j(x) = \frac{1}{N} \frac{\sin\left(\frac{N+1}{2}(x - x_j)\right)}{\sin\left(\frac{1}{2}(x - x_j)\right)}$$

We can use **Lemma A.1** (See appendix B) to show that

$$h_j(x) = \frac{1}{N} \frac{\sin\left(\frac{N+1}{2}(x - x_j)\right)}{\sin\left(\frac{1}{2}(x - x_j)\right)} = \frac{1}{N} \sin\left(\frac{N}{2}(x - x_j)\right) \cot\left(\frac{1}{2}(x - x_j)\right)$$

This allows us to calculate the interpolating polynomial for all points, and verify that these polynomials are mostly non-zero over most of the domain, as shown in figure 3.1

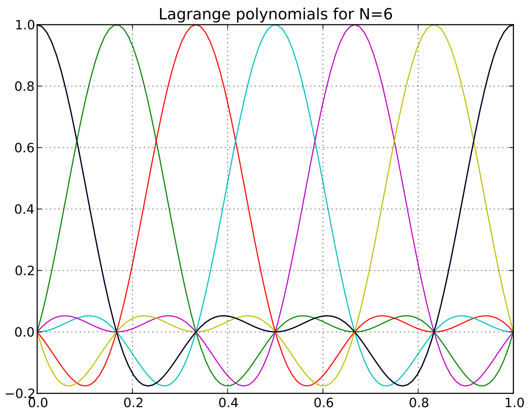


Figure 3.1: The Lagrange polynomials for $N = 6$.

For calculating the integrals for the periodic functions, we will be using a composite trapezoidal rule, as [Kop09, pg.13] shows this to be approximating the integral exactly, giving us the quadrature rule

$$Q_F[f] = \frac{2\pi}{N} \sum_{j=0}^{N-1} f(x_j), \quad x_j = \frac{2j\pi}{N}$$

3.1.1.2 Non-periodic functions

For non-periodic functions, we will be using Jacobi polynomials to model our functions. The Jacobi polynomials which is a family of polynomials that is defined by 3 variables, α, β and n. The variables α and β define which type of polynomial, and n is the order. The polynomial is then defined recursively

$$\begin{aligned}
P_0^{(\alpha,\beta)}(x) &= 1 \\
P_1^{(\alpha,\beta)}(x) &= \frac{1}{2}(\alpha - \beta + (\alpha + \beta + 2)x) \\
a_{n+1,n}^{(\alpha,\beta)} P_{n+1}^{(\alpha,\beta)}(x) &= \left(a_{n,n}^{(\alpha,\beta)} + x\right) P_n^{(\alpha,\beta)}(x) - \left(a_{n-1,n}^{(\alpha,\beta)} + x\right) P_{n-1}^{(\alpha,\beta)}(x)
\end{aligned}$$

Where

$$\left(a_{n-1,n}^{(\alpha,\beta)} + x\right) = \frac{2(n+\alpha)(n+\beta)}{(2n+\alpha+\beta+1)(2n+\alpha+\beta)}$$
$$\left(a_{n,n}^{(\alpha,\beta)} + x\right) = \frac{\alpha^2-\beta^2}{(2n+\alpha+\beta+2)(2n+\alpha+\beta)}$$
$$\left(a_{n+1,n}^{(\alpha,\beta)} + x\right) = \frac{2(n+1)(n+\alpha+\beta+1)}{(2n+\alpha+\beta+2)(2n+\alpha+\beta+1)}$$
$$\left(a_{-1,0}^{(\alpha,\beta)} + x\right) = 0$$

The Jacobi polynomials form a basis for $[-1,1]$, and [Kop09, pg. 26] shows that we can represent any square-integrable function f as an infinite series

$$f(x) = \sum_{n=0}^{\infty} \hat{f}_k P_k^{\alpha,\beta}(x) \qquad \hat{f}_k = \frac{\left(f, P_k^{\alpha,\beta}\right)_w}{\|P_k^{\alpha,\beta}\|_w^2}$$

This allows us to calculate the *Legendre* polynomials, which are defined as $P_n^{0,0}$, and are the normalized versions are shown in figure 3.2

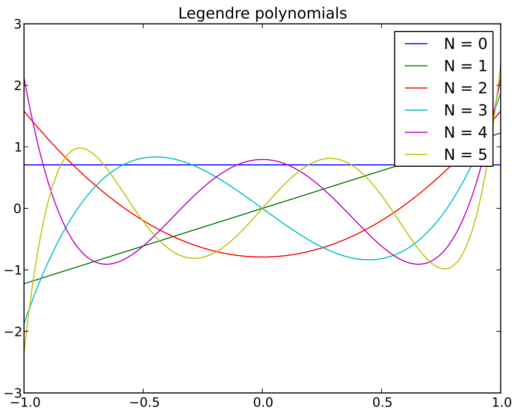


Figure 3.2: The first six Legendre polynomials.

For the Legendre polynomials, we will be using a Legendre-Gauss quadrature, or the Legendre-Gauss-Lobatto rules, depending on wether or not we will be including the boundary points. This uses a simple system, where

$$Q_J[f] = \sum_{j=0}^{N} f(x_j) w_j$$

When appropriate x_j and w_j are chosen.

For the Legendre-Gauss quadrature, where the end points are not included, are defined in [Kop09, eq. 1.127] as (3.2). For the Legendre-Gauss-Lobatto case, the points and weights are defined by (3.3)

$$x_j = \text{ zeros of } L_{N+1}(x) \qquad w_j = \frac{2}{\left(1 - x_j^2\right)\left[L'_{N+1}\right]^2} \tag{3.2}$$

$$x_j = +1, -1, \text{ zeros of } L'_N(x) \qquad w_j = \frac{2}{N(N+1)} \frac{1}{\left[L_N(x_j)\right]^2} \tag{3.3}$$

We will be using the Legendre-Gauss-Lobatto nodes exclusively, since these will allow us points on the boundary which we will need for enforcing boundary conditions.

3.1.2 Differentiating the polynomials

For the nodal interpolating polynomials, we can devise a matrix that can differentiate the nodal values when the matrix product is calculated. This will of course require a tailored matrix to the problem, but when the problem is scaled to the spectrum of the basis functions, and the points are chosen according to the relevant quadrature rule, we can generate a fixed matrix for all systems using the same number of points.

3.1.2.1 Periodic functions

For periodic functions, we recall from section 3.1.1.1 that we can portray them as

$$I_N f(x) = \sum_{j=0}^{N-1} f(x_j) h_j(x)$$

Which means that we can calculate the differentiated value as

$$\frac{d}{dx} \mathcal{I}_N f(x) = \sum_{j=0}^{N-1} f(x_j) h'_j(x) \tag{3.4}$$

We therefore calculate the differentiate of $h_j(x)$. Since we can write $h_j(x)$ as

$$h_j(x) = \frac{1}{N} \sin\left(\frac{N}{2}(x - x_j)\right) \cot\left(\frac{1}{2}(x - x_j)\right)$$

We can differentiate it using `Maple`. If we need to use this on a discrete set of points with equal spacing, we can calculate an expression that is easier to comprehend, since we will be using the indices instead of references to the exact point. Our interval is $x \in [0; 2\pi]$ which is parted in N parts, x_j will be defined as $x_j = \frac{2\pi}{N}j$ where $j = 1, 2, \cdots, N-1$, and we can substitute x with a discrete point as well, giving us $x = \frac{2\pi}{N}k$ where $k = 1, 2, \cdots, N-1$. With this, `Maple` evaluates

$$h_j'(x_k) = \frac{1}{2}(-1)^{1+k+j}\cot\left(\frac{\pi(j-k)}{N}\right)$$

Now, since we already now how to calculate the derivative of the function from (3.4), we can simply multiply design a matrix D of $h_j'(x_k)$ such that we can calculate a vector product rather than a sum. Since $h_j'(x_k)$ is defined in a way, where $h_j'(x_j)$ does not exist, we will have to set this manually to zero. This gives us the following matrix

$$D_{jk} = \begin{cases} \frac{1}{2}(-1)^{1+k+j}\cot\left(\frac{\pi(j-k)}{N}\right) & , j \neq k \\ 0 & , j = k \end{cases} \tag{3.5}$$

This allows us to easily calculate the differential by $\mathbf{f_N'} = \mathbf{Df_N}$, as long as $\mathbf{f_N}$ are the coefficients of the interpolant polynomial – which are identical to the function value in the nodal points.

3.1.2.2 Non-periodic functions

For non-periodic function, we will need to define the Vandermonde matrix $\mathcal{V}$, which is is a matrix characterized by being the matrix that can couple the nodal function values $\tilde{f}$ with the modal function values $\hat{f}$ in the relationship

$$\tilde{f} = \mathcal{V}\hat{f} \qquad\qquad \mathcal{V}_{ij} = \Phi_j(x_i) \tag{3.6}$$

Where $\Phi_j(x)$ is the jth basis function, being $P_j^{(0,0)}$ in our case. The Vandermonde matrix can be used to construct a first order differentiation matrix. This is due to the representation of the differentiation in a nodal expansion

$$\frac{d}{dx}\left(\sum_{j=0}^{N} f_i h_j(x)\right) = \sum_{j=0}^{N} f_i \frac{d}{dx} h_j(x)$$

And the modal expansion

$$\frac{d}{dx}\left(\sum_{j=0}^{N} \hat{f}_i \Phi_j(x)\right) = \sum_{j=0}^{N} \hat{f}_i \frac{d}{dx} \Phi_j(x)$$

If we construct a matrix with the derivatives of h called D and a matrix with the derivatives of Φ called $\mathcal{V}_x$, we can construct the following relation.

$$\frac{d\mathbf{f}}{dx} = D\mathbf{f} = D\mathcal{V}\hat{\mathbf{f}} = \mathcal{V}_x\hat{\mathbf{f}}$$

From where we can isolate D to get a matrix that can calculate the derivates akin to the method we used for periodic functions.

$$D = \mathcal{V}_x\mathcal{V}^{-1}$$

This requires knowledge of the first derivative of our Legendre polynomials, in order to calculate $\mathcal{V}_x$, and we use the definition of the first derivative of the Jacobi Polynomials, as described in [EK11a, Slide 20]

$$\frac{d}{dx}\tilde{P}_n^{(\alpha,\beta)}(x) = \sqrt{n(n+\alpha+\beta+1)}\tilde{P}_{n-1}^{(\alpha+1,\beta+1)}(x)$$

This allows us to easily differentiate values of the interpolant polynomial for non-periodic functions as well. The transformation between nodal and modal coefficients from (3.6), will be useful later, since we can use this to transform values from one nodal set to another nodal set, using two Vandermonde matrices.

3.1.3 Challenges for discrete modeling

The discrete modeling we will be using will create a couple of problems, since we are truncating the infinite sums.

3.1.3.1 Gibbs phenomenon

As mentioned in section 3.1.1, the error will decrease in a speed determined by the speed that the coefficients decrease. *Gibbs phenomenon* is a problem that arises when we use only smooth functions to approximate a not-smooth function. When the approximated function is not smooth, the coefficients will never go towards zero, as it will need an infinite amount of basis functions to approximate the discontinuity. We will explore this with the function

$$f(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

Since this function is not periodic, we will approximate it using the Legendre polynomials.

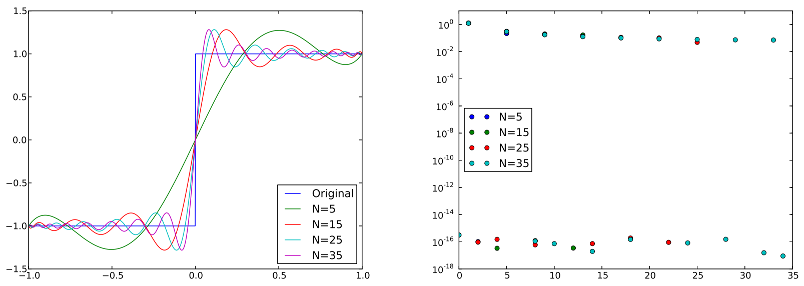


Figure 3.3: An illustration of Gibbs phenomenon by approximating a non-smooth function using only smooth functions. The approximations are shown to the left, the coefficients are shown to the right.

Figure 3.3 shows that despite the increase in resolution, we maintain an error of roughly the same size, it simply moves closer to the discontinuity. We also see that the coefficients assume a close to constant value for rising n, which explains that the error will not decrease either.

3.1.3.2 Aliasing

Aliasing is another problem with discrete sampling, where a set of values can correspond to a multiple basis functions. We will use a Fourier interpolation to illustrate this, as the two complex exponentials $f_1(x) = e^{i2x}$ and $f_2(x) = e^{-i6x}$ are sampled as the same function for a grid where $N = 8$. [Kop09, pg. 20] shows that a sinusoid with wavenumber k and points $x_j = \frac{2\pi j}{N}$ will evaluate to the same values as a sinusoid with wavenumber $k + nN$ where $n \in \mathbb{N}$. Figure 3.4 displays this problem, where the dots mark the sampled values, and f_1 and f_2 are illustrated as patched lines.

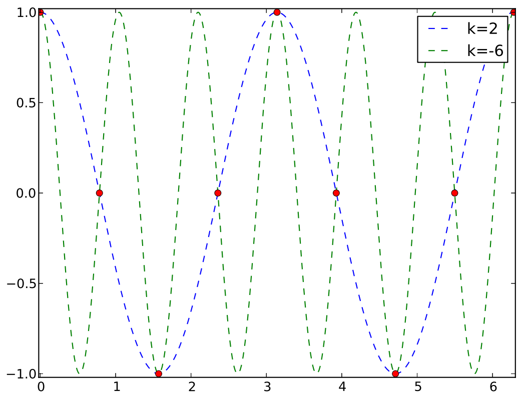


Figure 3.4: On a grid with $N = 8$ points, the functions $f_1(x) = e^{i2x}$ and $f_2(x) = e^{-i6x}$ are illustrated.

3.2 Constructing spectral solvers

On the basis of the orthogonal polynomials, we will be constructing spectral solvers for differential equations. We will be introducing two spectral methods, the *collocation* method and the *Galerkin* method. We will not be implementing the *Galerkin* method, but briefly explain what it is, to indicate the different ways to implement spectral methods.

We will look at both the solution for boundary value problems, as well as the solution for initial value problems.

3.2.1 Spectral methods

We will explain both the collocation method and the Galerkin method, and while the methods differ in how they approximate the solution as well as ease of implementation, they are both spectral methods, and can be used to achieve spectral convergence for a differential problem.

3.2.1.1 The collocation method

The collocation method is based on the idea of *collocation points.* The basic gist of this method is that we will aim to find an interpolating polynomial $I_N f$ of a function f, and we then want to enforce the relationship (3.7) at the collocation points.

$$I_N f(x_j) - f(x_j) = 0 \tag{3.7}$$

This allows us to fulfill the differential equation at each of the collocation points, giving us N independent differential equations. Because of (3.7), we can completely eliminate the need to convert coefficients for the actual implementation, since we will be using the nodal coefficients, which are identical to the function values. This allows us to simple use the matrices derived in section 3.1.2 to generate the values for the differentials in the differential equation.

The collocation method is very easy to implement, as we will show below, since it relies only on the differential matrices, which can be calculated beforehand – but it is also sensitive to aliasing errors, since it only concerns itself with the solution at the collocation points. We have to implement the boundary conditions as specific alterations to the operators, should we not be using the Fourier collocation methods – where the boundaries are fulfilled by the basis functions.

For at problem defined as

$$u_x + u_y = 0$$

We simply construct a linear operator $\mathcal{L} = D_x + D_y$, where the differential-operators are generated from the methods found in section 3.1.2, and we can solve the problem by solving the linear equation

$$\mathcal{L}\mathbf{u} = \mathbf{0}$$

3.2.1.2 The Galerkin method

The Galerkin method takes a different approach than the collocation method. We will seek a solution that fulfills the differential equation in basis space only, which means it will be using the weak form – fulfilling the differential equation only as projected on the basis functions. For the problem

$$u_x + u_y = f(x)$$

The Galerkin method will find the approximated solution v, which simply fulfills

$$(v_x, \phi) + (v_y, \phi) = (f(x), \phi)$$

Where ϕ are the basis functions and $(\bullet_1, \bullet_2)$ is the inner product, or projection of $\bullet_1$ unto $\bullet_2$.

The orthogonality of the basis functions gives us $N + 1$ equations – one for each basis function including $n = 0$ – and knowledge of the form the basis functions take will make us able to calculate an ordinary ODE for each of the $N + 1$ equations. This allows us to calculate directly using the modal coefficients, transforming into modal coefficients before we start calculations, and transforming the coefficients to the solution after the calculations.

Since the Galerkin method operates in the modal basis, aliasing errors will not be a problem. The Galerkin method requires us to choose basis functions that fulfill the boundary conditions, and calculating the the derivatives of these according to the differential equation. This makes it more cumbersome to implement in general, since it will need to be tailored to each problem individually.

3.2.2 Types of differential equation problems

We have two main categories of partial differential equations, which we will be solving using our spectral methods, the initial value problems (IVPs) and the boundary value problems (BVPs).

3.2.2.1 Boundary Value Problems

Boundary values are a classic case of differential equations, specifically PDEs. They are characterized – as the name suggests – by supplying the values on the boundaries, and the differential equation then dictates how the interior of the solution is related.

The primary method we will be using for these solutions, will be to create a linear system of equations, where the solutions are the unknowns in the equation. We will be able to approximate the derivative of the function with the differential operators, allowing us to isolate the function values in a linear system. The collocation approximation will generally lead to a dense system since their differential operators are dense, while an application of the Galerkin method might yield a sparse system.

3.2.2.2 Initial Value Problems

Initial value problems are another set of differential equation problems, where we will be generating a solution from the initial value. This is typically time-related problems, where we will be generating the time-derivative from the solution at the prior time-step and its derivatives. We will be employing the same strategy as in the BVPs, where we will calculate the derivative values directly from the function values. This allows us to generate a set of time derivatives, which will be used to solve a standard ODE problem.

Because the spectrum for the time-dimension is not exactly defined, we will not be able to properly utilize the whole spectrum for our calculations, limiting our spectral methods to work on the remaining dimensions. This makes the method used for solution a *pseudo-spectral* method, as it will be bounded by the accuracy of the time-stepping integrator and not by the spectral accuracy of our methods. It will allow us to make N ordinary ODEs for time-stepping, which will be spectrally accurate in between time-steps.

3.3 Solving differential equations

We will employ the discussed methods on a selection of problems, where we will solve a boundary-value problem and an initial-value problem in the form of Burgers' equation, which we will use for uncertainty quantification later as well.

3.3.1 Using the spectral collocation method

For this problem, we will be solving the differential equation

$$-\epsilon \frac{d^2}{dx^2} u - \frac{d}{dx} u = 1 \qquad\qquad u(0) = u(1) = 0 \tag{3.8}$$

Where $\epsilon = 0.01$. The solution to this problem is found in `Maple` as

$$u(x) = \frac{e^{(1-x)/\epsilon} + e^{1/\epsilon}(x-1) - x}{1 - e^{1/\epsilon}} \tag{3.9}$$

We will use a Legendre basis for this calculations, which means that we will be using points defined from the Legendre-Gauss-Lobatto quadrature defined in (3.3). These nodes are defined for $x_{GL} \in [-1, 1]$, and we will need to scale them to our domain, which is $x \in [0,1]$, which means that we will apply the transformation

$$x = \frac{x_{GL}+1}{2} \Leftrightarrow x_{GL} = 2x - 1$$

This transformation gives us a scaling factor, which we can calculate as

$$\frac{dx_{GL}}{dx} = 2$$

Using the spectral collocation method, defined in (3.7), we will seek to satisfy the solution u_N the following problem

$$-\epsilon \frac{d^2 u_N}{dx^2}(x_i) - \frac{du_N}{dx}(x_i) = 1 \qquad 1 \leq i \leq N - 1 u_N(x_i) = 0 \qquad i = 0, N$$

Using the differential operator from section 3.1.2.2, we can approximate $\frac{du_N(\mathbf{x})}{dx} = 2D \cdot \mathbf{u}$ and $\frac{d^2 u_N(\mathbf{x})}{dx^2} = 4D^2 \cdot \mathbf{u}$. This allows us to write the problem as

$$-\epsilon 4D^2\mathbf{x} - 2D\mathbf{x} = 1 \Leftrightarrow \big(-\epsilon D^2 - D\big)\mathbf{x} = 1$$

By creating the linear operator $\mathcal{L} = -\epsilon 4D^2 - 2D$, we can impose boundary conditions simply by modifying the first and last rows of $\mathcal{L}$ such that it corresponds to $1 \cdot x_0 + 0 \cdot x_n = 0$ and $1 \cdot x_N + 0 \cdot x_m = 0$ where $n \in [1, N]$ and $m \in [0, N-1]$. The right-hand-side vector b needs to be changed as well, such that $b_i = 1$ for $1 \leq i \leq N-1$ and $b_i = 0$ otherwise.

This allows us to solve the problem defined in (3.8) spectrally by solving the linear system $\mathcal{L}\mathbf{u} = 1$. The implementation of this problem will be handled in section 4.4.1.

3.3.2 Burgers' equation

The viscous Burgers' equation is a partial differential equation of the form

$$\begin{aligned} &u_t + uu_x = vu_{xx} && x \in [-1,1] \\ &u(-1,t) = 1 && u(1,t) = -1 \quad, \forall t > 0 \\ &v > 0 \end{aligned} \tag{3.10}$$

This equation will assume a steady state after t has become high enough, assuming it has an initial condition that satisfies the boundaries. We will be using the function $-\tanh(x)\frac{1}{|-\tanh(-1)|}$, which will be normalized for the endpoints to 1 and -1.

In order to derive the time-step, we will be using the collocation method as described in section 3.3.1 to approximate the solution, which gives us the solution approximated with the Legendre-Gauss-Lobatto nodes x_i

$$\begin{aligned} &\frac{du_N}{dt}(x_i,t) + u_N(x_i,t)\frac{du_N}{dx}(x_i,t) = v\frac{d^2u_N}{dx^2}(x_i,t) && 1 \leq i \leq N-1 \\ &u(x_0,t) = 1 && u(x_N,t) = -1 \end{aligned}$$

Since we are using the collocation points x_i, we can generate the differential operator D from section 3.1.2.2 and approximate the differentials, giving us.

$$\begin{aligned} \frac{d\mathbf{u_N}(t)}{dt} &+ \mathbf{u_N}(t)(D\dot{}\mathbf{u_N}(t)) = v\big(D^2 \cdot \mathbf{u_N}(t)\big) \Leftrightarrow \\ \frac{d\mathbf{u_N}(t)}{dt} &= v\big(D^2 \cdot \mathbf{u_N}(t)\big) - \mathbf{u_N}(t)(D\dot{}\mathbf{u_N}(t)) \end{aligned} \tag{3.11}$$

Since we have an established initial condition, we will simply need to ensure that $\frac{du_{(0)}}{dt} = 0$ and $\frac{du_{(N)}}{dt} = 0$, and we can solve this as a coupled ordinary differential equation, using this pseudo-spectral method.

The implementation of Burgers' equation will be handled in section 4.4.2.

Chapter 4

Implementation of spectral methods

The implementation of the methods is critical to the methods. If the implementation is wrong, we will not be able to achieve the spectral convergence we desire. It is important that the method of implementation is chosen wisely, such that it will be able to support the computations accurately. We will be using [Big12] for many of the polynomial computations we will be doing, as this package includes functions to evaluate most polynomials apart from the Fourier.

`Python` has been chosen as the programming language due to the portability compared to the licensed `MATLAB` – `Python` can run on most platforms, and it is free to download and install. `Python` also supports a wide array of libraries that mimic the functionality of `MATLAB`, allowing us to use `Python` with relative ease.

4.1 How to use `Python` for numerical computations

Using `Python` as a programming language is not that different from using `MATLAB` – the code is easy to read, write and understand. `Python` is unlike a lot of other programming languages in that it does not use parentheses to indicate nesting of functions or conditions, but rather uses indentation – as most other languages use only as a standard. This gives `Python` code a guarantee that it will be easy to read, as the standardization of the code formulation is inherent in the way it works.

`Python` is – just like `MATLAB` – a high-level interpreted programming language, allowing the user to focus on the coding aspects of the code, and leave handling memory-management, variable types and other machine-dependent things to the `Python` interpreter. This also gives it the quality for effective debugging, as you literally step your way through the code, seeing the results as you go, which can be invaluable when trying to discover a calculation error in an implemented method.

In addition to this, `Python` is non-commercial and cross platform, allowing `Python` scripts to be run on any machine without a license. This, and the fact that it is designed as a general purpose language, allows for a very versatile implementation, that can be improved and run anywhere, by anyone. Where `MATLAB` requires a commercial license to use or even run already compiled code, `Python` is free. `Python` also allows for easy interfacing with most other languages, specifically C and C++, further broadening the spectrum of use with `Python`.

4.1.1 A comparison to `MATLAB`

Most `MATLAB` code can be almost cleanly translated to `Python` code, thanks to the `Python` packages `NumPy`, `SciPy` and `Matplotlib`. There are however some clear differences that must be accounted for.

- **Zero index** – `Python` uses a zero-indexing standard, while `MATLAB` uses a one-indexing standard, making most algorithms give a one-off error if directly converted. This is quickly fixed though, but requires some application of thought for algorithms where the index is part of the mathematical calculations.

- **Basic array actions** – `Python` is designed as a general purpose language and does not automatically assume that we are working in matrices. This gives most common operators an element-wise property, with specialized functions for matrix products and other matrix operations. While `Python` contains classes for simulating the matrix class in `MATLAB`, it is not broadly employed as it is limited to 2-dimensional matrices.

- **Differencing between arrays and functions** – Python does not allow accessing arrays by the `a(1)` standard and instead uses `a[1]`. This can cause some confusion as to why a function will not run but can be quickly eliminated, as it casts an error. Functions are still called with soft parentheses.

- **Implementing sub-functions** – Unlike MATLAB, a Python file can contain several sub-functions that can be accessed from another file, and it is not restricted to the same name as the function either. This allows for a cleaner code-hierarchy.

- **Slicing arrays** – A Python slice of an array is issued with the `:` operator like MATLAB, but unlike MATLAB, beginning and end are assumed unless stated otherwise. This allows the formulation `a[3:]` if you want the array from the fourth element onwards. For accessing the elements in the last part of an array, negative numbers are used, and `a[-1]` will net the last value of an array. Unlike MATLAB, slice operations do not copy the array, but simply provides a view into it. This can cause problems with iterative algorithm that assumes the array is constant – Python uses the `copy` function to copy arrays, where MATLAB copies them as default.

4.1.2 Employed Python packages

Python allows for a vast number of different capabilities, with a wide-ranging package-portfolio. We will concentrate on the five packages that we will be using for our spectral methods and uncertainty quantification.

4.1.2.1 NumPy — Numeric calculations in Python

NumPy is a function library that supports large multi-dimensional arrays, as well as including modified versions of standard mathematical functions, so they can be employed upon these large arrays. It draws heavy inspiration from MATLAB, with most of the functions being called the exact same thing as in MATLAB, like `linspace` and `ones`. It also supplies functions for matrix multiplication and linear solving through the `linalg` module. It will most commonly be imported simply as `np` in this code, allowing calling of functions by `np.linspace`.

A complete description and list of features can be found on `http://docs.scipy.org/doc/numpy/contents.html`

4.1.2.2 `SciPy` — Scientific methods for `Python`

SciPy is a function library that employs a lot of scientific functions, such as fast Fourier transform, ODE integrators, data input and output, statistical functions and much more. It is intimately tied to NumPy, as many of these functions use the NumPy array as standard input and output type.

A complete list of features and can be found on `http://docs.scipy.org/doc/scipy/reference/`

4.1.2.3 `Spyder` — A `MATLAB` imitating development environment

Spyder imitates the development environment provided in `MATLAB`, to a lot of features. It supplies easy step-by-step debugging, variable editing while running, and multiple `Python` consoles. The layout is easily recognizable as the `MATLAB` interface, and even supports an automatic object inspector, which provides the `help` documentation for any functions imported properly.

The homepage of the Spyder development interface is found at `http://code.google.com/p/spyderlib/`

4.1.2.4 `Matplotlib` — The `Python` plotting tool

Matplotlib supplies many visualization options to `Python`, but most notable is the `pyplot` module, which mimics `MATLAB`s plotting features. The pyplot module contains functions nearly impersonating `MATLAB` functions, where the names of the functions are called the same as in `MATLAB`, allowing for very easy conversion between these two development interfaces. It only supports 2D plotting, needing support in order to achieve 3D plotting, and is also limited by the fact that a plot needs to be closed before data can be manipulated again. These are minor annoyances, and we will be using this library to visualize most of our functions. We will mostly be importing this library as `plt`, allowing us to plot with the command `plt.plot(x,y)`

Matplotlib is a comprehensive library, with all the documentation found at `http://matplotlib.org/`

4.1.2.5 DABISpectral1D

DABISpectral1D is a module developed by Daniele Bigoni, PhD student at DTU Compute. It is used for generating the needed values for orthogonal polynomials used in spectral methods. We will be using this to calculate most transformations and quadratures with the Legendre polynomials as well as the Hermite polynomials needed later. It works by first initializing a polynomial as a variable, and this polynomial will then contain the functions needed for quadratures and transformations as sub-functions. We will typically be importing it as `DB`, allowing us to call functions as `DB.Poly1D()`

4.2 Challenges using Python instead of MATLAB

Using `Python` instead of `MATLAB` does prove somewhat difficult regarding some pitfalls one may encounter. We will document here how they might affect the process, and how to overcome them.

4.2.1 Integer division

`Python` is not used exclusively as a numeric language, and because of this it does not automatically cast integer division as a floating point number, should the division not have an integer solution. `Python` automatically casts integer division to another integer, rounding the result down. This can prove problematic if any of the mathematical expressions include a fraction like $\frac{1}{2}$. It can be easily overcome though, with two easy solutions.

- Adding a punctuation mark after the integer, to indicate it should be treated as a floating point value. In the above example we would then write `1./2.` instead of `1/2`.
- Importing the built-in function to convert integer division to floating point numbers automatically. This will require the very first line of code in the relevant file to be "`from __future__ import division`".

Both of these fixes are very easy to implement, although it can easily ruin a code if not remembered, as it will not throw any error, despite reducing the product it is a part of to zero. We will mainly be employing the second method to ensure that we will not have any fractions we have forgotten.

4.2.2 Interfacing with `MATLAB`

One of the biggest challenges is interfacing with the many `MATLAB` scripts which are available. Many results are approximated using `MATLAB` scripts, which will be useful for verifying implementations. There are a few ways that we can solve this problem

- **Convert the scripts** – If we convert the scripts manually, they will run natively with `Python`. This is fairly straightforward for smaller programs, but can be a daunting task for longer scripts. For most longer scripts we would have to keep track of which variables are matrices/vectors and which are simply constants, in order to update the script correctly, as well as finding `Python` replacements for build-in functions that are not standard.
- **Saving the data** – `MATLAB` can save the data as a `.mat` file, which is fairly easy data-format to read, which `SciPy` can easily read. It requires that the problem can be replicated exactly in order to compare with the data, something which is not always possible.
- **Actual interface** – Many `Python` libraries exist that can directly interface with `MATLAB`. This requires the running computer to also have a `MATLAB` installation, which eliminates the absolute portability of the scripts. This can easily be used for verification purposes, and be disabled in the final scripts, allowing seasoned `MATLAB` coders to test their `Python` scripts against old experiences before shipping `Python` code.

We will not be dependent on `MATLAB` scripts that are big enough to not be converted properly. Most of the data we will be using for confirmation is also of the size to simply write into our files. This means that we will at no time be using the options of loading `MATLAB` data or interfacing with `MATLAB`.

4.2.3 `Python` overhead

`Python` is a general purpose programming language, and is thus not optimized to do the advanced mathematical computations, as `MATLAB` is. While `Python` has many features, they come with the price of an overhead time-cost `MATLAB` does not have. Unlike `MATLAB`, effort must be made if `Python` has to become as effective as `MATLAB`, but it is unlikely that it is possible, due to the de-centralized development of `Python`. While this overhead in general is not pronounced, it becomes quite pronounced when dealing with big systems as the ones we will handle in chapter 6. While this is unfortunate for aspiring numerical users of `Python`, it is not crippling to the process. This issue is magnified the more different packages are used for the calculations, so it is always a good idea to only import the packages you need and use when doing numerical computations in `Python`.

4.3 Implementation for spectral methods

Before we will be able to solve our problems with spectral methods, we will need to construct some of the parts that we will be using for our spectral solvers. These implementations or scripts will be used in the development of our spectral methods.

4.3.1 Generating the differential operators

The differential operators will be central to the collocation method, as these will find the numerical differentials of the nodal values by a simple matrix product. We will split this into the definition of the Lagrange Fourier matrix `DF` and the Legendre matrix `DP`, which will both be included in a `Python` script called `Diff.py`, from where we will import them when needed. We will not be utilizing the Fourier differential operator, but implementing it is necessary to be able to handle periodic boundaries in problems.

4.3.1.1 The Fourier differential operator

We recall the definition of the Fourier differential operator from (3.5), and note that it is only defined for a grid of even points. Using this, we will implement the matrix by the following algorithm.

```
Data: N - Positive integer designating the size of the array containing the
      equally spaced points.
Result: A Fourier differentiation matrix $D$
$D$ = Zero $NxN$ Matrix
for $i \leftarrow 0$ to $N-1$ do
    for $j \leftarrow 0$ to $N-1$ do
        if $j \neq k$ then
            $D[i][j] = \frac{1}{2}(-1)^{1+k+j}\cot\left(\frac{\pi(j-k)}{N}\right)$
        else
            $D[i][j] = 0$
        end
    end
end
```

Algorithm 1: Generating the Fourier differential matrix

The reason that we are excluding the last point, is that it is automatically assumed that the last point be equal to the first one by periodicity. The code for this implementation can be found in appendix D.1.

We will test this implementation by creating a function $v(x) = e^{\sin \pi x}$ defined on the grid $x \in [0, 2]$. This will have the true solution $v'(x) = \pi \cos(\pi x)e^{\sin \pi x}$, which we will be able to test up against. Using algorithm 1, we will calculate the derivatives, and the convergence with regards to the number of points N

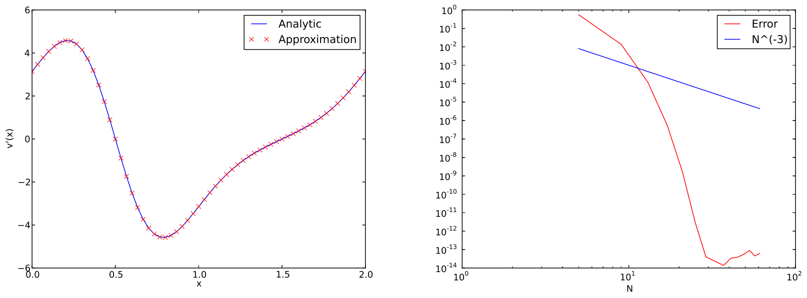


Figure 4.1: The numeric differential (by Fourier operator) of $v(x)$ to the left, and the convergence to the right.

Figure 4.1 shows that the differentials are quite precise at even relatively small numbers of N, and that the error converges faster than polynomial speed, satisfying the condition to be used in a spectral solver by having spectral accuracy.

4.3.1.2 The Legendre differential operator

The definition for the differential operator we will need for non-periodic functions is explained in section 3.1.2.2. Since we know that the differential operator is calculated as

$$D = \mathcal{V}_x \mathcal{V}^{-1}$$

Where $\mathcal{V}$ is the Vandermonde matrix, and $\mathcal{V}_x$ is the Vandermonde matrix of the differentiated basis polynomials, when we have calculated them, we will be able to construct the differential matrix. The package `DABISpectral1D` contains a routine to compute the Vandermonde matrix of derivative order n on a set of points x, which we will be using. This allows us to generate the differential operator as

```
Data: N - Positive integer designating the size of the array containing
        Legendre-Gauss-Lobatto quadrature points.
Result: A Legendre differentiation matrix $D$
$D$ = Zero $NxN$ Matrix
Create the Legendre-Gauss-Lobatto nodes $xGL$ from $N$ with
GaussLobattoQuadrature from DABISpectral1D
Create $\mathcal{V}$ from $N$ and $xGL$ with GradVandermonde from DABISpectral1D
Create $\mathcal{V}_x$ from $N$ and $xGL$ with GradVandermonde from DABISpectral1D
Solve the system $\mathcal{V}D = \mathcal{V}_x$ for $D$ with numpy.linalg.solve
```

Algorithm 2: Generating the Fourier differential matrix

This implementation can be found in appendix D.1.

We will test the algorithm 2 implementation on the same function as the test for algorithm 1.

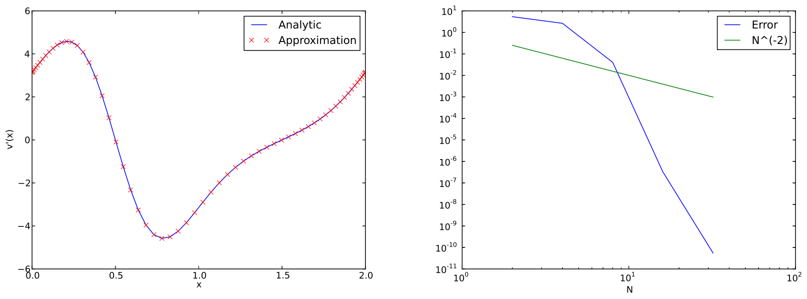


Figure 4.2: The numeric differential (by Legendre operator) of $v(x)$ to the left, and the convergence to the right.

As we can see in figure 4.2 the nodal points are not located with the same equal spacing as in figure 4.1. The nodal values are centered more towards the ends of the spectrum. We also see that the Legendre differential operator also achieves spectral convergence, qualifying it for use with spectral methods.

4.3.2 Employing the time-stepping method

For the time-dependent problems, we will need an established time-stepping algorithm in order to get dependable results. For this we will use the function `scipy.integrate.odeint` in most cases. This method is based on the LSODA algorithm from the `FORTRAN` ordinary differential equation solver pack, `ODEPACK`. It automatically adjusts the time-steps, adjusting to the problem stiffness, making it a good choice for us, since we will not have to calculate a stable solver for each problem.

The function is called as `odeint(func, y0, t, (args))`, where the inputs are

func	A right-hand-side function which returns the time-derivatives
y0	A vector of the initial states
t	A vector containing the time-values we would like to have output the function values for. The first element should be the initial time, and the last should be the last time.
args	A `Python` tuple, containing all the extra arguments to pass to `func`.

The function then returns a $\mathbb{R}^{N_y \times N_t}$ matrix, where each column represents a time-step.

Using this function will force us to define the right-hand-side function explicitly, and ordering all our values into a single vector. This will force us to gather all input values into a vector before using this, and including a splitting procedure in `func`. With this in mind, we can employ this as part of the solution for initial value problems.

An example of such a right hand side function can be found in the implementation of Burgers' equation from section 4.4.2.

```
def dudt(u,t,v,Dx,BoundaryFixer):
        #Calculate the linear operator
    unew= -u*np.dot(Dx,u)+v*np.dot(Dx,np.dot(Dx,u))

        #Adjust for boundaries
    unew = unew*BoundaryFixer
    return unew
```

4.3.3 Handling multi-dimensional problems

One problem that will arise will be how to handle multi-dimensional problems. The problem with multi-dimensional problems, is that not only will the problem likely need to be passed as a vector for some functions, but our differential operators are also only defined in the 1D spectrum.

4.3.3.1 Multiple dimensions in a single vector

To define a grid of values, we will use a vector of values along each dimension, x and y, and we will create the needed grid by the command `X,Y=numpy.meshgrid(x,y)`, which will net us the grids

$$X = \begin{bmatrix} x_0 & x_1 & \cdots & x_{N_x} \\ x_0 & \ddots & & \vdots \\ \vdots & & \ddots & \\ x_0 & \cdots & \cdots & x_{N_x} \end{bmatrix} \qquad Y = \begin{bmatrix} y_0 & x_0 & \cdots & y_0 \\ y_1 & \ddots & & \vdots \\ \vdots & & \ddots & \\ y_{N_y} & \cdots & \cdots & y_{N_y} \end{bmatrix}$$

We will need to be able to order an entire multi-dimensional array into one vector for the time-integration scheme we have developed. We will do this by utilizing some of the built-in functions of `Python`, `flatten` and `reshape`.

In order to vectorize our multi-dimensional array, we will use the build in `flatten` function, where we will specify we want to flatten it using the fortran standard, which is column-major – the first column will be the first part of the new vector, followed by the second column and so on (same result as the `MATLAB` notation `a(:)`). Now, we can reform the vector to its previous state using `reshape`, where we will need to specify the number of elements in each dimension, and the order with which we flattened it.

To avoid continuously flattening and reshaping, we will create a multi-dimensional array of indexes, which will include the index of each point in the vector, located at the point it should be in the matrix. We will accomplish this by creating an array of indexes, and reshaping this as we would have our vector. This allows us to use the notation `A[index[x,y]]` instead of reshaping `A` to use `A[x,y]`.

An example for the functions mentioned used in included below

```
x,wx = LegPol.GaussLobattoQuadrature(Nx)
y,wy = LegPol.GaussLobattoQuadrature(Ny)

X,Y = np.meshgrid(x,y)

u = u.flatten("F")

index = np.arange((Nx+1)*(Ny+1)).reshape(Ny+1,Nx+1,order='F').copy()
```

4.3.3.2 Differentiation of multiple dimensional array

Since our differential operators are only defined on the single dimensional space, we will need to find a way to differentiate the multi-dimensional arrays as if they were in a single dimension. We will use the vectorized version of the multi-dimensional array. If we recall that the first column would be ordered at the top, followed by the next column. If we individually used the differentiation operator on each column, we could differentiate those values along the first axis. To do this, we can generate a new matrix $D_X \neq D_x$, which would be defined as $D_X = D_x \otimes I_{N_y}$, where I_{N_y} is the identity matrix of size $N_y \times N_y$, and $\otimes$ is the kronecker product. This would allow us to use the dot product between D_X and our vector of values $\mathbf{v}$ to generate $\frac{dv}{dx} = D_X \cdot \mathbf{v}$. Using the same logic, we can differentiate along the other axis using $D_Y = D_y \otimes I_{N_x}$. For $N > 2$ dimensional problems, we would add a kronecker product for each dimension.

In order to verify this, we create a mesh from $x \in [-1, 1]$ and $y \in [-1, 1]$, where we will generate the function $\cos(xy)$ over. Differentiating this along both axis will give

$$\frac{d}{dx}\frac{d}{dy}\cos(xy) = -xy\cos(xy) - \sin(xy)$$

Which we can use to test the derivative qualities of the functions. We will be employing the Legendre derivative operators for this problem, and in both directions. The test algorithm in algoritm 3, which produces figure 4.3.

```
Data: None
Result: Plots detailing the error of our differentiation.
Generate $x$ and $y$ using GaussLobattoQuadrature from DABISpectral1D
Create a mesh $X, Y$ using numpy.meshgrid(x,y)
Create $D_x$ and $D_y$ using Algorithm 2
Calculate the $C = \cos(xy)$ and the correct derivative $C'$
Flatten $C$
Create $D_X$ and $D_Y$ using numpy.kron and numpy.identity
Calculate the approximated derivative using $C^d = D_X \cdot D_Y \cdot C$
Calculate and plot the error $|C^d - C'|$
```

Algorithm 3: Testing multi-dimensional differentiation.

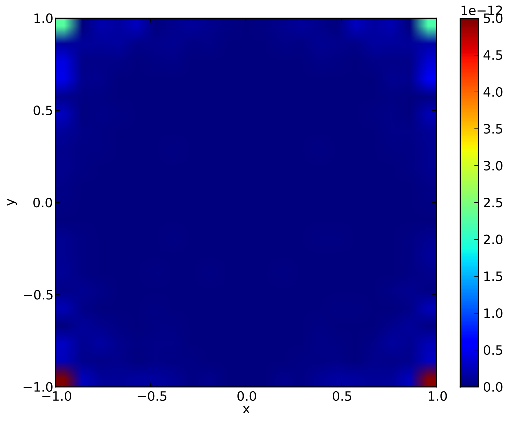


Figure 4.3: The error after differentiating $\cos(xy)$ along both axis numerically.

As figure 4.3 shows, the error is in the order of machine accuracy, showing that this differentiation scheme works as intended.

The full code for the test can be found in appendix D.2.

4.3.4 Sparse matrices

The multi-dimensional differential-operator matrices – as a kronecker product with the identity matrix – consist largely of zeroes. NumPy does not natively support sparse matrices, but SciPy includes a library that supports sparse matrices. This means that we will need to monitor which of our matrices are sparse, and modify the expressions where they are a part.

Since we will mainly be using this on the differential operators – from which all we need is the dot product – we can utilize the inherent function of the sparse matrices `A.dot(b)` to get the dot product $A \cdot b$ in a standard vector format.

SciPy offers many different formats for sparse matrices, each suited for different application, and we will be using a form classified as compact-sparse-row matrix, as this will be most efficient for the dot-product according to [JOP+].

In order to verify that this is indeed faster, and as correct, we will construct a test-case for this sparse matrix, where we will generate a random $1000x1000$ matrix A, with non-zero elements in 100 places in the first two rows, and nonzero elements along the diagonal, and generate a random nonzero $1000x1$ array b. We time both the sparse version of the dot product and the full version, and repeat this procedure 1000 times. This gives us an average of $2.778 \cdot 10^{-3}$ seconds for the full dot product, and only $1.289 \cdot 10^{-4}$ seconds in average for the sparse dot product, giving us a significant upgrade in speed. The average error $\|x^2_{\text{sparse}} - x^2_{\text{full}}\|_2$ was $5.4 \cdot 10^{-13}$, which is within acceptable bounds, as we can attribute that to machine accuracy.

The full code for this test can be found in appendix D.2.

4.4 Practical implementation

We have outlined how to implement the different parts of the code, and now we will implement the actual solutions to the problems from section 3.3, in order to verify the implementation works, and the convergence behaves as expected.

4.4.1 Implementing the spectral collocation method

We will use the problem from section 3.3.1, which is defined as

$$-\epsilon\frac{d^2}{dx^2}u - \frac{d}{dx}u = 1 \qquad\qquad u(0) = u(1) = 0$$

And has the solution (3.9).

Since the derivation was handled in the previous chapter, we will simply implement the method here, using the routines described in this chapter. We choose the Legendre polynomials to model the equations, which means that we will generate the differential matrices using algorithm 2. Since this algorithm already uses the untransformed points, we will use these as well for our problem.

The construction and modification of our linear operator to adhere to boundary points is done as follows

```
    #Construct L
L = -epsilon*4*np.dot(D,D) + b*2*D

#Implement boundary conditions
L[0,:] = 0
L[-1,:] = 0
L[0,0] = 1
L[-1,-1] = 1

#Generate right-side function with boundary conditions
f = np.ones((n,1))
f[0]  = 0
f[-1] = 0
```

After having constructed our linear operator, we will use the function `solve` from the `numpy.linalg` pack to find the solution. We will continuously increase the amount of points between $10 \leq N \leq 80$ in order to generate an error convergence plot.

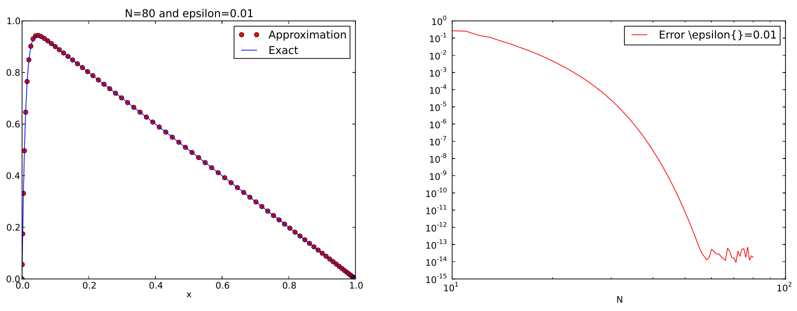


Figure 4.4: To the left: The calculated solution using $N = 80$ to the problem given in (3.8), compared to the correct solution as given in (3.9). To the right: The error for the approximation as a function of N

Figure 4.4 shows us that the solution is indeed spectral in convergence, and that the spectral collocation method can be implemented with the previously described methods.

The full code for the test case can be found in appendix D.3.

4.4.2 Burgers' equation – implementing the solver

We will be implementing the solution we found in section 3.3.2, which is the viscous Burgers' equation described as

$$\begin{aligned} &u_t + uu_x = vu_{xx} && x \in [-1, 1] \\ &u(-1, t) = 1 && u(1, t) = -1 \quad , \forall t > 0 \end{aligned}$$

Since the problem has Dirichlet boundary conditions, we will be using the Legendre polynomials to, and thus the differential operator from algorithm 2. Using the derived form from (3.11), we construct a separate right-hand-side function, in which we include the condition that the time-derivative will be zero on the boundaries.

The right hand side and the "boundary fixer" are constructed as such

```
#Create boundary fixer to cancel the time derivatives
# at the boundaries.
```

```
BDFix = np.ones(x.shape)
BDFix[0] = 0
BDFix[-1] = 0

def dudt(u,t,v,Dx,BoundaryFixer):
    #Calculating linear operator
    unew= -u*np.dot(Dx,u)+v*np.dot(Dx,np.dot(Dx,u))
    #Adjust for boundaries
    unew = unew*BoundaryFixer
    return unew
```

By multiplying the time-derivative with BDFix, all values will remain the same, except at the boundaries, where the boundary values time-derivative is equal to zero.

We use the time-stepping function described in section 4.3.2, where we generate the initial condition from the function $u_0(x) = -\tanh(x)\frac{1}{|-\tanh(x_0)|}$.

In order to generate a numerical solution, we will need to define $v > 0$, which we choose to be $v = 0.1$.

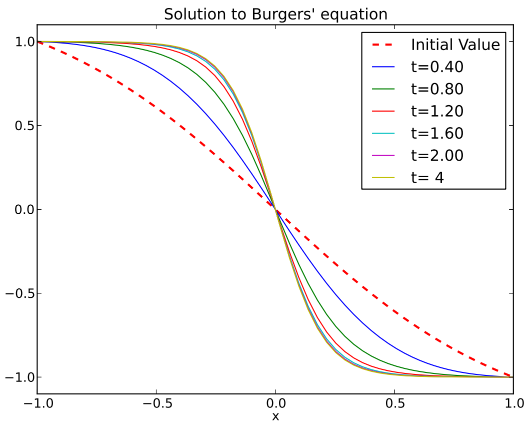


Figure 4.5: The solution to Burgers' viscous equation with $v = 0.1$.

We see in figure 4.5 that the equation quickly achieves a steady state.

The full code for the test case can be found in appendix D.3.

Chapter 5

Stochastic formulation and uncertainty quantification

The influence of measuring errors or other uncertain quantities in a differential equation can sometimes change the solution drastically. To enable us to to correctly gauge the effect of uncertainties, this chapter describes how to quantify and implement these factors into our differential equation models.

This will an effective aid in assessing the impact uncertainty can have on complicated systems where it can often be very hard or impossible to predict. We will mainly be using the method called generalized polynomial chaos, which allows us to drastically reduce the time needed to do calculations with acceptable errors – making the effort worthwhile even under a certain time-constraint.

5.1 Probability theory

In this section, we will explore the probability theory needed to utilize and understand uncertainty quantification. We will start by introducing the basic concepts for probability and stochastic computations, and move on to how we will formulate and solve stochastic problems numerically. We will draw heavily on [Xiu10, Chp. 2], which presents and defines these concepts.

5.1.1 Basic concepts

In order to include stochastic variables in our models, we need to classify these and the concepts associated with them.

5.1.1.1 Random variables

The concept of random variables is essential to the formulation of our problems. A random variable is, in essence, what allows us to represent the random nature of our system. We assign a possible outcome the designation ω, wether it be a physical outcome, such as heads or tails in a coin-flip, or a numerical outcome. This allows us to create the random variable concept as a real-valued function dependent on ω, such as the random variable $X = X(\omega)$. The random variable can now always be represented in a mathematical model, despite the outcomes not always being mathematical in nature.

This allows us to represent information about a given mechanic, where we can define how the outcome might work, but might not be sure how the exact mechanics work.

The random variables in this thesis will have outcomes already defined on the numerical scale, and but will retain the notation $X(\omega)$ as it is still a function of the random outcome.

5.1.1.2 Distributions

Random variables are all associated with a given probability, which measures the likelihood that each outcome will be the realization of ω. The probability of each outcome is a number $0 \leq p \leq 1$, where 1 signifies that the outcome always happens, and 0 that it never happens. Associated with the probability is the distribution function F_X, which is the accumulated probabilities $F_X(x) = P(X \leq x)$.

The distribution is often used to characterize the variable for continuous distributions, which are a large part of what we will be working with. Most continuous distributions also have a *density* f_X which is used to characterize the probability for a given range of outcomes. Since the distribution is continuous it follows that there is exactly $P(X = x) = 0$ chance for a given outcome to occur. We therefore define the density as

$$F_X(x) = \int_{-\infty}^{x} f_X(y)\,\mathrm{d}y$$

Where $f_X(x) \geq 0$ and the sum of all outcomes is exactly 1, $\int_{-\infty}^{\infty} f_X(y)\,\mathrm{d}y = 1$.

The two distributions we will be working with will be the *gaussian* distribution and the *uniform* distribution. The density of the gaussian distribution $\mathcal{N}\big(\mu, \sigma^2\big)$ where $x \in \mathbb{R}$ is defined as

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{5.1}$$

And the uniform distribution $U(a, b)$ is constant for $x \in [a, b]$

$$f_X(x) = \begin{cases} \frac{1}{b-a} & x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

5.1.1.3 Moments and expectation

Moments are a very useful way of describing a random variable X, since we can characterize the behavior of a random variables using a few concepts. The mth moment is defined as

$$\mathbb{E}[X^m] = \int_{-\infty}^{\infty} x^m f_X(x)\,\mathrm{d}x$$

The most commonly used moment is probably the first moment, which is also called the *expectation* or the *mean value*. This characterizes the most likely value to be assumed by the variable, hence the name *expectation*. It is typically denoted as μ, and is used as this to characterize the gaussian distribution, as seen in (5.1).

With the expectation defined, we can define another widely-used concept derived from moments, which are the centered moments. They are defined as

$$\mathbb{E}[(X - \mu_X)^m] = \int_{-\infty}^{\infty} (x - \mu_X)^m f_X(x)\,\mathrm{d}x$$

The first centered moment will always evaluate to zero. The second centered moment is called the *variance*, denoted σ^2, and is the square of the standard deviation σ. The standard deviation is used to describe how far the variation is around the mean, for example 95% of values are within $\mu \pm 2\sigma$ for a gaussian distribution. It is worth noting that a gaussian distribution is completely characterized by the mean and standard deviation.

We will operate with functions of random variables rather than random variables themselves. Many of the concepts can be directly applied to functions as well, although the mathematical calculation differs a bit. For any real valued function g the expectation is calculated by

$$\mathbb{E}[g(X)] = \int_{-\infty}^{\infty} g(x) f_X(x)\,\mathrm{d}x$$

5.1.2 How to formulate a stochastic problem

To investigate the effects of having stochastic variables in the formulation of our system, we will need to formulate the problems differently. Given a differential system with stochastic variables, we will need to identify and parameterize the stochastic inputs, allowing us to do computations based on these.

Parametrization is generally uncomplicated if the stochastic variables are already operating in $\mathbb{R}^N$, and are independent. Since this thesis is generally concerned with already parameterized domains, we can typically model our stochastic variables to this domain – making further parameterization unnecessary.

5.1.2.1 Generalized polynomial chaos

Generalized polynomial chaos (gPC) is a technique used for parametrization of a stochastic variable. It involves modeling a stochastic variable by an appropriately chosen polynomial, allowing easing of the calculations for expectations. By modeling the stochastic variables as a polynomial, we can choose the values of our stochastic variables with a deterministic routine instead of allowing these values to be "random".

In general the basis functions of gPC are the polynomials satisfying

$$\mathbb{E}[\Phi_m(Z)\Phi_n(Z)] = \gamma_n \delta_{mn} \quad m, n \in \mathcal{N} \tag{5.2}$$

It is clear from the delta-function, that these polynomials must be orthogonal, and that

$$\gamma_n = \mathbb{E}\left[\Phi_n^2(Z)\right] \quad n \in \mathcal{N}$$

For the probability density function $\rho(z)$, we can calculate the expectation from

$$\mathbb{E}[\Phi_m(Z)\Phi_n(Z)] = \int \Phi_m(z)\Phi_n(z)\rho(z)\,\mathrm{d}z = \gamma_n\delta_{mn} \tag{5.3}$$

[Xiu10, Table 5.1] suggest that for gaussian distributions, Hermite polynomials are chosen for the basis, and for uniform distributions Legendre polynomials are chosen.

Hermite polynomials Hermite polynomials are a family of orthogonal polynomials defined as

$$H_n(x) = (-1)^n e^{x^2/2}\frac{d^n}{dx^n}e^{-x^2/2}$$

And satisfying

$$\int_{-\infty}^{\infty} H_m(x)H_n(x)w(x)\,\mathrm{d}x = n!\delta mn \qquad w(x) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2} \tag{5.4}$$

The similarity between the weight function and the actual density for a gaussian variable suggests that Hermite polynomials are easy to use when modeling gaussian variables. The density of gaussian variables is

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{5.5}$$

Which means that the Hermite weight function is exactly the density of the gaussian variable $X\ \mathcal{N}(0,1)$. The first six Hermite polynomials are visualized in figure 5.1

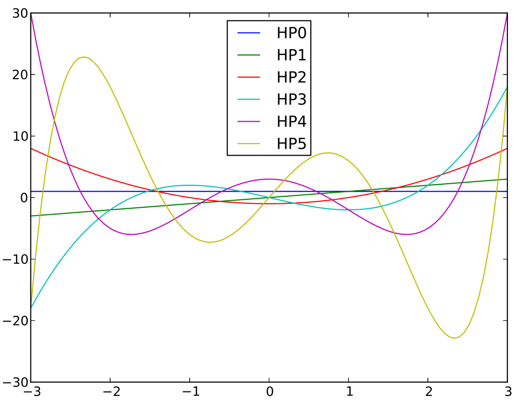


Figure 5.1: The first six Hermite polynomials visualized

5.1.3 Calculating the expectation

The expectation of a stochastic variable X is defined as

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f_X(x)\,\mathrm{d}x$$

For our calculations, we will not always have the analytic solution to the distribution of our values we will not be able to use this integral, but rather a quadrature. For a set of N points chosen at random, we will assign each point the same weight, giving us

$$\mathbb{E}[X] = \sum_{n=0}^{N} \frac{1}{N} x_n \qquad \text{for large } N$$

This is a a result of the law of large numbers, which states that

$$\lim_{N\to\infty} \sum_{n=0}^{N} \frac{1}{N} x_n = \mu_x$$

5.1.3.1 Gaussian distribution

For a gaussian distribution, we use the Hermite polynomials to approximate the variable, we will instead use the Gauss-Hermite quadrature, which is defined as

$$\int_{-\infty}^{\infty} f(x) e^{-\frac{x^2}{2}}\,\mathrm{d}x \approx \sum_{i=1}^{n} w_i f(x_i) \tag{5.6}$$

If we want to calculate the expectation of the modeled variable, we use the definition from section 5.1.1.3

$$\mathbb{E}[g(X)] = \int_{-\infty}^{\infty} g(x) f_X(x)\,\mathrm{d}x$$

Since we will only be modeling gaussian variables by Hermite polynomials, we can calculate these expectations as

$$\mathbb{E}[H(X)] = \int_{-\infty}^{\infty} \frac{1}{\sigma\sqrt{2\pi}} H(x) e^{-\frac{(x-\mu)^2}{2\sigma^2}}\,\mathrm{d}x$$

Since we would like to use (5.6), we will need to employ a transformation to $x \to y$, such that $e^{-\frac{(x-\mu)^2}{2\sigma^2}} = e^{-\frac{y^2}{2}}$

$$y = \frac{x-\mu}{\sigma} \Leftrightarrow x = \sigma y + \mu$$

This would allow us to calculate the expectation by (5.6)

$$\mathbb{E}[H(X)] = \int_{-\infty}^{\infty} \frac{1}{\sqrt{\pi}} H(\sigma y + \mu) e^{-\frac{y^2}{2}} \,\mathrm{d}x \approx \frac{1}{\sqrt{\pi}} \sum_{i=1}^{n} w_i H_i(\sigma y_i + \mu) \tag{5.7}$$

5.1.3.2 Uniform distribution

If we want to use the uniform distribution instead of the gaussian, we will be using the Legendre-Gauss quadrature instead. We assume a random variable Y is following the uniform distribtion $Y \sim U(a, b)$

$$\mathbb{E}[L(Y)] = \int_a^b L(y) f_Y(y) \,\mathrm{d}y = \int_a^b L(y) \frac{1}{b-a} \,\mathrm{d}y$$

Since the Legendre-Gauss quadrature only works on the interval $[-1, 1]$, so we transform y

$$x = \frac{b-a}{2} y + \frac{a+b}{2} \quad \frac{dx}{dy} = \frac{b-a}{2}$$

This allows us to calculate the expectancy by the Legendre-Gauss quadrature

$$\mathbb{E}[L(Y)] = \frac{b-a}{2} \int_{-1}^{1} L(x) \frac{1}{b-a} \,\mathrm{d}x = \frac{1}{2} \int_{-1}^{1} L(x) \Leftrightarrow$$

$$\mathbb{E}[L(Y)] \approx \frac{1}{2} \sum_{i=1}^{N} w_i L_i(x_i) \tag{5.8}$$

5.2 Quantification of uncertainty

Uncertainty is differentiated between the two causes of uncertainty in a model, the statistic uncertainty, called *aleatoric* uncertainty, and systematic uncertainty, called *epistemic* uncertainty. Aleatoric uncertainty is the inherent uncertainty in many problems, such as measuring errors and errors due to fabrication differences of the products used. This uncertainty is hard to suppress in many cases, as it is increasingly difficult to increase measuring accuracy. Based on the understanding of the model, most aleatoric uncertainty can be quantified, since we are aware of the sources of uncertainty, we can bound them to a quantifiable domain.

Epistemic uncertainty is the uncertainty inherent in our model, if our model neglects certain effects either due to inferior knowledge of the system, or due to simplification of the model – for example the exclusion of air-friction. Epistemic uncertainty is hard to quantify by design, as it is often not measurable. We will be concerning ourselves with aleatoric uncertainty, exploring the effects errors on the input might have on the result.

While we already posses the knowledge to quantify the uncertainty of a variable, what we really need to establish is how uncertainty propagates through a model. The simplest way to do this would be to simply run the model multiple times with different input, but as we will discuss in the next chapter, it is not optimal. The general way we will be showing how the uncertainty propagates is also by the use of *mean* and *standard derivation*. This will simplify most of the points through plots, as we will characterize each function value in a certain point as its own random variable, we will easily be able to calculate the expectation and variance of the function using the methods from section 5.1.1.3 and quadrature rules.

For our models we will start by constructing a model solution for a problem with fixed constants, allowing us to verify the correctness of our solution. We will then introduce the uncertainty in variables after the model has been established, and use the complete model to calculate how the uncertainty propagates. This puts some limits on our models, as we cannot account for random variables that change according to a random pattern - at least not without modifying the solution model. This generally limits us to allowing stochastic input and not stochastic processes.

5.3 Sampling methods

Sampling methods are the different way we will choose the values of our random variables. For our problems we will stick to non-intrusive methods, though we will quickly touch on what an intrusive method is.

5.3.1 Non-intrusive methods

In this section we will explore some of the non-intrusive methods for calculating propagating uncertainty through a model. That the methods are not intrusive means that we will simply vary the input to the model, and not the actual solution process. This allows us to reuse our deterministic model for an uncertain problem, easing the derivation and implementation of our methods, and the verification of our models.

5.3.1.1 Monte Carlo method

The Monte Carlo method is derived from the name of a big casino in Monaco. The name is descriptive for the method, which essentially is about randomly generating input by the distribution of the variable. For each input generated, the model is run, and the output collected for calculation of expectation and moments. This makes it extremely simple to implement, as it will typically reduce to a simple iteration over the solution of the problem. This does not provide a very efficient way of solving our problems however, since the convergence rate, according to [Xiu10, p. 54] is $O(M^{-1/2})$, which means that we will need a hundred times more calculation for a precision increase in one digit. This is very inefficient, since most problems does not allow for quick computation, and it will require a large amount of samples to generate a stable basic solution for most problems. The Monte Carlo method relies on the law of large numbers, which states that for an infinite set of random points x, we will be able to calculate the exact mean by

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{N} x_i = \mu_x$$

5.3.1.2 Stochastic collocation method

The stochastic collocation method derives its name from the numerical collocation method – since they both require the residual of the approximation to be zero in the *collocation* points. A stochastic collocation method requires that the solution to the model is formed from the results of the model being run on different sets of nodes from the random space. This essentially qualifies the Monte Carlo method as a collocation method as well, but the sense we will be using it in employs the use of general polynomial chaos. The method will depend on choosing a set of points to accurately represent the polynomial as detailed in section 5.1.2.1, solving the model with these inputs, and then calculating the results based on the chosen polynomial type. This requires some more calculations than the Monte Carlo method, but is still fairly easy to implement. Once the nodes for the random space has been chosen, and their weights calculated, the method is calculated as normal. This makes it a preferable method, since it is easy to implement on most systems, and vastly improves the convergence over the Monte Carlo method.

In summation, the collocation method can be described as an algorithm.

```
Data: A deterministic solver for a differential equation, the stochastic
      definition of a variable - $\alpha$
Result: The mean and variance of the stochastic function
$(x_N, w_N) \leftarrow$ Generate points and weights according to gPC polynomial.
Generate $\alpha$ based on $x$.
$\mathbb{E} = 0$
for $a_0$ to $a_N$ do
  | $u_i \leftarrow$Solution to system for $a_i$
end
$\mathbb{E} = \sum_{i=0}^{N} u_i w_i$
$\mathbb{E}_{\text{var}} = \sum_{i=0}^{N} (u_i - \mathbb{E})^2 w_i$
```

Algorithm 4: The Stochastic Collocation Method

5.3.2 Intrusive methods

Intrusive methods are characterized by their need to modify the deterministic solver. The reason for this need is, with the Stochastic Galerkin method, that the different differential equations we need to solve can be coupled, and as such need to be solved as a system of equations rather than as a single equation.

5.3.2.1 Stochastic Galerkin method

The stochastic Galerkin method is a method that uses expectations to guarantee that a correct solution is found. When the appropriate gPC basis functions are chosen, we will formulate the system as just the expectancies and using the polynomial approximations. For a system $u_t = \mathcal{L}(u)$ which is modeled using Hermite polynomials, the equations that need to be satisfied are

$$\mathbb{E}[(v_N)_t H_k(Z)] = \mathbb{E}[\mathcal{L}(v_N) H_k(Z)] \quad \text{for } k \leq N \tag{5.9}$$

Where $v_N = \sum_{i=0}^{N} \hat{v}_i H_i(Z)$ is our solution.

This procedure allows us to construct a system of coupled differential equations based on the appropriate quadrature for the basis polynomial. The exact form of the system will differ from problem to problem, as the problem is defined from (5.9), however for linear problems, they can be reduced to a simple matrix system of the form $\frac{d\mathbf{v}}{dt}(t) = \mathbf{A}^T\mathbf{v}$.

5.4 Examples of uncertainty quantification

In this section, we will be going through the derivation of the methods we can use for uncertainty quantification. We will be starting with the derivation of the test-equation, where $u' = -\alpha u$, which will make the derivations easy. We will use this equation to show some of the different methods we can use to solve an uncertain system. We will then be expanding into Burgers' equation, where we will show the impact small uncertainties can have on the solution to a model.

5.4.1 The test equation – stochastic collocation method

The test equation is characterized as

$$\frac{du(t)}{dt} = -\alpha u, \quad u(0) = \beta \tag{5.10}$$

5.4.1.1 Gaussian distributed variable

In order to test our stochastic system, we assume that the the α parameter is a Gaussian random variable with mean μ and standard deviation σ, such that $\alpha(\omega) \sim \mathcal{N}(\mu, \sigma^2)$. The full solution to the problem is given in

$$u(t, \omega) = \beta e^{-\alpha(\omega)t}$$

Since we possess the analytical solution to the problem, we can calculate the expectation. We will be using the definitions of expectations of functions from section 5.1.1.3.

$$\begin{aligned}\mathbb{E}[u(t,\omega)] &= \mathbb{E}\left[\beta e^{-\alpha(\omega)t}\right] = \mathbb{E}[\beta]\mathbb{E}\left[e^{-\alpha(\omega)t}\right] \Leftrightarrow \\ \mathbb{E}[u(t,\omega)] &= \mathbb{E}[\beta]\int_{-\infty}^{\infty} e^{-xt} f_\alpha(x)\,\mathrm{d}x\end{aligned}$$

Since we know that the distribution of $\alpha(\omega)$ is a gaussian random variable, we know from (5.1) that the density is $f_\alpha(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$. This leads us to

$$\begin{aligned}\mathbb{E}[u(t,\omega)] &= \mathbb{E}[\beta]\int_{-\infty}^{\infty} e^{-xt}\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}\,\mathrm{d}x \Leftrightarrow \\ \mathbb{E}[u(t,\omega)] &= \mathbb{E}[\beta]\frac{1}{\sqrt{2\pi\sigma^2}}\int_{-\infty}^{\infty} e^{-xt} e^{-\frac{(x-\mu)^2}{2\sigma^2}}\,\mathrm{d}x\end{aligned}$$

Where we will solve the infinite integral using `Maple`

$$\mathbb{E}[u(t,\omega)] = \mathbb{E}[\beta]\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2}t\left(-t\sigma^2+2\mu\right)}\sqrt{2\sigma^2\pi} \Leftrightarrow$$
$$\mathbb{E}[u(t,\omega)] = \mathbb{E}[\beta]e^{-\frac{1}{2}t\left(-t\sigma^2+2\mu\right)} \tag{5.11}$$

We will want to calculate the variance as well, which is done by the same procedure as the expectation, but with the second centered moment

$$\mathbb{E}\left[\left(\beta e^{-\alpha(\omega)t} - \mathbb{E}\left[\left(\beta e^{-\alpha(\omega)t}\right)\right]\right)^2\right] = \mathbb{E}\left[\left(\beta e^{-\alpha(\omega)t} - \beta e^{-\frac{1}{2}t\left(-t\sigma^2+2\mu\right)}\right)^2\right] \Leftrightarrow$$
$$\mathbb{E}\left[\left(\beta e^{-\alpha(\omega)t} - \mathbb{E}\left[\left(\beta e^{-\alpha(\omega)t}\right)\right]\right)^2\right] = \frac{1}{\sqrt{2\pi\sigma^2}}\int_{-\infty}^{\infty}\left(\beta e^{-xt} - \beta e^{-\frac{1}{2}t\left(-t\sigma^2+2\mu\right)}\right)^2 e^{-\frac{(x-\mu)^2}{2\sigma^2}}\,\mathrm{d}x$$

This is solveable in `Maple`, which gives us the solution

$$\mathbb{E}\left[\left(\beta e^{-\alpha(\omega)t} - \mathbb{E}\left[\left(\beta e^{-\alpha(\omega)t}\right)\right]\right)^2\right] = \beta^2\left(e^{t^2\sigma^2}-1\right)e^{-2t\mu+t^2\sigma^2} \tag{5.12}$$

(5.11) and (5.12) allows us to properly calculate the expectancy and variance, and evaluate the correctness of our solution. We will set $\beta = 1$.

As u is simply a point in time, we will not need a spectral solver for this problem. We will start by using a Monte Carlo approximation to the problem, where we will be generating a new $\alpha(\omega)$ for each iteration of the problem. We will be calculating the running mean for each step in the approximation, as to show convergence. We will be generating the random parameter with the `Python` function `numpy.random.normal`

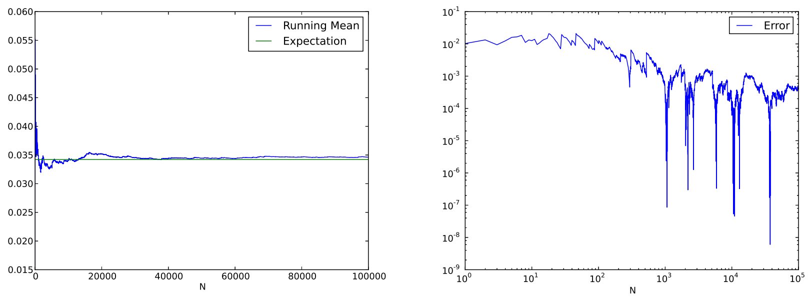


Figure 5.2: The mean calculated by the Monte Carlo method. The running mean is seen compared to the analytic mean in the left picture, and the running error is seen in the right picture.

We see in figure 5.2 that we need about 10^5 realizations of the model to get an accuracy of *about* 10^{-3}. The full code to this calculation is in appendix E.2.1.

To try and improve this, we implement the stochastic collocation method instead, as described in algorithm 4. We elect to use Hermite polynomials, as we have a gaussian distribution, and will use the Gauss-Hermite quadrature detailed in (5.7).

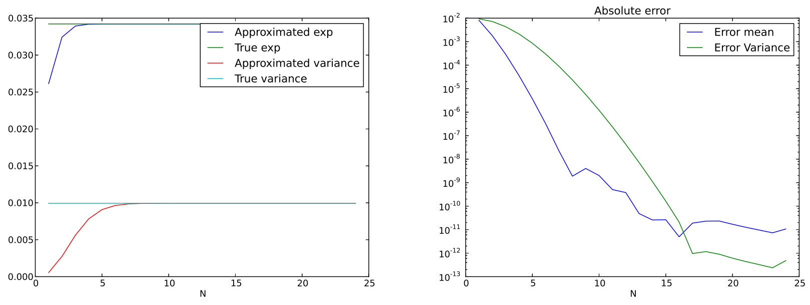


Figure 5.3: The mean calculated by the stochastic collocation method. The mean is seen compared to the analytic mean in the left picture, and the error is seen in the right picture.

We see in figure 5.3 that the collocation methods nets us a machine-error at $N = 15$ – for both mean and variance – which is a vast improvement based only on choosing the α_i based on a quadrature rule, to approximate the integrals with a gaussian quadrature instead of the Riemann integral. The full code used in this calculation is in appendix E.2.1.

5.4.1.2 Uniformly distributed variable

For a uniformly distributed variable we will of course be needed to recalculate the analytic expectation with regard to the new distribution. Since we have a uniform distribution, where $\alpha \sim U(a, b)$, we have a density function $f_\alpha(x) = \frac{1}{b-a}$ for $x \in [a, b]$ and $f_\alpha(x) = 0$ otherwise. We will now calculate the correct expectation.

$$
\begin{aligned}
\mathbb{E}[u(t)] &= \mathbb{E}[\beta]\int_{-\infty}^{\infty} e^{-xt} f_\alpha(x)\,\mathrm{d}x \Leftrightarrow \\
\mathbb{E}[u(t)] &= \mathbb{E}[\beta]\int_{a}^{b} e^{-xt}\frac{1}{b-a}\,\mathrm{d}x \Leftrightarrow \\
\mathbb{E}[u(t)] &= \mathbb{E}[\beta]\frac{1}{b-a}\int_{a}^{b} e^{-xt}\,\mathrm{d}x \Leftrightarrow \\
\mathbb{E}[u(t)] &= \mathbb{E}[\beta]\frac{1}{b-a}\frac{e^{-at}-e^{-bt}}{t}
\end{aligned}
$$

Using this, we will repeat the above exercise using Monte Carlo method in figure 5.4, for $\alpha \sim U(-7, 2)$

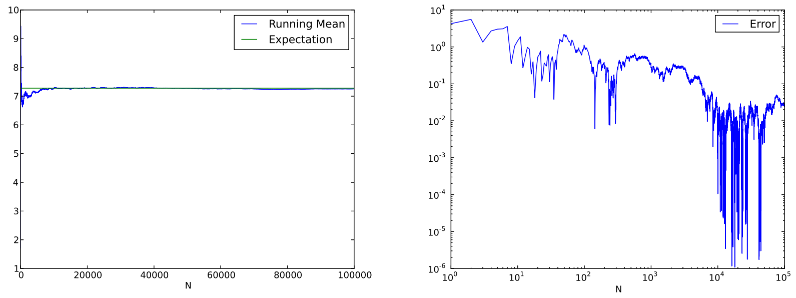


Figure 5.4: The mean calculated by the Monte Carlo method on a uniform distribution. The running mean is seen compared to the analytic mean in the left picture, and the running error is seen in the right picture.

The full code for the calculation can be found in appendix E.2.1.

For the stochastic collocation method, we will be using Legendre polynomials, and the Legendre-Gauss quadrature detailed in (5.8).

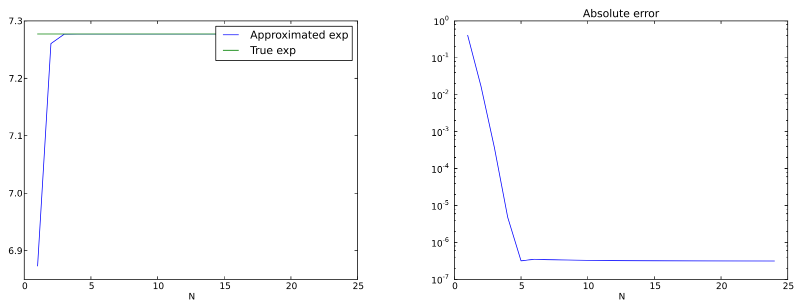


Figure 5.5: The mean calculated by the stochastic collocation method for a uniformly distributed variable. The mean is seen compared to the analytic mean in the left picture, and the error is seen in the right picture.

We see the result in figure 5.5, which shows us the same spectral convergence as was achieved in figure 5.3. The full code for the calculation can be found in appendix E.2.1.

5.4.1.3 A comparison between the Monte Carlo method and the stochastic collocation method

In both the uniformly and gaussian distribution cases, we see a more rapid convergence with the stochastic collocation method than with the Monte Carlo method. Combined with the minimal effort (after the derivation) to shift to a stochastic collocation method suggests we will be using this method from this step forward. The Monte Carlo method keeps one key aspect, as it guarantees $\sqrt{N}$ convergence, and is almost impossible to implement incorrectly, allowing us to use it to verify our models for models where we do not have an exact solution.

5.4.2 The test equation – stochastic Galerkin method

We will briefly showcase the stochastic Galerkin method, which is another way to achieve spectral convergence towards the mean. We will be modeling the differential equation (5.10) with the Galerkin method. We will assume α is a gaussian distributed value, allowing us to calculate the expectation by (5.11). In accordance to the method described in section 5.3.2.1, since $u(0) = \beta$ we will be seeking a solution

$$v_N = \sum_{i=0}^{N} \hat{v}_i H_i(Z)$$

where

$$\alpha_N = \sum_{i=0}^{N} a_i H_i(Z) \qquad \beta_N = \sum_{i=0}^{N} b_i H_i(Z)$$

We can truncate the last two sums, as only the two first a_is are defined as not zero - as $a_0 = \mu$ and $a_1 = \sigma$ - as well as the first b_i - $b_0 = \beta$. Using the Galerkin method, we acquire the formulation of the system

$$\mathbb{E}\left[\frac{dv_n}{dt} H_k(Z)\right] = \mathbb{E}[-\alpha_N v_N H_k(Z)] \quad \text{for } k = 0, \ldots, N$$

If we substitute our gPC approximations into this system, we get

$$\mathbb{E}\left[\frac{d}{dt}\sum_{j=0}^{N} \hat{v}_j H_j(Z) H_k(Z)\right] = \mathbb{E}\left[-\sum_{i=0}^{N} a_i H_i(Z) \sum_{j=0}^{N} \hat{v}_j H_j(Z) H_k(Z)\right] \quad \text{for } k = 0, \ldots, N$$

Where we can draw the coefficients out, since they are not affecting the calculation of the expectancy.

$$\frac{d}{dt}\sum_{j=0}^{N} \hat{v}_j \mathbb{E}[H_j(Z) H_k(Z)] = -\sum_{i=0}^{N}\sum_{j=0}^{N} a_i \hat{v}_j \mathbb{E}[H_i(Z) H_j(Z) H_k(Z)] \quad \text{for } k = 0, \ldots, N$$

Since we know our basis functions have the orthogonality quality described in (5.2), we can reduce the right-hand side of the equation

$$\frac{d}{dt}\hat{v}_k \gamma_k = -\sum_{i=0}^{N}\sum_{j=0}^{N} a_i \hat{v}_j \mathbb{E}[H_i(Z) H_j(Z) H_k(Z)] \quad \text{for } k = 0, \ldots, N \Leftrightarrow$$

$$\frac{d}{dt}\hat{v}_k = \frac{-1}{\gamma_k}\sum_{i=0}^{N}\sum_{j=0}^{N} a_i \hat{v}_j \mathbb{E}[H_i(Z) H_j(Z) H_k(Z)] \quad \text{for } k = 0, \ldots, N$$

We know that $\gamma_k = \mathbb{E}\big[H_k^2(Z)\big]$, but we will use the analytical version shown in [Xiu10, Eq. 6.8], as $\gamma_k = k!$ for Hermite polynomials. We can use Gauss-Hermite quadrature to calculate the expectancy from the right-hands side, as

$$e_{ijk} = \mathbb{E}[H_i(Z) H_j(Z) H_k(Z)] = \int_{-\infty}^{\infty} H_i(Z) H_j(Z) H_k(Z) dF_Z(z) \Leftrightarrow$$

$$e_{ijk} = \sum_{m=0}^{M} H_i(z_m) H_j(z_m) H_k(z_m) w_m$$

Using this gives us the system

$$\frac{d\hat{v}_k}{dt} = \frac{-1}{k!} \sum_{i=0}^{N} \sum_{j=0}^{N} a_i \hat{v}_j e_{ijk}$$

Which can be formulated as the matrix system

$$\frac{d\hat{\mathbf{v}}}{dt} = \mathbf{A}^T \mathbf{v} \qquad A_{jk} = \frac{-1}{k!} \sum_{i=0}^{N} a_i e_{ijk}$$

Now the only thing we need is the definition of v_0, which is equal to b_n. We will implement this with $\mu = 9$ and $\sigma = 3$, for easy comparison to the Month Carlo method calculated in section 5.4.1.1. The method we have derived produces the coefficients for the Hermite transformation, and we need to transform these back to get our proper values.

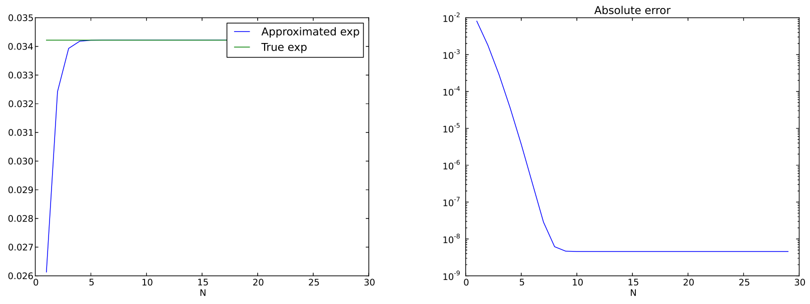


Figure 5.6: The mean calculated by the stochastic Galerkin method. The mean is seen compared to the analytic mean in the left picture, and the error is seen in the right picture.

We see in figure 5.6, that Galerkin method converges faster than the collocation method for the same problem, described in figure 5.3. This is due to the fact that the collocation method introduces aliasing errors by working with an interpolation approach, whereas the Galerkin method is not prone to these errors, as the formulation (5.9) ensures that the residue is orthogonal to the linear space spanned by the gPC polynomials, as described in [Xiu10, pg. 68]. This comes at the cost of a more complicated derivation, that needs to be redone for each problem, and a coupled system of equations, which necessitate designing a solver which can handle this. The full code for the stochastic Galerkin method is located in appendix E.2.1.

5.4.3 Burgers' equation – the influence of uncertainty

We recall the viscous Burgers' equation from section 3.3.2

$$\begin{array}{ll} u_t + uu_x = vu_{xx} & x \in [-1,1] \\ u(-1,t) = 1 & u(1,t) = -1 \quad ,\forall t > 0 \\ v > 0 & \end{array}$$

With the initial condition $u(x,0) = -\tanh(x)\frac{1}{|-\tanh(-1)|}$

We saw in figure 4.5 that the solution assumes a steady state which features a steady curve with a rapid shift in the middle of the spectrum.

5.4.3.1 Introducing uncertainty

To introduce uncertainty, we will make the value of v uncertain, as it will allow us to model the problem the same way we modeled it for the test equation, and we will use the stochastic collocation method to approximate the stochastic variable. We will let v attain a uniform distribution, since we know that v must always be greater than zero, disallowing us to model it with a gaussian variable. This gives us $v\ U(0.05, 0.2)$, which allows us to model the uncertainty over v. We will implement the uncertainty dimension much as we would with a 2-dimensional spectral method, using one vector to represent all the solutions, and reforming the solution afterwards. This will require some code-wise tricks described in section 4.3.3.1.

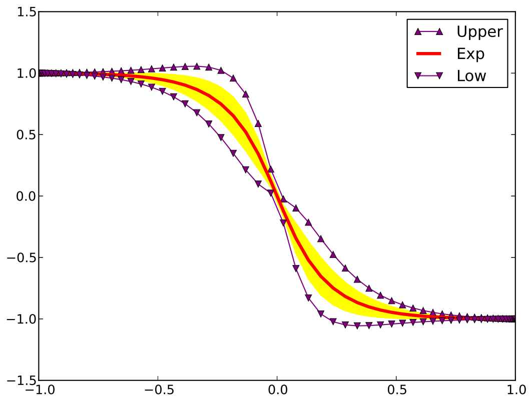


Figure 5.7: The solution to Burgers' equation for $N = 60$ and v $U(0.05, 0.2)$ modeled by 10 iterations of the Legendre polynomial. The yellow band represents the mean $\mu \pm$ the standard derivation σ, and the purple lines define $\mu \pm 2\sigma$. 95% of the solutions will be inside the purple lines.

Figure 5.7 shows that variation over v affects neither the approach towards the boundaries or the point at which $u = 0$, but rather the slope around $u = 0$. This means that a variation over v changes the slope of the rapid shift. Since we can compare to figure 4.5, where we know what the solution looks like for $v = 0.05$, we can say that the greater v is, the less rapid the shift is. The full code for this problem is in appendix E.2.2

5.4.3.2 Uncertainty on the boundary

Since the v parameter did not influence the Burgers' equation, we will introduce an extra parameter on the left boundary, ϵ, making the boundary condition $u(-1) = 1 + \epsilon$. Since our model already keeps the boundaries as their initial values, all we need to to is modify the initial condition in order to accommodate our new boundary condition. We will adjust the current initial condition, by multiplying all values above zero with $(1 + \epsilon)$, allowing for a smooth transition. We will reuse the construction from the previous part, setting $v = 0.05$, and simply letting ϵ be the dimension of uncertainty. We will assume that epsilon is a gaussian distributed variable $\epsilon \sim N\big(0.1, 0.05^2\big)$.

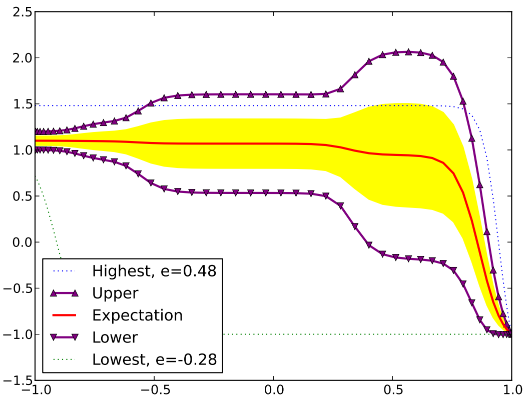


Figure 5.8: The solution to Burgers' equation with a modified boundary of $\epsilon \sim N(0.1, 0.05^2)$ and $v = 0.05$, where ϵ is modeled using 20 iterations of the Hermite polynomials.

Figure 5.8 shows that the variation over ϵ can have a drastic effect on the location of the shift, as seen on the highest and lowest parts. Since $\epsilon \sim N(0.1, 0.05^2)$, the shift of the expectation is located to the right of where we would expect it to for $\epsilon = 0$ from figure 4.5. The expectation and standard deviation is greater in the spectrum where the shift is most likely to occur, indicating that the shift is mostly happening for $x \in (0.25, 0.75)$. The full code for this problem is in appendix E.2.2

We solve this system again, but this time for a uniform distribution $\epsilon \sim U(0, 0.1)$, since we want to limit the shift to moving right from the center. This gives us the results shown in figure 5.9, and show the same tendency to move the shift to the right, and it shows a steady move towards the right from the lowest ϵ to the highest, indicating a linear relationship between ϵ and the location of the shift.

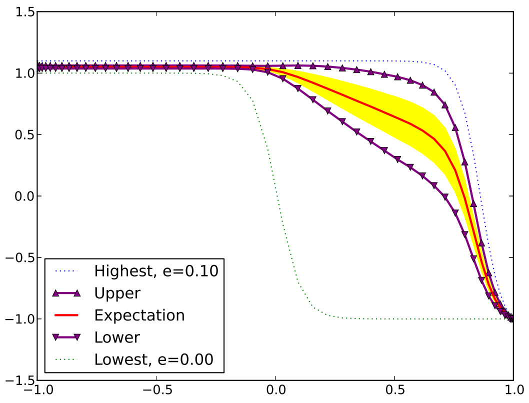


Figure 5.9: The solution to Burgers' equation with a modified boundary of $\epsilon \sim U(0, 0.1)$ and $v = 0.05$, where ϵ is modeled using 20 iterations of the Legendre polynomials.

The full code for this problem is in appendix E.2.2

5.4.4 The test equation – two dimensional uncertainty

We revisit the test equation, introducing multidimensional uncertainty. We will now both let α and β in (5.10) be stochastic variables that follow gaussian distributions $\alpha \sim N\big(4, 1.5^2\big)$ and $\beta \sim N\big(3, 0.15^2\big)$. This requires us to calculate the expectation for the solution with both of these as stochastic variables. We can reuse the expression found in (5.11), since we know that $\mathbb{E}[\beta] = \mu_\beta$, allowing us to calculate the expectation analytically.

For the implementation, depending on the amount of nodes we use for α_N and β_M, the stochastic part of the system will become of size NM, since all values of α will be paired with all values of β. We model this in the same way that we would the 2D spectral problems, we create a mesh of α and β and use these to generate the beginning solutions. We also create a mesh of the nodes associated weights, and find the product of these weights as the weight for that particular instance of the solution. Using this method, we can reuse the `Python` algorithms for calculating multi-dimensional spectral problems for multidimensional stochastic problems.

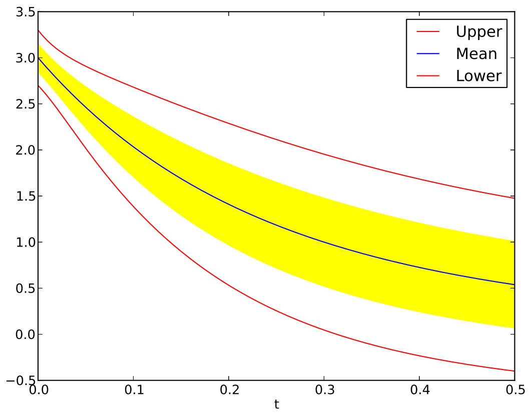


Figure 5.10: The solution to 2-dimensional uncertainty applied to the test equation.

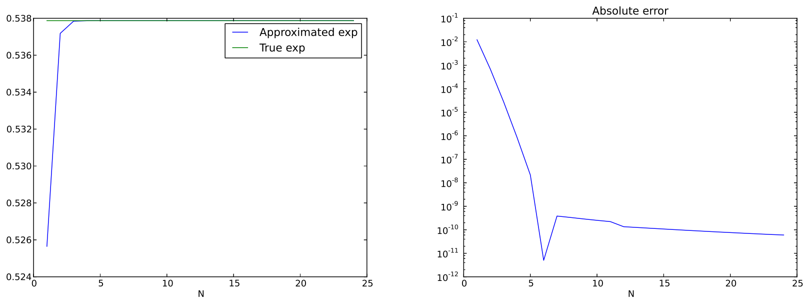


Figure 5.11: The approximated expectation and the error of the expectation as a function of the size of our samples.

We see from figure 5.11 that we conserve the fast spectral convergence for multiple dimensions of uncertainty, and we can see in figure 5.10 that in the start the variation seems to be controlled by the β uncertainty, but as t gets bigger, the variation rises due to the great effect of negative values of α.

The entire code for this problem can be found in appendix E.2.1.

While we will not be using this method from this point on, it is worth noting that the stochastic collocation method is easily expanded to increase the number of random parameters.

Chapter 6

Numerical experiments

In this chapter we will focus on pseudo-practical applications of the uncertainty quantification and spectral methods developed in this book. We will concern ourselves with two problems, a lid driven cavity problem, and a wave model simulator.

6.1 Lid driven cavity

The lid driven cavity problem aims to describe the flow in a container, where the lid is driving the flow in a certain direction, as if to simulate the flow within a cavity where a steady stream is passing over. The lid driven cavity problem comes from the non-dimensional steady-state Navier-Stokes equations for incompressible flow as described in [WZ10], which we will be employing on a physical domain $\Omega = [0,1] \times [0,1]$..

$$\nabla \cdot \mathbf{u} = 0 \tag{6.1}$$

$$(\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + Re^{-1}\nabla^2 \mathbf{u} \tag{6.2}$$

Here each point is represented by the velocity $\mathbf{u} = (u, v)$, where u and v are the velocity components in the respectively x- and y-direction. Each point also has a value of the pressure p associated with it, but only on the interior nodes, in order to eliminate the need for boundary conditions and the possibility of pollution from nonphysical modes [WZ10]. *Re* is the Reynolds number, which is defined as $V_{ch}d/\nu$, which is found by choosing the characteristic velocity V_{ch} – or the speed of the lid – and length d, with ν being the kinematic viscosity.

The problem will be formulated with Dirichlet boundaries, where $(u, v) = (0, 0)$ on all boundaries for the top boundary, where $u = 1$.

In order to achieve the steady state, we will introduce pseudo-time variables τ, which will detail the initial-value-problem we will ned to step through.

$$\frac{1}{\beta^2}\frac{\partial p}{\partial \tau} + \nabla \cdot \mathbf{u} = 0 \tag{6.3}$$

$$\frac{\partial \mathbf{u}}{\partial \tau} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + Re^{-1}\nabla^2 \mathbf{u} \tag{6.4}$$

Where $\beta^2 = 5$ is chosen in [WZ10], and since this will not affect the final result, we will choose this as well.

With the model we will seek to examine the influence of uncertainty in the lid-speed, represented through the Reynolds number, and how the stream in the cavity is effected by this.

6.1.1 Derivation of the spectral model

We will be employing the Legendre spectral collocation method to solve this initial-value problem. Since we are employing Dirichlet boundaries, we will be using Legendre polynomials to model both dimensions, both for the full (u, v) grid, as with the inner p grid.

6.1.1.1 Scaling the working grid

This means that with the original Legendre grid $x_L, y_L \in [-1, 1]$ we will need to employ the transformation to $x_N, y_N \in [0, 1]$.

$$x_N = \frac{x_L + 1}{2} \qquad\qquad y_N = \frac{y_L + 1}{2}$$

Which gives us the scaling factors of

$$\frac{dx_L}{dx_N} = 2 \qquad\qquad \frac{dy_L}{dy_N} = 2 \tag{6.5}$$

Since the pressure p is also modeled by Legendre polynomials, but is connected to the inner grid of points x_i, y_i for $1 \leq i \leq N-1$, we will need to scale these points as well to $\tilde{x}_{N-2}, \tilde{y}_{N-2} \in [-1, 1]$. This is easily done with

$$\tilde{x}_i = \frac{x_i}{x_c} \qquad \tilde{y}_i = \frac{y_i}{y_c} \qquad \text{for } 1 \leq i \leq N-1 \wedge c = N-1 \tag{6.6}$$

Which will give us the secondary[1] scaling factor of

$$\frac{dx_i}{d\tilde{x}_i} = x_c \qquad \frac{dy_i}{d\tilde{y}_i} = y_c \qquad \text{for } 1 \leq i \leq N-1 \wedge c = N-1 \tag{6.7}$$

6.1.1.2 Using the collocation approximation

Now all we need to do is define the equations (6.3) and (6.4) in the collocation approximation. Since $\mathbf{u} = (u, v)$, we can write the divergence as

$$\nabla \cdot \mathbf{u} = \begin{pmatrix} \frac{d}{dx} \\ \frac{d}{dy} \end{pmatrix} \cdot \begin{pmatrix} u \\ v \end{pmatrix} = \frac{du}{dx} + \frac{dv}{dx}$$

Which means that we can approximate this using the collocation differentiation operator

$$\nabla \cdot \mathbf{u}_N = D_x u_N + D_y v_N$$

This will allow us to formulate the continuity equation (6.3) from the system as

$$\begin{aligned} \frac{1}{\beta^2}\frac{\partial p_N}{\partial \tau} + \nabla \cdot \mathbf{u}_N = 0 \Leftrightarrow \\ \frac{\partial p_N}{\partial \tau} = -\beta^2(\nabla \cdot \mathbf{u}_N) \Leftrightarrow \\ \frac{\partial p_N}{\partial \tau} = -\beta^2(D_x u_N + D_y v_N) \end{aligned} \tag{6.8}$$

In order to approximate the momentum equation (6.4), we will need to calculate the Laplacian $\nabla^2\mathbf{u}$ and the convection $(\mathbf{u} \cdot \nabla)\mathbf{u}$. The Laplacian will be

$$\begin{aligned} \nabla^2\mathbf{u} &= \left(\begin{pmatrix} \frac{d}{dx} \\ \frac{d}{dy} \end{pmatrix} \cdot \begin{pmatrix} \frac{d}{dx} \\ \frac{d}{dy} \end{pmatrix} \right) \begin{pmatrix} u \\ v \end{pmatrix} \Leftrightarrow \\ \nabla^2\mathbf{u} &= \left(\frac{d^2}{dx^2} + \frac{d^2}{dy^2} \right) \begin{pmatrix} u \\ v \end{pmatrix} \Leftrightarrow \\ \nabla^2\mathbf{u} &= \begin{pmatrix} \frac{d^2u}{dx^2} + \frac{d^2u}{dv^2} \\ \frac{d^2v}{dx^2} + \frac{d^2v}{dv^2} \end{pmatrix} \end{aligned} \tag{6.9}$$

[1] Since we will also need to apply the scaling factor from (6.5)

And the convection

$$
\begin{aligned}
(\mathbf{u}\cdot\nabla)\mathbf{u} &= \left(\begin{pmatrix} u \\ v \end{pmatrix} \cdot \begin{pmatrix} \frac{d}{dx} \\ \frac{d}{dy} \end{pmatrix}\right)\begin{pmatrix} u \\ v \end{pmatrix} \Leftrightarrow \\
(\mathbf{u}\cdot\nabla)\mathbf{u} &= \left(u\frac{d}{dx} + v\frac{d}{dy}\right)\begin{pmatrix} u \\ v \end{pmatrix} \Leftrightarrow \\
(\mathbf{u}\cdot\nabla)\mathbf{u} &= \begin{pmatrix} u\frac{du}{dx} + v\frac{du}{dy} \\ u\frac{dv}{dx} + v\frac{dv}{dy} \end{pmatrix} \qquad (6.10)
\end{aligned}
$$

(6.9) and (6.10) allows us to formulate the momentum equation for our model

$$
\begin{aligned}
&\frac{\partial\mathbf{u}}{\partial\tau} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla p + Re^{-1}\nabla^2\mathbf{u} \Leftrightarrow \\
&\frac{\partial\mathbf{u}}{\partial\tau} = -(\mathbf{u}\cdot\nabla)\mathbf{u} - \nabla p + Re^{-1}\nabla^2\mathbf{u} \Leftrightarrow \\
&\frac{\partial\mathbf{u}}{\partial\tau} = -\begin{pmatrix} u\frac{du}{dx} + v\frac{du}{dy} \\ u\frac{dv}{dx} + v\frac{dv}{dy} \end{pmatrix} - \begin{pmatrix} \frac{dp}{dx} \\ \frac{dp}{dy} \end{pmatrix} + \begin{pmatrix} \frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} \\ \frac{d^2v}{dx^2} + \frac{d^2v}{dy^2} \end{pmatrix}
\end{aligned}
$$

Which we can also approximate using the collocation approximation

$$
\frac{du}{d\tau} = -(uD_xu + vD_yu) - \frac{dp_{N-2}}{dx} + D_x^2u + D_y^2u \qquad (6.11)
$$

$$
\frac{dv}{d\tau} = -(uD_xv + vD_yv) - \frac{dp_{N-2}}{dy} + D_x^2v + D_y^2v \qquad (6.12)
$$

The reason why we have not used a differential operator to approximate ∇p is that p is only defined on the inner grid, and thus approximated by p_{N-2}. We will need to differentiate this on the inner grid, and then interpolate the values to the full grid.

6.1.1.3 Interpolating the pressure

The pressure will be modeled by the inner grid with p_{N-2}. Since we calculate the τ differential in (6.8), we will simply only apply the inner-grid values of this differential, and disregard the outer grid completely. For the equations (6.11) and (6.12) however, we will need the space derivative of the pressure for application on the full grid – and we will need to interpolate the values from the inner grid to the outer grid for this.

We will be using the Vandermonde matrix as defined in (3.6), since it can transform the nodal values to the modal values. Once we have the modal values, we can interpolate these to the outer grid nodal values using another Vandermonde matrix. The second Vandermonde matrix will need to be defined on the inner grid extrapolated to the outer values, which means that we will have to reverse the scaling, giving us

$$\tilde{x}_N = x_N x_c \qquad \tilde{y}_N = y_N y_c \qquad c = N-1 \tag{6.13}$$

This allows us to interpolate the pressure using the functions

$$\begin{aligned} p_{N-2} &= V_1 \hat{p}_{N-2} \Leftrightarrow \hat{p}_{N-2} = V_1^{-1} p_{N-2} \\ \tilde{p}_N &= V_2 \hat{p}_{N-2} \end{aligned}$$

This requires the p values to be allocated into a vector, with the Vandermonde matrix V_1 being of size $(N-2)^2 \times (N-2)^2$, while V_2 will be of size $N^2 \times (N-2)^2$.

In order to calculate these Vandermonde matrices, we will use the same method as we used for the differential operators in section 4.3.3.2, and create the Vandermonde for the x-direction and the y-direction independently using the points x_{N-2} and y_{N-2}, then creating $V_1 = V_{x_{N-2}} \otimes V_{y_{N-2}}$. For the second Vandermonde matrix, we will be creating the Vandermonde matrices using the points $\tilde{x}_N$ and $\tilde{y}_N$, and then creating $V_2 = V_{\tilde{x}_N} \otimes V_{\tilde{y}_N}$.

This allows us to define the interpolation as

$$\nabla \tilde{p}_N = V_2 V_1^{-1} \nabla p_{N-2}$$

Where ∇p_{N-2} is found with the differential operator defined on the inner grid.

6.1.1.4 Applying the boundary conditions

Since we are working with nodal points and a collocation approximation, we will simply ensure that the points on the boundary fulfill the boundary conditions, this being that $(u, v) = (0, 0)$ on all boundaries, except where $y = 1$, where the boundary is $(u, v) = (1, 0)$. The pressure has no boundary conditions as it is only defined on the inner grid.

6.1.1.5 The τ-steps

For the time stepping procedure we will be using the explicit four-stage Runge-Kutta method, as it is used in [WZ10], where the time step to guarantee convergence is supplied. This means that we will be using the following method

$$\begin{aligned}
\phi^{(1)} &= \phi^n + \frac{1}{4}\Delta\tau R(\phi^n) \\
\phi^{(2)} &= \phi^n + \frac{1}{3}\Delta\tau R\Big(\phi^{(1)}\Big) \\
\phi^{(3)} &= \phi^n + \frac{1}{2}\Delta\tau R\Big(\phi^{(2)}\Big) \\
\phi^{(n+1)} &= \phi^n + \Delta\tau R\Big(\phi^{(3)}\Big)
\end{aligned} \tag{6.14}$$

Where the $R(\bullet)$ is the right-hand-side function, which is defined independently for u, v and p from (6.11), (6.11) and (6.8).

The pseudo-time steps are defined by

$$\Delta\tau = \frac{\text{CFL}}{\lambda_x + \lambda_y} \tag{6.15}$$

Where CFL is the Courant-Friedrichs-Lewy's number, which we will set to 0.5, and the λs are the limits contributed by the Navier-Stokes equations, which are calculated as

$$\begin{aligned}
\lambda_x &= \frac{|u_{\text{max}}| + \sqrt{u_{\text{max}}^2 + \beta^2}}{\Delta x} + \frac{1}{Re \cdot \Delta x^2} \\
\lambda_y &= \frac{|v_{\text{max}}| + \sqrt{v_{\text{max}}^2 + \beta^2}}{\Delta y} + \frac{1}{Re \cdot \Delta y^2}
\end{aligned}$$

Where Δx and Δy are the minimum collocation spacing on their respective grids.

6.1.2 Implementation of the spectral model

The implementation of the spectral model is necessary to examine the influence of the Reynolds-number, and we will develop a model for $Re = 100$, where we have results to compare against from [WZ10].

6.1.2.1 Generation of points and differential matrices

Since we will be using Legendre polynomials to model the data, we will be generating the points for the outer grid with the function `GaussLobattoQuadrature` from `DABISpectral1D`. One these points are generated, we will calculate the scaling factor to the inner grid, and create the points for the inner grid using (6.6). We create meshes for both grids, and order them into vectors while calculating an index grid as detailed in section 4.3.3.1.

Generating the differential operators is done with algoritm 2, but when creating the operators for the inner grid, we use the modified inner points for the Vandermonde matrices instead of newly generated Legendre-Gauss-Lobatto points. We will also apply the scaling factors from (6.5) and (6.7) to the differentiation operators directly

The generation of the inner-grid differentiation matrices is done for x_p and y_p being the inner points

```
Vxp = LegPol.GradVandermonde1D(xp,Nx-2,0)
Vyp = LegPol.GradVandermonde1D(yp,Ny-2,0)

V = np.kron(Vxp,Vyp)

VxpD = LegPol.GradVandermonde1D(xp,Nx-2,1)
VypD = LegPol.GradVandermonde1D(yp,Ny-2,1)


Dxp = np.linalg.solve(Vxp.T,VxpD.T).T
Dyp = np.linalg.solve(Vyp.T,VypD.T).T

Dxp = 1/cx*2*Dxp
Dyp = 1/cy*2*Dyp
```

We will test these differential matrices while simultaneously testing the index matrices, where we will generate the $f_{\text{test}}(x,y) = \cos(x)\sin(y)$ for both the inner and outer grid, and approximate $\frac{d}{dx}\frac{d}{dy}f_{\text{test}}(x,y)$ for which the solution is $\frac{df_{\text{test}}(x,y)}{dxy} = -\sin(x)\cos(y)$. We will use the differential operators to obtain the solution from the original solution, and use the index matrix to reshape the solution to a matrix.

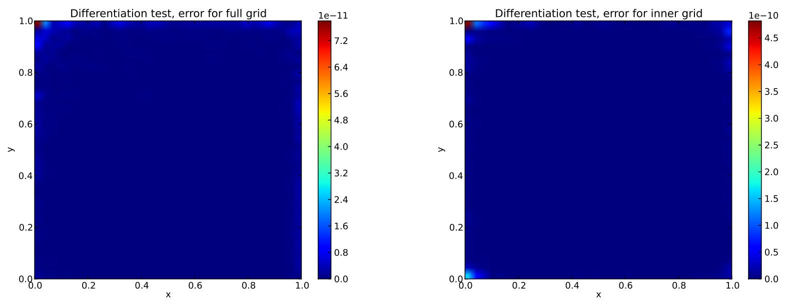


Figure 6.1: The errors of our differential approximation test, the full grid to the left, and the inner grid to the right

As we can see from figure 6.1, the errors for both test cases are acceptable, and we can consider these methods implemented correctly. The full script for testing the errors is featured in the approximation script in appendix F.1.

6.1.2.2 Interpolation of points from the inner grid

The generation of the matrices to interpolate from the inner grid to the outer grid will rely on Vandermonde matrices generated by the function `GradVandermonde` from `DABISpectral1D`. The matrix V_1 is calculated as a standard Vandermonde matrix for the points on the inner grid, while the matrix V_2 is generated as a standard Vandermonde matrix for the inner grid, but with the scaled points from (6.13).

In order to test the interpolation, we create the grid of a function e^{xy}, which we interpolate from the inner grid to the outer grid, and compare to the correct values on the outer grid.

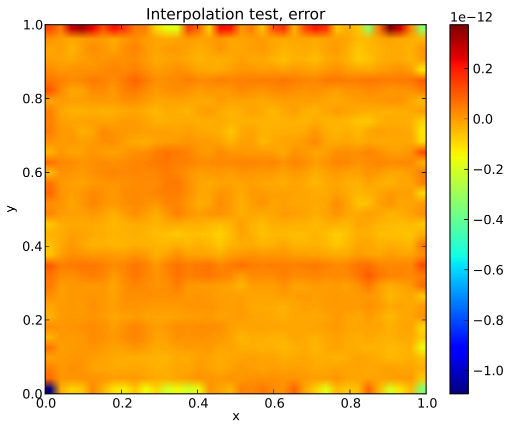


Figure 6.2: The error of the interpolation to the full grid.

As we see from figure 6.2, the error is largest along the boundary, as would be expected, but the error is still very small, which affirms that the method is implemented correctly. The full script for testing the errors is featured in the approximation script in appendix F.1.

6.1.2.3 Pseudo-time step integrator

Even though we will not be using `scipy.integrate.odeint` to do the time-stepping, we will use the same structure as this solution, as it will simplify the implementation of the Runge-Kutta method to a single right-hand-side function.

Right-hand-side The right-hand-side function is constructed such that it calculates all the derivative values, and interpolates the inner derivatives of the pressure to the outer grid first. Afterwards it assembles the different τ-derivatives, and applies the boundary conditions for u and v, and removes the boundary completely for p. Since the function is build up around the same principle as for the input function to `scipy.integrate.odeint`, the right-hand-side function features both assembly and disassembly routines, to split and join all the values into one vector.

The implementation of the time stepping method is exactly as described in (6.14), where the Runge-Kutta steps are simply performed on a single vector with all the collected data. We recalculate the size of the time-steps according to (6.15). Since we are seeking a steady-state, we will be setting a high end-time $\tau_{\text{end}} = 200$, we will include a test, to see if the difference between the newly computed solution and the previous solution – in the measuring data u – is within a certain tolerance of e^{-6}.

6.1.2.4 Testing the model

We will initially test the model with a end-point of $\tau = 2$, which will give us a notion if the model is working correctly.

We will make a stream plot, where we unfortunately required to conform our data to an equidistant grid, which we do by transformation to modal values and back to new nodal values by Vandermonde matrix.

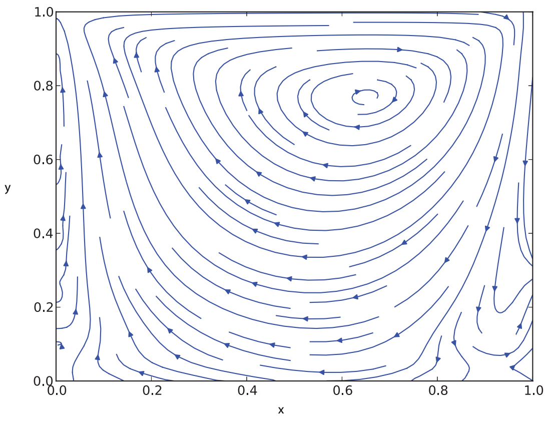


Figure 6.3: The stream plot of (u, v) for our test run to $\tau = 2$.

Figure 6.3 shows that a flow develops in our model, as expected. Since this is not the steady-state, we cannot say anything about the actual flow, but we can clearly see that a proper flow develops, which confirms that our model is correct.

While the code that calculates the values is located in appendix F.1, the code that shows these values, and computes the transformation to an equidistant grid is shown in appendix F.2.

6.1.3 Introducing uncertainty on the Reynolds number

We wish to introduce uncertainty in the surface speed, which will effectively change the Reynolds number, and investigate how the values affects our computations. This is done by the stochastic collocation method, where we can simply run the model several times, for the calculated values of our new Reynolds numbers, where we will compare the results to the velocity profile of u along the middle of the cavity in the y-axis direction. Since the problem is normalized to a speed of $u = 1$ along the upper boundary, this will not change.

We will use a modified solver, which simply runs the problem several times with different Reynolds numbers, and saves each output. Each run is assigned a weight according to the GPC polynomial, and mean and variance are found using the different weighted values.

We do a test run for $Re \sim N\big(100, 10^2\big)$, and let it run to $\tau = 1$, just to see if we are able to quantify the change in the values.

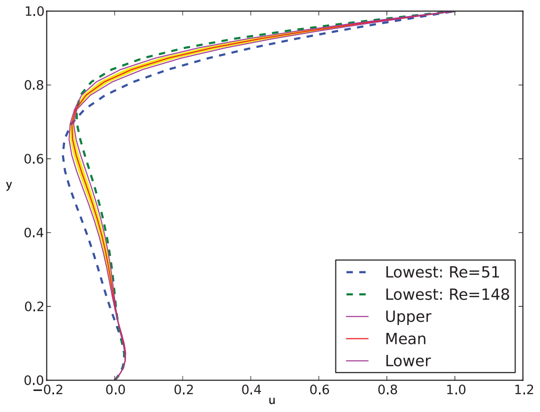


Figure 6.4: The test for our uncertainty quantification run of the lid driven cavity problem

Figure 6.4 shows us that while the uncertainty in not very pronounced, that there is a difference between the highest and lowest values of Re, but that in general the solution will not change much for $Re \sim N\big(100, 10^2\big)$. The code for this run can be found in appendix F.1.

6.1.4 Numerical experiments with UQ on the Reynolds number

We have results from [UG82], for the velocity profile of u down the middle of the cavity at certain points for $Re = [100, 400, 1000]$, which we will take as a measure for the correctness of our model.

We will start by running from the initial condition $(u, v, p) = (0, 0, 1)$ on the inner grid, and approximating the results from [UG82] with the same Reynolds numbers. This will serve as an initial condition for our uncertainty calculations, since it will likely be closer to the solution, saving computational time.

For our tests, since we are measuring against the velocity u, we will stop at iteration j, once our metric $M_j = \frac{\|\mathbf{u}_j - \mathbf{u}_{j-1}\|_2}{\|\mathbf{u}_{j-1}\|_2}$ becomes less than 10^{-6}, which we deem as a steady-state.

The uncertainty we will introduce will be that we have up to 5% uncertainty in the speed of the lid for our problems – and thus a 5% uncertainty in our Reynolds number – allowing our model to quantify this uncertainty. Since we will not be allowing more or less uncertainty, we will assume that Re is uniformly distributed, and modeled using Legendre polynomials.

We will repeat the tests for uncertainty up to 10%, allowing us to study in which degree the increase in uncertainty has an impact on the propagating uncertainty.

All the scripts used in these simulations are either in appendix F.1 or appendix F.2.

6.1.4.1 Uncertainty at $Re = 100$

The initial results for the approximation of the results from [UG82], we get the solution shown in figure 6.5.

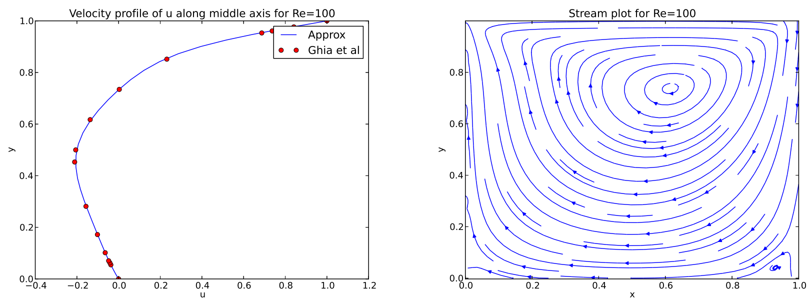


Figure 6.5: The steady state results compared to the results of [UG82], for $Re = 100$ to the left. The stream-plot for the steady state solution is shown to the right.

Figure 6.5 shows us that $Re = 100$ gives us a rather smooth curve for the velocity profile, with the stream-plot suggests that the case of $Re = 100$ is a steady stream where a lot of the liquid is not moving very fast.

To allow for a 5% uncertainty, we will model $Re \sim U(95, 105)$. We will be modeling it with $N_{UQ} = 10$.

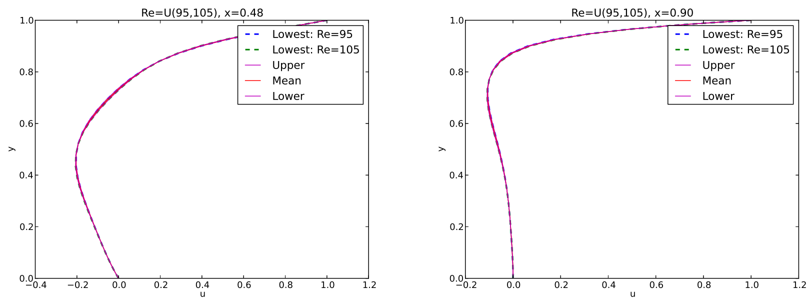


Figure 6.6: The uncertainty on the velocity profile for the Lid Driven Cavity problem with $Re \sim U(95, 105)$. The profile is shown in the middle of the cavity (left picture) and near the right boundary of the cavity (right picture).

The uncertainty of the velocity profile shown in figure 6.6 shows us that a 5% uncertainty does not change the velocity profile significantly for $Re = 100$. It does not change the profile of the sides of the cavity either, which are relatively uneventful for $Re = 100$, as expected from figure 6.5.

We repeat the computations for $Re \sim U(90, 110)$, which gives the results in figure 6.7.

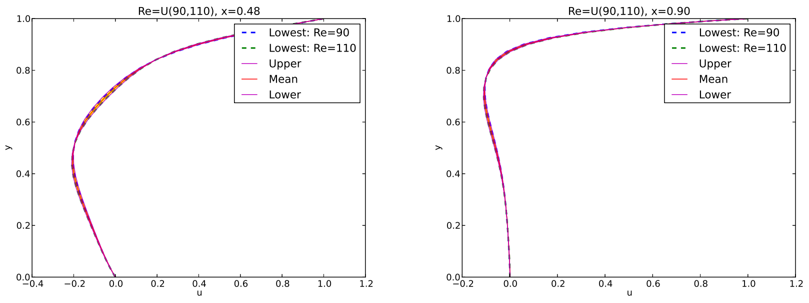


Figure 6.7: The uncertainty on the velocity profile for the Lid Driven Cavity problem with $Re \sim U(90, 110)$. The profile is shown in the middle of the cavity and near the right boundary of the cavity.

Figure 6.7 shows still a very small amount of uncertainty, however it can be seen on the velocity profile for the middle of the cavity. This suggests where in the flow the water will change if Re is changed, but the flow is still fairly stable with 10% uncertainty in the Reynolds number.

6.1.4.2 Uncertainty at $Re = 400$

The initial results for this system is calculated and compared to the results from [UG82] again.

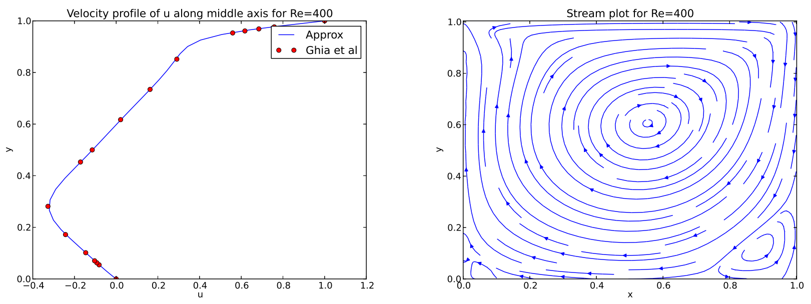


Figure 6.8: The steady state results compared to the results of [UG82], for $Re = 400$ to the left. The stream-plot for the steady state solution is shown to the right.

We see in figure 6.8 that our approximation matches the results from [UG82], and that the velocity profile is significantly changed from figure 6.5. We see in the stream-plot that recirculation is starting in the corner where the water comes down from the lid.

We allow for 5% uncertainty, giving us $Re \sim U(380, 420)$.

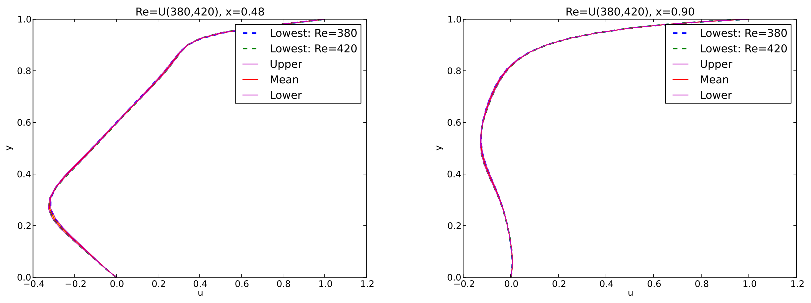


Figure 6.9: The uncertainty on the velocity profile for the Lid Driven Cavity problem with $Re \sim U(380, 420)$. The profile is shown in the middle of the cavity and near the right boundary of the cavity.

Figure 6.9 shows us that even with a faster flow, the model is still robust towards uncertainty. We see minor variations in the area where velocity changes from rising to falling in both plots. We also see that recirculation towards the bottom of the right plot is relatively robust towards the variations, as the speed in the recirculation is not that great.

We model this case with 10% uncertainty as well, giving us $Re \sim U(360, 440)$.

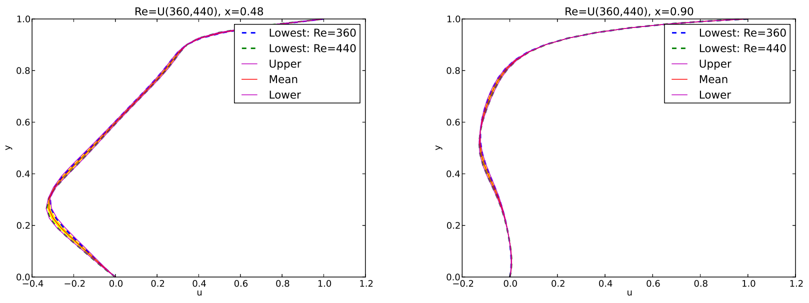


Figure 6.10: The uncertainty on the velocity profile for the Lid Driven Cavity problem with $Re \sim U(360, 440)$. The profile is shown in the middle of the cavity and near the right boundary of the cavity.

Figure 6.10 shows us that the variations are not at all situated where the velocity changes sign, but rather around the axis where the velocity is zero – the speed at which the water is moving around the point where horizontal movement is null, near the "swirl" shown in figure 6.8. The recirculation is still very stable towards the uncertainty, while the uncertainty seems to change the speed at which the values in the right part of the cavity approach the largest negative value of u, while not affecting the size of the the largest negative value.

6.1.4.3 Uncertainty at $Re = 1000$

The initial results of this simulation is compared to the results from [UG82] as the two cases before have been.

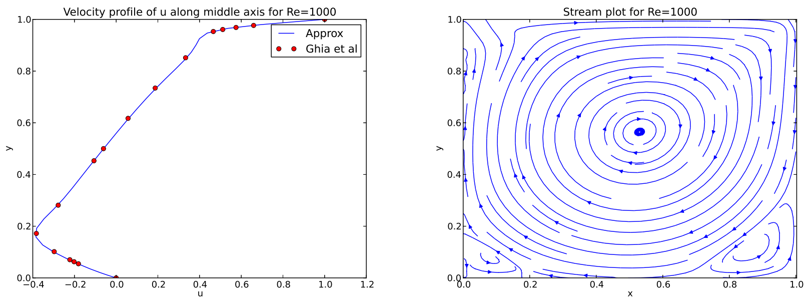


Figure 6.11: The steady state results compared to the results of [UG82], for $Re = 1000$ to the left. The stream-plot for the steady state solution is shown to the right.

We see in figure 6.11 that the system is now having a very well established flow. We also notice that recirculation has developed in both lower corners of the cavity. We can also verify that we have achieved the same results as [UG82]

We introduce 5% uncertainty, giving us $Re \sim U(950, 1050)$.

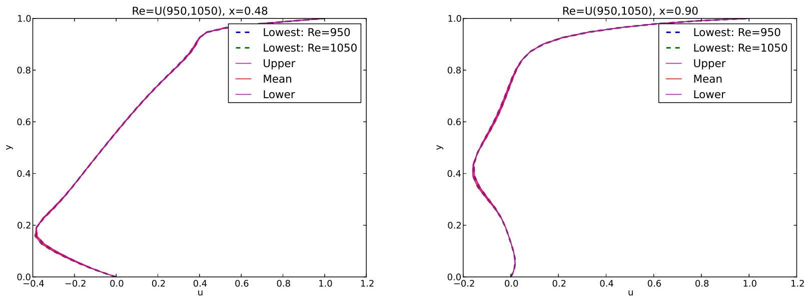


Figure 6.12: The uncertainty on the velocity profile for the Lid Driven Cavity problem with $Re \sim U(950, 1050)$. The profile is shown in the middle of the cavity and near the right boundary of the cavity.

Figure 6.12 shows us that with this well established flow, the model is very robust towards uncertainty, both along the center of the cavity, and along the edges and in the recirculation zone.

We increase the uncertainty to 10%, giving us $Re \sim U(900, 1100)$.

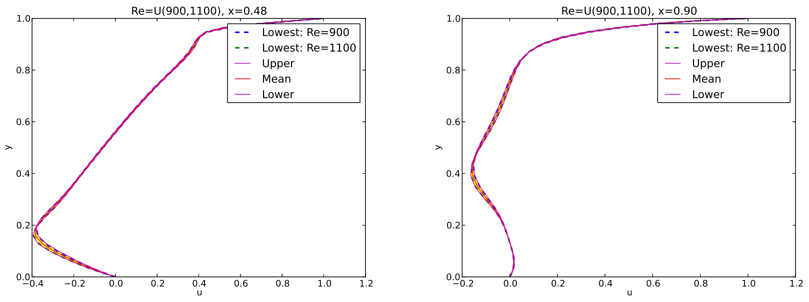


Figure 6.13: The uncertainty on the velocity profile for the Lid Driven Cavity problem with $Re \sim U(900, 1100)$. The profile is shown in the middle of the cavity and near the right boundary of the cavity.

Figure 6.13 shows that the uncertainty mainly focuses around the areas with the greatest flow towards the left of the cavity, particularly the parts lower than these points. This indicates that the uncertainty at $Re = 1000$ will affect how deep the flow manages to manifest, towards the unmoving bottom of the cavity. We again notice that even though the recirculation zone is more pronounced that for $Re = 400$, it is still very stable towards uncertainty, indicating that this zone is not affected by the uncertainty, but driven by the overall unchanging flow.

6.1.4.4 Uncertainty at $Re = 1$

For $Re = 1$ we have no results to compare with, but run the test-case under the same conditions that applied for the other test-cases.

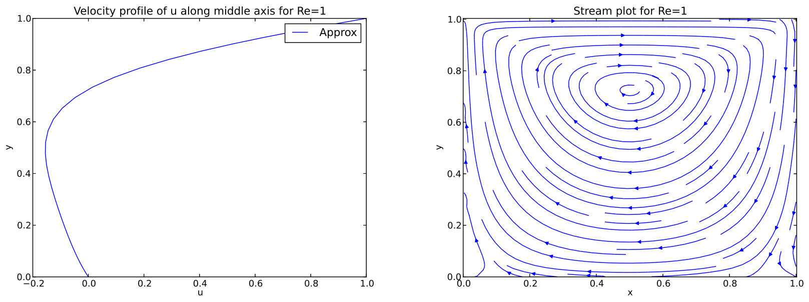


Figure 6.14: The steady state results for $Re = 1$ to the left. The stream-plot for the steady state solution is shown to the right.

Figure 6.14 shows us that for $Re = 1$, the stream is very uniform, which is to be expected for a very slow moving lid. We introduce the 5% uncertainty to this domain as well, giving us $Re \sim U(0.95, 1.05)$.

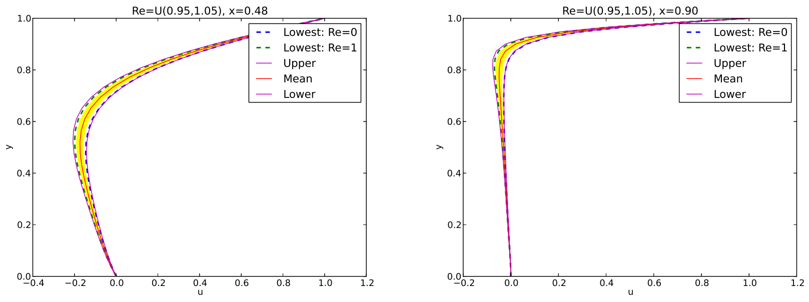


Figure 6.15: The uncertainty on the velocity profile for the Lid Driven Cavity problem with $Re \sim U(0.95, 1.05)$. The profile is shown in the middle of the cavity and near the right boundary of the cavity.

We see from figure 6.15 that this model is significantly more sensitive to small changes that the previous models. As it is a stable system captured from a very slow lid, the small changes in the lid speed will change the entire composition of the flow, as indicated by figure 6.15 – the uncertainty is centralized around where the flow changes direction rapidly, and is evenly distributed over these areas.

We introduce 10% uncertainty on this problem as well, giving us $Re \sim U(0.90, 1.10)$.

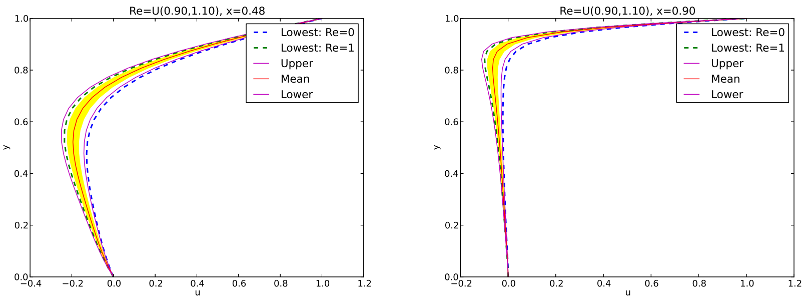


Figure 6.16: The uncertainty on the velocity profile for the Lid Driven Cavity problem with $Re \sim U(0.90, 1.10)$. The profile is shown in the middle of the cavity and near the right boundary of the cavity.

Figure 6.16 confirms what we saw in figure 6.15, but simply expanding the areas where we have uncertainty. There are no significant changes otherwise.

6.1.5 Conclusions for the lid driven cavity flow model

The numerical experiments from the lid driven cavity problem shows us that uncertainty in the input parameter can have drastically different manifestations dependent on which state the problem is in. This means that in order to effectively quantify the uncertainty of a system, we will have to do active uncertainty quantification on that system, and not simply relate to another similar problem. This enhances the strength of the generalized polynomial chaos approach, as this enables us to effectively calculate the mean and standard deviation. With a Monte-Carlo approach to this problem would we not only be unable to utilize the last computed solution as a start in order to save computational time – the amount of times we would need to run the model would likely be unfeasibly high. This would discourage the need to recompute the mean and variance for a new state of the system, and might lead to inconclusive results drawn from a similar problem instead of computed correctly.

6.2 A nonlinear 2D wave tank model

This problem is inspired by [EK11b], and we wish to model the movement of the free surface, as defined by the free surface elevation η, and the surface scalar velocity potential $\tilde{\phi}$. The model we use is a two-dimensional wave tank, illustrated in figure 6.17, and

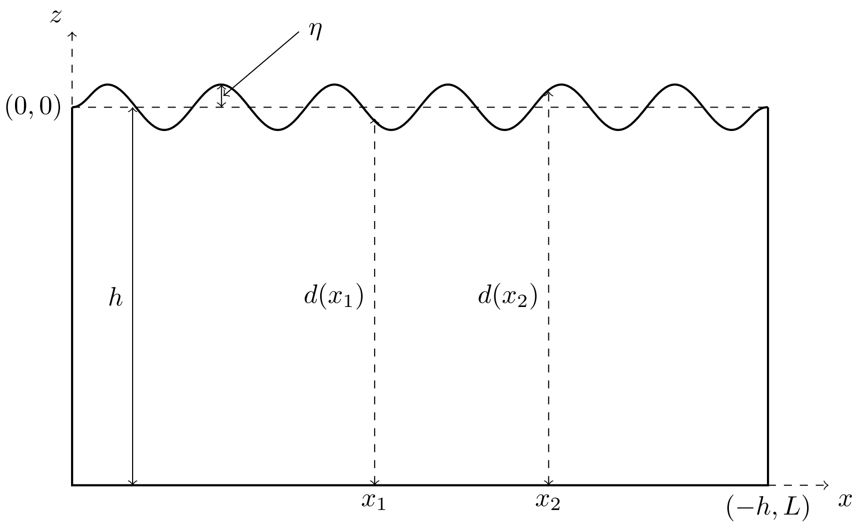


Figure 6.17: The 2D wave tank

[EK11b] describes the free surface to be governed by kinematic boundary (6.16) and dynamic boundary (6.17)

$$\partial_t \eta = -\partial_x \eta \partial_x \tilde{\phi} + \tilde{w}\left(1 + (\partial_x \eta)^2\right) \tag{6.16}$$

$$\partial_t \tilde{\phi} = -g\eta - \frac{1}{2}\left(\left(\partial_x \tilde{\phi}\right)^2 - \tilde{w}^2\left(1 + (\partial_x \eta)^2\right)\right) \tag{6.17}$$

Where g is the gravitational acceleration, and $\tilde{w}$ is the vertical velocity at the surface.

Potential flow theory represents the velocity field as $\begin{pmatrix} u \\ w \end{pmatrix} = \begin{pmatrix} \partial_x \\ \partial_w \end{pmatrix} \phi$, which we can find by solving the two-dimensional Laplace problem

$$\partial_{xx}\phi + \partial_{zz}\phi = 0 \qquad -h \leq z < \eta \tag{6.18}$$

$$\phi = \tilde{\phi} \qquad z = \eta \tag{6.19}$$

$$\begin{pmatrix} n_x \\ n_z \end{pmatrix} \cdot \begin{pmatrix} \partial_x \\ \partial_z \end{pmatrix} \phi = 0 \qquad (x,z) \in \delta\Omega \tag{6.20}$$

The Dirichlet boundary (6.19) ensures that our Laplacian problem is persistent with the free surface, and the Neumann boundaries (6.20) are ensuring that the walls of the tank are are modeled as solid.

We want to examine the effects the size of a wave have in relation to how much force the walls at the boundary are affected by. This can be used in gauging the amount of force a structure is required to withstand, which can help optimizing the design and development process of such a structure.

6.2.1 Derivation of the spectral model

For the computational problem, we will seek to transform the domain to a square domain, which we can approximate by Legendre polynomials. We will seek to do this by transforming the uneven vertical dimension to a fixed computational domain

$$\sigma = \frac{z+h}{d(x,t)} \qquad d(x,t) = \eta(x,t) + h \tag{6.21}$$

(6.21) enables [EK11b] to rewrite the Laplace problem (6.18) - (6.20) to the time-independent system for $\Phi(x,\sigma) = \phi(x,z,t)$

$$\partial_{xx}\Phi + \partial_{xx}\sigma(\partial_\sigma\Phi) + 2\partial_x\sigma\partial_{x\sigma}\Phi + \Big((\partial_x\sigma)^2 + (\partial_z\sigma)^2\Big)\partial_{\sigma\sigma}\Phi = 0 \quad 0 \le \sigma < 1 \tag{6.22}$$

$$\Phi = \tilde{\phi} \quad \sigma = 1 \tag{6.23}$$

$$\begin{pmatrix} n_x \\ n_z \end{pmatrix} \cdot \begin{pmatrix} \partial_x \\ \partial_z\sigma\partial_\sigma \end{pmatrix} \Phi = 0 \quad (x,\sigma) \in \delta\Omega \tag{6.24}$$

Where the time-dependent coefficients can be calculated for a specific time at

$$\begin{aligned} \partial_x\sigma &= -\frac{\sigma}{d}\partial_x\eta \\ \partial_{xx}\sigma &= -\frac{\sigma}{d}\left(\partial_{xx}\eta - \frac{(\partial_x\eta)^2}{d}\right) - \frac{\partial_x\sigma}{d}(\partial_x\eta) \\ \partial_z\sigma &= \frac{1}{d} \end{aligned} \tag{6.25}$$

This enables us for a initial condition of $\left(\eta, \tilde{\phi}\right)$, we can calculate the velocity field, which allows us to calculate the time-derivatives of $\left(\eta, \tilde{\phi}\right)$. This allows us to build a complete initial value problem for the free surface movement of the water in our two-dimensional tank.

6.2.1.1 Deriving the collocation approximation

We will be using the collocation approximation with the Legendre polynomials in both directions, allowing us to utilize the differential operators calculated in section 3.1.2. We will be using these as single-dimensional operators in the calculation of the time-dependent coefficients (6.25), which are calculated individually for each point in our domain.

$$\begin{aligned}
\partial_x \sigma(\mathbf{x},\sigma) &= -\frac{\sigma}{d(\mathbf{x})} D_x \eta(\mathbf{x}) \\
\partial_{xx} \sigma(\mathbf{x},\sigma) &= -\frac{\sigma}{d(\mathbf{x})}\left(D_x^2 \eta(\mathbf{x}) - \frac{\left(D_x \eta(\mathbf{x})\right)^2}{2} \right) - \frac{D_x \sigma}{d(\mathbf{x})} (D_x \eta(\mathbf{x})) \\
\partial_z \sigma(\mathbf{x},\sigma) &= \frac{1}{d(\mathbf{x})}
\end{aligned} \tag{6.26}$$

Using the differential operators, we can calculate differential operators for use on the whole domain as explained in section 4.3.3.2 – where we will refer to the differential operator in the horizontal direction as D_X and in the vertical direction as D_σ – which leads us to formulate a linear operator for solving (6.22)

$$\mathcal{L} = D_X^2 + \partial_x x \sigma D_\sigma + 2\partial_x \sigma D_X D_\sigma + \left(\left(\partial_x \sigma\right)^2 + \left(\partial_z \sigma\right)^2 \right) D_\sigma^2 \tag{6.27}$$

$$\mathcal{L}\Phi = 0 \quad (x,\sigma) \notin \partial\Omega \tag{6.28}$$

In order to enforce the Dirichlet boundary condition, we change the rows of $\mathcal{L}$ corresponding to the top of the grid, such that the points on the upper boundary are exactly equal to the right-side, for which we introduce a right-hand-side function

$$f_{x,\sigma} = \begin{cases} \tilde{\phi}_x & , \text{ for } \sigma = 1 \\ 0 & , \text{ otherwise} \end{cases}$$

For the Neumann boundary conditions, we modify linear operator, such that differential at the outer points in the respective direction is always zero, which is done by assigning

$$\begin{aligned}
\mathcal{L} &= -D_X && \text{for the horizontal boundaries} \\
\mathcal{L} &= -\partial_z \sigma D_\sigma && \text{for the bottom boundary}
\end{aligned} \tag{6.29}$$

This allows us to calculate the velocity potential for the domain, which we can use to calculate the time-derivatives for the free surface described in (6.16) and

(6.17).

$$\partial_t \eta = -D_x \eta D_x \tilde{\Phi} + \tilde{w}\Big(1 + (D_x \eta)^2\Big)$$
$$\partial_t \tilde{\Phi} = -g\eta - \frac{1}{2}\bigg(\Big(D_x \tilde{\Phi}\Big)^2 - \tilde{w}^2\Big(1 + (D_x \eta)^2\Big)\bigg)$$

Where $\sim$ denotes the values at the top of the grid, and $w = D_\sigma \Phi$.

6.2.1.2 Calculating the force on the wall

The force on the side of the domain can be calculated by an integral along the vertical axis of the pressure, according to [EK06, Eq. 2.84].

$$F = \int_{-d}^{\eta} p(z)\,\mathrm{d}z$$

In order to compute this, we will need to calculate the pressure on the right boundary. This is defined in [EK06, Eq. 2.83] as

$$\frac{p(z)}{\rho} = g(\eta - z) + \int_z^{\eta} \partial_t w\,\mathrm{d}z + \frac{1}{2} + \left(\tilde{u}^2 - u(z)^2 + \tilde{w}^2 - w(z)^2\right)$$

Where ρ is the density of the water, which is $\rho = 0.998$ and $u = D_X \Phi$. The integral $\int_z^{\eta} \partial_t w\,\mathrm{d}z$ requires us to transform w to z points where we can utilize the Legendre-Gauss-Lobatto quadrature, which we do by the method described in section 6.1.1.3, by converting to modal values first, and then transforming to the new nodal values. Since we are modelling the vertical axis by a Legendre polynomial, we can use the Legendre-Gauss-Lobatto quadrature to calculate the force integral, allowing us to compute F. These calculations will require a transformation of the Legendre polynomials from $z_L \in (-1, 1)$ to $z \in [-h, \eta]$, which is

$$z = \frac{z_L + 1}{2}(h + \eta) - h$$

For the time-derivative of w, we will be using a simple first order approximation for the time-derivative.

6.2.2 Implementation of the spectral model

The implementation of the spectral model will consist of three parts, a solver for our Laplace problem, the integrator for the time-dependent problem, and a calculation of the pressure.

6.2.2.1 The solver for the Laplace problem

The implementation to our Laplace problem is largely based on (6.28) and the boundary-modifications to the linear operator and right hand side function. We will construct a grid of Legendre points for both dimensions (with N_x and N_z points) and generate the differential operators from algorithm 2. We will handle the two dimensions by the methods outlined in section 4.3.3, by ordering our mesh into vectors, and creating differential operators based on this transformation.

We calculate the time-dependent coefficients (6.26) from the input, and form these into diagonal matrices, allowing us to assign the correct value to each point, while being able to use the matrix product when constructing $\mathcal{L}$.

We employ the boundary conditions as described in section 6.2.1.1 by simply changing the values of $\mathcal{L}$ for the rows which correspond to boundary points. For the Neumann boundaries, we substitute the linear operator row by the corresponding row to the differential operator as specified in (6.29). For the Dirichlet boundary, we simply set the whole row to zero for all the points but place in the row corresponding to the point itself, where we will place a 1. We then modify the right-hand-side function, which previously contained all zeroes, to contain the desired value for each point.

The boundaries are calculated in the code

```
LBCsigma = np.dot(DSIGMADZ,DSIGMA)
Lopperator = Lopperator.todense()
LBCsigma = LBCsigma.todense()

DX = DX.todense()
for i in range(Ny):
    Lopperator[index[i,0],:] = -DX[index[i,0],:]
    Lopperator[index[i,-1],:] = -DX[index[i,-1],:]
for i in range(Nx):
    Lopperator[index[0,i],:] =  -LBCsigma[ index[0,i] , : ]
    Lopperator[index[-1,i],:] = 0
    Lopperator[index[-1,i],index[-1,i]] = 1
f[index[-1,:]] = u0
```

The `todense()` operation is to allow us to access the elements by using our `index` array, which `SciPys` sparse functionality does not allow.

With the linear operator constructed, we can solve for Φ, which is allows us to calculate the time-derivatives (6.26) and output these.

The full Laplacian calculator can be found in appendix G.1

6.2.2.2 The time-integrator

The time integrator is composed of two steps, calculating the initial condition and the time-stepping part. The initial condition is easy to calculate, as we only need to define a surface elevation η and the surface velocity potential $\tilde{\phi}$. For the time-integration part, we will be using the Runge-Kutta 4-step method outlined in (6.14).

For testing this, we introduce the a standing wave in our system, as [YA96, pg. 145], which is defined as

$$\eta_S(x) = \frac{1}{2}\sum_{J=1}^{8} J^{\frac{1}{4}} a_J \cos(J\pi x)$$

There the coefficients a_J are defined as

J	$\mathbf{a}_J$
1	$0.8867 \cdot 10^{-1}$
2	$0.5243 \cdot 10^{-2}$
3	$0.4978 \cdot 10^{-3}$
4	$0.6542 \cdot 10^{-4}$
5	$0.1007 \cdot 10^{-4}$
6	$0.1653 \cdot 10^{-5}$
7	$0.2753 \cdot 10^{-6}$
8	$0.4522 \cdot 10^{-1}$

This allows us to test our model, and we use this as initial conditions, with the zero vector being the scalar velocity potential $\tilde{\phi}$.

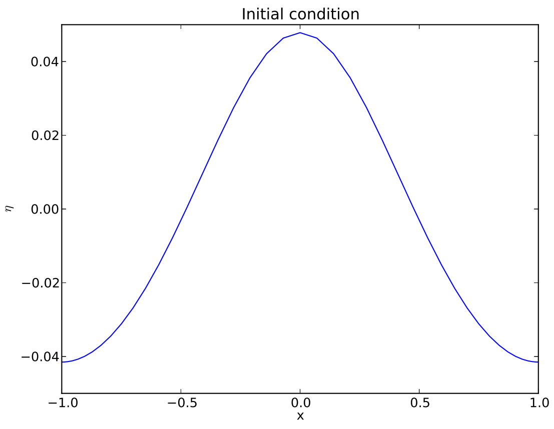


Figure 6.18: The initial value of the standing wave for the wave model.

Figure 6.18 shows us the initial condition for our standing wave, which we use to test our model. [YA96, pg. 145] tells us that the period of the standing wave is $T = 1.13409$, which allows us to check exactly one period of movement.

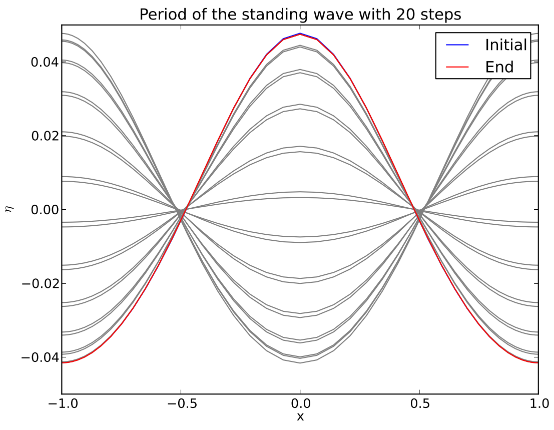


Figure 6.19: The period of movement for the standing wave.

Figure 6.19 shows us that the wave is indeed a standing wave, and the period is as expected, confirming our model.

6.2.2.3 Calculating the force on the right boundary

The calculation of the right boundary is done according to the procedure explained in section 6.2.1.2. We can calculate all parts of the pressure easily from the values we already posses, as we can find $U = D_X\Phi$ and $W = \partial_z\sigma D_\sigma\Phi$. Since we will only be concerning ourselves with the right boundary, the calculation of p apart from the integral become simple vector operations. For the calculation of the integral, we will need to transform the interval between the surface and the point to a new Legendre-Gauss-Lobatto grid, where we can approximate the integral using this quadrature. This is done for each point, where the values of W are transformed. Since we will be needing the time-derivative of W, we will need to transform the previous value of W as well, approximating the time derivative by a simple difference quotient dependent on our time-step size.

This calculation is done by the following code-snippet – where the analytic expressions of the pressure are already stored in `t`

```
for i in range(Ny):
    Z,WZ = LegPol.GaussLobattoQuadrature(Ny-1)

    V = LegPol.GradVandermonde1D(Z[Ny-1-i:],i,0)
    V2 = LegPol.GradVandermonde1D(Z,i,0)
    VInv = np.linalg.inv(V)
    TRANS = np.dot(V2,VInv)
    W2Z = np.dot(TRANS,W2[Ny-1-i:])
    WLASTZ = np.dot(TRANS,WLAST[Ny-1-i:])
    WDT = (W2Z-WLASTZ)/tstep
    integral = 0
    for j in range(Ny):
        integral = integral + WZ[Ny-1-j]/2*WDT[Ny-1-j]

    p[Ny-1-i] = p[Ny-1-i] + (zeta[-1]-ZREAL[Ny-1-i])/2*integral

p = p*dens
Force = 0
for j in range(Ny):
    Force = Force + p[j]*WZ[j]
Force = Force*2/(d)
```

The scaling factors are calculated according to the change in variable from real basis to Legendre-Gauss-Lobatto basis, as calculated here

$$
\begin{array}{lll}
z_L = 2\dfrac{z+h}{\eta+h} - 1 & \dfrac{dz_L}{dz} = \dfrac{2}{\eta+h} = \dfrac{2}{d} & \begin{array}{r} z \in [-h,\eta] \\ z_L \in [-1,1] \end{array} \\
\tilde{z}_L = 2\dfrac{\tilde{z}-z_0}{\eta-z_0} - 1 & \dfrac{d\tilde{z}_L}{d\tilde{z}} = \dfrac{2}{\eta-z_0} & \begin{array}{r} \tilde{z} \in [z_0,\eta] \\ \tilde{z}_L \in [-1,1] \end{array}
\end{array}
$$

In order to test this, we will calculate the force from the standing wave, as well as the force from the completely steady water.

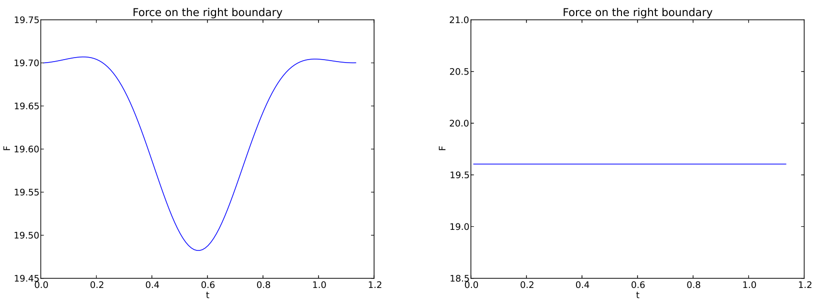


Figure 6.20: The force on the right boundary for the standing wave to the left, and the steady water to the right

Figure 6.20 shows us that the force of the steady water is steady, as expected, and that when there is movement in the water, the force will vary around this steady force. We can also compare the static force of the steady water, with the definition of the static force from [EK06, Eq. 2.86]

$$F_{\text{static}} = \rho(\eta+h)g\eta - \frac{1}{2}\rho g\left(\eta^2 - h^2\right)$$

Since the water is steady, $\eta = 0$ and $h = 2$, which leads us to the following

$$F_{\text{static}} = \frac{1}{2}\rho g h^2 = 2\rho g$$

Since $\rho = 0.998$ and $g = 9.82$, we get that $F_{\text{static}} = 19.6$, which excactly the same as we got for our steady water solution shown in figure 6.20.

6.2.3 Introducing uncertainty into the amplitude

We want to introduce uncertainty into the amplitude, to see how it effects the size of the force against the boundary. We want to test it initially, and stick with our standing wave start-condition, where we let the amplitude be modified by a constant $K \sim U(0.9, 1.1)$.

We will be using the stochastic collocation method, and since we want to model the constant by the uniform distribution, we want to model the stochastic variable using the Legendre polynomials. We will use the stochastic collocation method, simply by running the standard model $N_{UQ} = 10$ times for different iterations of the stochastic variables, and we will calculate the mean and variance after all these realizations of the problem.

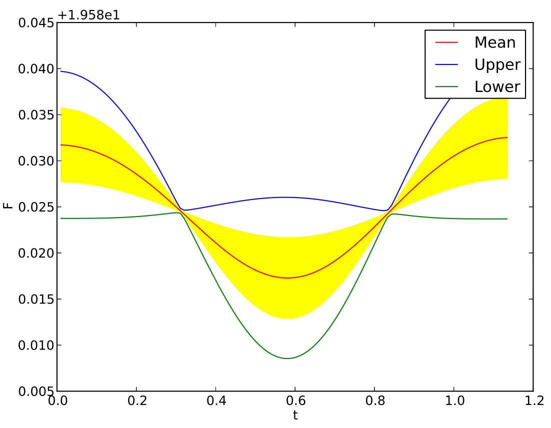


Figure 6.21: The quantification of uncertainty on the force affecting the right side of the boundary in our wave model.

As we can see in figure 6.21, as the amplitude of the standing wave initial condition is uncertain, so is the amount of force the right side is affected with, in amplitude. The points where the the amplitude shifts, there is very little uncertainty, signifying that the amplitude of the wave is not affecting the speed at which the force of the wave hits the boundary – at least not at this scale.

6.2.4 Numerical experiments with uncertainty

For the numerical experiments, we will be using a simple gaussian curve, very much like the density function for the gaussian distribution, to emulate a starting wave in our domain.

For this purpose we will be using the function

$$\eta_0(x) = \frac{e^{\frac{(x-0.01)^2}{2*0.08^2}}}{100} \tag{6.30}$$

Where we will be truncating all values less that e^{-6}, as to simulate a singular wave. We will use this since it confines to our space nicely, and is smooth enough to approximate correctly.

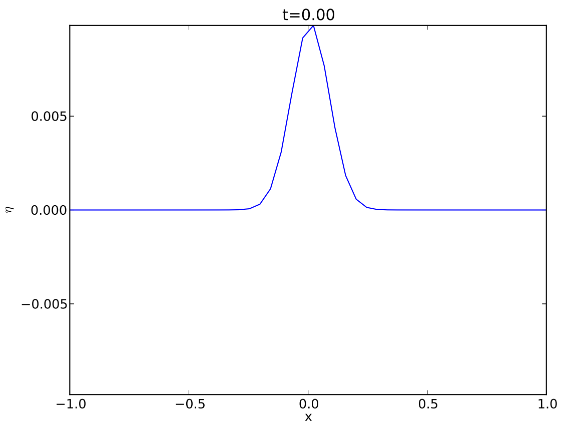


Figure 6.22: The initial condition for our gaussian wave on a grid with $N_x = 70$ and $N_z = 25$

We will first calculate the solution for this, as to analyze it before introducing the uncertainty. We simply solve it until $t = 1$, to briefly evaluate the evolution of the solution.

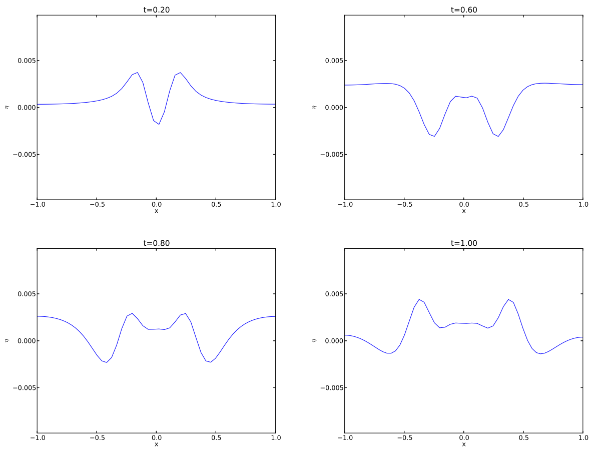


Figure 6.23: The evolution of the wave model for the initial condition (6.30) at certain points in time

We see in figure 6.23 that the initial wave collapses into the wave-tank and splits into two separate waves, which move towards their respective boundaries.

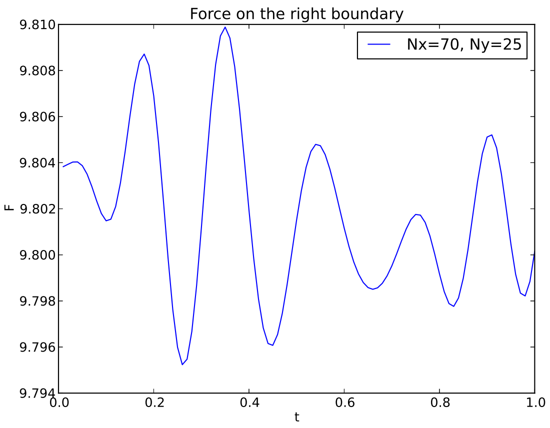


Figure 6.24: The force on the right boundary wall, exerted by the water.

Figure 6.24 shows us that the force varies as the water moves internally. We will be expecting a peak somewhat when the wave we see forming hits the wall.

6.2.4.1 Variation in the amplitude

We will start by varying the amplitude of our gauss pulse initial condition. We allow the gaussian pulse to have 10% uncertainty in the amplitude, by introducing the new initial value function

$$\hat{\eta}_0(x) = K\eta_0(x) \qquad\qquad K \sim U(0.9, 1.1)$$

We will be implementing the stochastic collocation method for this, using Legendre polynomials as we have done before. The code for this implementation is located in appendix G.2.

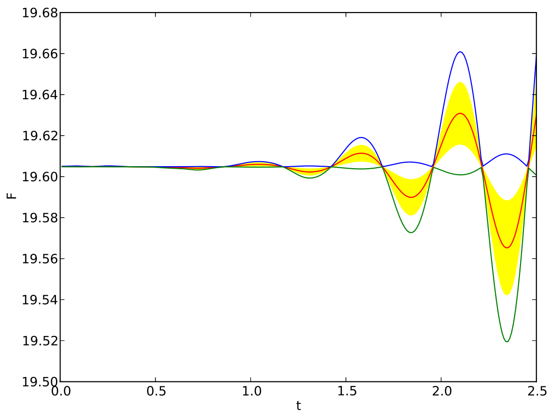


Figure 6.25: The 10% uncertainty on the initial amplitude propagated through the force on the right-hand side boundary.

We see in figure 6.25 that the changes in the force, albeit growing, are relatively small compared to the static force. The uncertainty does however do quite a lot of change at the local minima and maxima, where the value of the force can change quite a deal from the mean. We notice that there are certain knots where the force is always the same value, regardless of the initial amplitude.

6.2.4.2 Variation in the starting point of the wave

In order to shed some light on the constant knots we saw in figure 6.25, we want to keep the initial amplitude constant, and instead introduce a uncertainty on the midpoint of the wave. We create a new initial value function, from (6.30)

$$\tilde{\eta}_0(x) = \frac{e^{\frac{(x-\omega)^2}{2*0.08^2}}}{100} \qquad\qquad \omega \sim U(-0.02, 0.02)$$

We use the exact same framework as when we introduced uncertainty to the amplitude, and compute the result

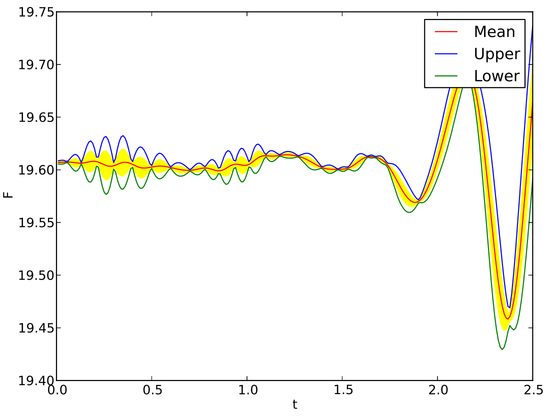


Figure 6.26: The uncertainty in the mid-point of our initial pulse $\omega \sim U(-0.02, 0.02)$ propagated to the force on the right hand wall.

We see from figure 6.26 that as long as the the starting amplitude is unchanged, the value of the force hitting the wall is close to equal, just either delayed or hurried according to mean. But that is only when the waves are more aligned, which they seem to become at around $t = 1.75$. Until that, it seems that where ever the pulse starts creates different waves, that hit the wall at different times, allowing those "bubbles" to form on our force plot.

6.2.5 Conclusions for the wave tank model

The wave tank experiment has shown us that the nonlinear systems can pack quite a surprise for us, as we have seen in figure 6.26, a small change in the initial condition can affect the solution in different ways at different times – something not readily apparent from the model itself. This proves that uncertainty quantification is a massively useful tool, as it allows us to quantify the effects of uncertainty we would not be able to understand without this technique. While this was a relatively heavy model, since each time-step required four solutions to a Laplace problem on a 70×25 grid, it would have been impossible to use the simple Monte Carlo method to acquire these results.

Chapter 7

Conclusions

The spectral methods and generalized polynomial chaos methods make for a great combination when used for quantifying uncertainty through differential equations. The combination of fast convergence for both types of methods grant enormous opportunity for implementation on relatively complex models, as the burden of giant matrix operations become somewhat lessened by having fewer points.

While the methods are smart, they require a good understanding of the problem to implement properly, and because they work on the whole spectra, it can be hard to locate a single error if the methods are not very well understood. While being more complicated than some easier models, the collocation methods, both spectral and stochastic merge some good of both worlds, allowing for relatively straightforward implementation, while retaining the spectral convergence. With these two methods, we will be able to do uncertainty quantification on most differential equation problems relatively easy, while the Galerkin methods require somewhat more work per problem.

These methods can of course be paired with other numerical methods or technologies, to increase the speed further, allowing even more applications of these methods. One such choice could be to parallelize the processes. With the uncoupled nature of the stochastic collocation method, many of the processes would easily run in parallel, allowing for fast and efficient calculations, without fear of interdependencies slowing the gain of parallelization.

The applications of uncertainty quantification are everywhere, as it is often hard to both measure correctly, and replicate exactly. For some applications this might change the outcome a whole lot, as we saw in burgers equation, or generate unanticipated results as we saw in the wave tank model. It might also just highlight the inner workings of a model that are not readily apparent from the way things work, as we saw in the lid driven cavity model.

Appendix A

Conventions for notation and plotting

A.1 Differential notation

We will be using the following notation for differential notation interchangeably

$$\frac{du(x,t)}{dt} = \partial_t u(x,t) = u_t(x,t)$$

A.2 Uncertainty quantification in plots

We will use two-dimensional plots as a way to visualize the uncertainty quantification. We will be using a plot from figure 5.7 to illustrate the point.

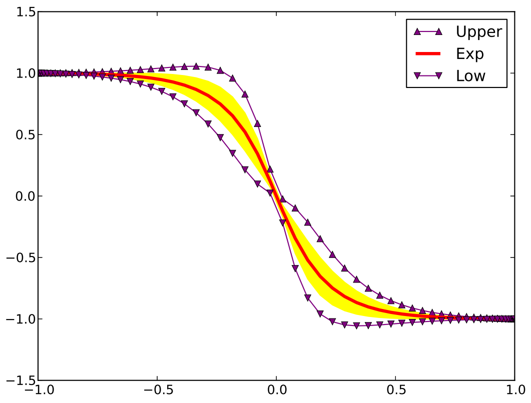


Figure A.1: The solution to Burgers equation for $N = 60$ and v $U(0.05, 0.2)$ modeled by 10 iterations of the Legendre polynomial. The yellow band represents the mean $\mu \pm$ the standard derivation σ, and the purple lines define $\mu \pm 2\sigma$. 95% of the solutions will be inside the purple lines.

We see here the mean of the of the function, as well as a yellow colored area which is within the standard deviation of the mean. The lines tagged "Upper" and "Lower" refer to the upper and lower bound for begin within the mean and two standard deviations. Optionally the highest and lowest value calculated might be plotted as well.

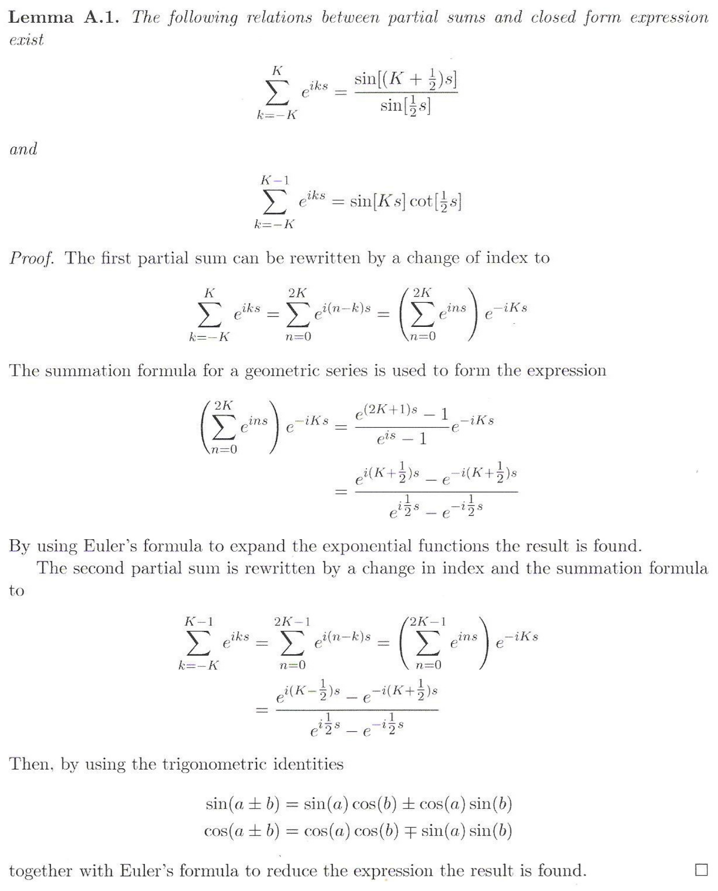
A.3 Coding notation

When reading `Python` code, one should be aware that indentation is a nesting of the functions, and as soon as there is a piece of code not indented, the nested part has ended.

The $\#$ sign signifies comments in `Python`, while three concurrent quotation marks either initialize or end a block comment. The character \ signifies that the current line is broken for ease of view only, continuing on the next line – akin to the `MATLAB` operator `...`.

Appendix B

Allan P. Engsig-Karup, notes



Lemma A.1. *The following relations between partial sums and closed form expression exist*

$$\sum_{k=-K}^{K} e^{iks} = \frac{\sin[(K+\frac{1}{2})s]}{\sin[\frac{1}{2}s]}$$

and

$$\sum_{k=-K}^{K-1} e^{iks} = \sin[Ks]\cot[\tfrac{1}{2}s]$$

Proof. The first partial sum can be rewritten by a change of index to

$$\sum_{k=-K}^{K} e^{iks} = \sum_{n=0}^{2K} e^{i(n-k)s} = \left(\sum_{n=0}^{2K} e^{ins}\right) e^{-iKs}$$

The summation formula for a geometric series is used to form the expression

$$\begin{aligned}\left(\sum_{n=0}^{2K} e^{ins}\right) e^{-iKs} &= \frac{e^{(2K+1)s}-1}{e^{is}-1}e^{-iKs}\\ &= \frac{e^{i(K+\frac{1}{2})s}-e^{-i(K+\frac{1}{2})s}}{e^{i\frac{1}{2}s}-e^{-i\frac{1}{2}s}}\end{aligned}$$

By using Euler's formula to expand the exponential functions the result is found.

The second partial sum is rewritten by a change in index and the summation formula to

$$\begin{aligned}\sum_{k=-K}^{K-1} e^{iks} = \sum_{n=0}^{2K-1} e^{i(n-k)s} &= \left(\sum_{n=0}^{2K-1} e^{ins}\right) e^{-iKs}\\ &= \frac{e^{i(K-\frac{1}{2})s}-e^{-i(K+\frac{1}{2})s}}{e^{i\frac{1}{2}s}-e^{-i\frac{1}{2}s}}\end{aligned}$$

Then, by using the trigonometric identities

$$\begin{aligned}\sin(a\pm b) &= \sin(a)\cos(b)\pm\cos(a)\sin(b)\\ \cos(a\pm b) &= \cos(a)\cos(b)\mp\sin(a)\sin(b)\end{aligned}$$

together with Euler's formula to reduce the expression the result is found. □

Figure B.1: Origin of Lambda A.1

Appendix C

Code used in chapter 3

C.1 Generating plots

Lagrange polynomials

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 22 15:20:30 2013

@author: cbrams

This function is simply showing the lagrance polynomials for N=6
"""
import numpy as np
import matplotlib.pyplot as plt

N = 6
i = np.array(range(N+1))
xj = np.zeros((len(i)))

#Calculate the points
```

```
for j in i:
        xj[j] = 2.*np.pi/float(N)*j

#Create a fine grid to plot on
x = np.linspace(xj[0]-1e-12,xj[N]+1e-12,100)

#Plot the Lagrange polynomials
for j in i:
    hj = 1./float(N) * np.sin(N/2. *(x-xj[j])) * np.cos(1./2.*(x-xj[j])) / np.sin(1./
    hj[np.isnan(hj)] = 0.
    plt.plot(x/(2.*np.pi),hj)
plt.grid()
plt.xlim(0.,1)
plt.title("Lagrange polynomials for N=%d"%N)
plt.savefig("../../Latex/Billeder/Chapter3/LagrangePolynomials.eps")
```

Legendre polynomials

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 22 15:20:30 2013

@author: cbrams

This script is intended to show the Legendre polynomials
"""

#Initilization
import numpy as np
import DABISpectral1D
import matplotlib.pyplot as plt

#Generate x-points
x = np.linspace(-1,1,100)
for N in range(6):

    #Initialize Legendre polynomial, and calculate values
    # (0.0,0.0) is (alpha, beta)
    polyLeg = DABISpectral1D.Poly1D(DABISpectral1D.JACOBI,(0.0,0.0))
    p5Leg = polyLeg.GradEvaluate(x, N, 0);   # N is N

    #Plot polynomials
    plt.plot(x,p5Leg,label='N = '+str(N))
    plt.legend()
plt.title('Legendre polynomials')
plt.savefig("../../Latex/Billeder/Chapter3/LegendrePolynomials.eps")
```

Gibbs phenomenon

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 20 11:38:19 2013

@author: cbrams

This script is intended to show Gibbs phenomenon
"""
import numpy as np
import DABISpectral1D as DB
import matplotlib.pyplot as plt

#Define the step-function
def f(x):
    F = np.zeros(x.shape)
    F[x>0] = 1
    F[x<0] = -1
    return F

#Plot the step-function
plt.figure(1)
xi = np.linspace(-1,1,num=1000)
plt.plot(xi,f(xi),label="Original")

#Initialize polynomial
LegPol = DB.Poly1D(DB.JACOBI,(0.0,0.0))

#Increasing N and room for error
N = np.array([5,15,25,35])
Err = np.zeros(N.shape)
for i in range(N.size):
    #Approximate the step function using nodal values
    n = N[i]
    x,w = LegPol.GaussLobattoQuadrature(n)
    F = f(x)

    #Calculate modal coefficients and interpolation
    fhat = LegPol.DiscretePolynomialTransform(x,F,n)
    fi = LegPol.LagrangeInterpolate(x,F,xi)

    #Plot the coefficients
    plt.figure(1)
    plt.plot(xi,fi,label="N=%d"%n)
    plt.figure(2)
    plt.semilogy(range(n+1),fhat,'o',label="N=%d"%n)
    Err[i] = max(abs(fi-f(xi)))

#Plot the results
plt.figure(1)
plt.legend(loc=4)
plt.savefig("../../Latex/Billeder/Chapter3/GibbsPhenomenon.eps")
plt.figure(2)
plt.legend(loc=6)
plt.savefig("../../Latex/Billeder/Chapter3/GibbsCoefficients.eps")
plt.figure(3)
plt.semilogy(range(N.size),Err)
plt.title("Error")
plt.show()
```

Aliasing

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 20 13:03:12 2013

@author: cbrams

This script is intended to illustrate the aliasing problem
"""
import numpy as np
import matplotlib.pyplot as plt

#Calculate the arrays
x = np.linspace(0,2*np.pi,1000)
X = np.linspace(0,2*np.pi,9,endpoint=True)

#Calculate the complex exponentials
```

```
z2 = 1j*2*x
z6 = 1j*-6*x
Z = 1j*2*X
E2 = np.exp(z2)
E6 = np.exp(z6)
E = np.exp(Z)

#Plot it all together
plt.figure()
plt.plot(x,np.real(E2),label="k=2",linestyle="--")
plt.plot(x,np.real(E6),label="k=-6",linestyle="--")
plt.plot(X,np.real(E),'o')
plt.xlim(0-0.02,2*np.pi+0.02)
plt.ylim(-1.02,1.02)
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter3/Aliasing.eps")
```

Appendix D

Code used in chapter 4

D.1 Differential Matrices

The Diff function

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  2 19:52:09 2013

@author: cbrams

This script is intended to calculate the differential operators
"""

#Initilization
import numpy as np
import DABISpectral1D as DB

#Function generating differentiative matrix
def DF(N):
    D = np.zeros((N,N))
    #Generate for even N
    if (N%2)==0:
        for l in range(N):
            for k in range(N):
                if l==k:
                    D[k][l] = 0
                else:
                    D[k][l] = (1/2.)*(-1)**(1+k+l)\
                    *np.cos(np.pi*(-k+l)/N)/np.sin(np.pi*(-k+l)/N)
    else:
        D = 0
    return D

def DP(N):
    #Initialize Polynomial
    P = DB.Poly1D(DB.JACOBI, (0.0,0.0))

    #Generate grid
    (xGL,w) = P.GaussLobattoQuadrature(N-1)

    #Generate Vandermonde Matrices
    V = P.GradVandermonde1D(xGL,N-1,0)
    Vx = P.GradVandermonde1D(xGL,N-1,1)

    #Solve for D
    D = np.linalg.solve(V.T,Vx.T).T
    return [D,xGL]
```

D.2 Test functions

The test for the Fourier differential operator

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  2 19:54:08 2013

@author: cbrams

This script is intended to test the fourier differentiation
operator
```

```python
"""

#Initilization
from __future__ import division
from Diff import DF
import numpy as np

#Setting number of N values evaluated
Niter = 15
error = np.zeros(Niter)

for i in range(Niter):
    #Initiliza loop
    N = 4*i+5
    x = np.linspace(0,2,N)

    #Generating true values
    dv = np.cos(np.pi*x)*np.pi*np.exp(np.sin(np.pi*x))
    v = np.exp(np.sin(np.pi*x))

    #Differentiating using D
    Dv = np.zeros(v.shape)
    Dv[0:N-1] = np.dot(DF(N-1),v[0:N-1])*np.pi
    Dv[N-1] = Dv[0]
    error[i] = max(abs(dv-Dv))


#Plotting approximation
import matplotlib.pyplot as plt
plt.figure()
plt.plot(x,dv,label='Analytic',color='b')
plt.plot(x,Dv,linestyle=' ',marker='x',color = 'r',\
label='Approximation')
plt.xlabel("x")
plt.ylabel("v'(x)")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter4/FourierDTestResult.eps")


#Plotting error
plt.figure()
NN = 4*np.array(range(Niter))+5
plt.loglog(NN,error,'r',label='Error')
plt.loglog(NN,1/NN**(3),'b',label='N^(-3)')
plt.xlabel("N")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter4/FourierDTestConvergence.eps")

#Show plots
#plt.show()
```

The test for the Legendre differential operator

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan  3 14:01:08 2013

@author: cbrams

This script is intended to test the LEgendre differential operator
"""
#Initialization
from numpy import pi,sin,exp,cos
import numpy as np
import DABISpectral1D as DB
from numpy.linalg import norm
import matplotlib.pyplot as plt
from Diff import DP

#Correct value functions
v = lambda x: exp(sin(pi*x))
dv = lambda x: pi*cos(pi*x)*exp(sin(pi*x))

#Initialize polynomials
P = DB.Poly1D(DB.JACOBI, (0.0,0.0))

#Initialize loop over N
nn = 5
error = np.zeros(nn)
for i in range(nn):
    N = 2**(i+2)

    #Generate D and x
    [D,xGL] = DP(N)
    x = xGL + 1

    #Calculate function values
    f = v(x)

    #Calculate approximated and analytic function values
    df = np.dot(D,f)
    Df = dv(x)

    #Record error
    error[i] = norm(df-Df,2)

plt.figure()
plt.plot(x,Df,label='Analytic',color='b')
plt.plot(x,df,linestyle=' ',marker='x',color = 'r',\
label='Approximation')
plt.xlabel("x")
plt.ylabel("v'(x)")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter4/LegendreDTestResult.eps")


#Print error
plt.figure()
NN = (np.ones(nn)*2)**(np.array(range(nn))+1)
plt.loglog(NN,error,label="Error")
plt.loglog(NN,NN**(-2),label="N^(-2)")
plt.xlabel("N")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter4/LegendreDTestConvergence.eps")
```

D.2.1 The test for the two-dimensional differential

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 23 16:42:52 2013

@author: cbrams

This is intended for creating a plot showing the effectiveness
of the differential operators.
"""

from __future__ import division
```

```
import numpy as np
import DABISpectral1D as DB
from Diff import DP
import matplotlib.pyplot as plt

N = 20

LegPol = DB.Poly1D(DB.JACOBI,(0.0,0.0))

#Generate the mesh
x,wx = LegPol.GaussLobattoQuadrature(N)
y,wy = LegPol.GaussLobattoQuadrature(N)

X,Y = np.meshgrid(x,y)

#Calculate the initial and true value
C = np.cos(X*Y)
Cprime = -np.cos(X*Y)*X*Y-np.sin(X*Y)

#Generate the matrices
[Dx,_] = DP(N+1)
[Dy,_] = DP(N+1)

#Creating the matrices for multidimensional use
DX = np.kron(Dx,np.identity(N+1))
DY = np.kron(np.identity(N+1),Dy)

#Flattening the initial condition
C = C.flatten("F")

#Calculating the derivative and reshaping
Cd = np.dot(DX,np.dot(DY,C))
Cd = Cd.reshape([N+1,N+1],order="F")

#Plotting
plt.figure()
plt.imshow(np.abs(Cd-Cprime),origin="lower",extent=[-1,1,-1,1])
plt.colorbar()
plt.xlabel('x')
plt.ylabel('y')
plt.axis('normal')
plt.savefig("../../Latex/Billeder/Chapter4/MultiDimDiffTest.eps")
```

D.2.2 The test for the sparse matrices speed

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 24 12:29:22 2013

@author: cbrams

This is intended to test the sparse matrix dot product speed
"""

import time
import numpy as np
from scipy.sparse import lil_matrix
from numpy.random import rand

#We're running the scheme 1000 times
tN = 1000
timeSparse = np.zeros(tN)
timeFull = np.zeros(tN)
errors = np.zeros(tN)
for i in range(tN):
    #Creating and populating the matrix
    A = lil_matrix((1000, 1000))
    A[0, :100] = rand(100)
    A[1, 100:200] = A[0, :100]
    A.setdiag(rand(1000))

    #Transforming the matrix
    A = A.tocsr()
    b = rand(1000)

    #Time the Dense product as well
    AA = A.todense()
    T1 = time.clock()
    x = A.dot(b)
    T2 = time.clock()
    y = np.dot(AA,b)
    T3 = time.clock()

    #Calculate the error between the two solutions
    errors[i] = np.linalg.norm(x**2-np.array(y)**2,ord=2)

    #Storing the time
    timeSparse[i] = T2-T1
    timeFull[i] = T3-T2

#Calculate and print the averages
TS = np.average(timeSparse)
TF = np.average(timeFull)
E = np.average(errors)

print TS
print TF
print E
```

D.3 Practical implementations

The test script for the boundary value problem

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 24 15:48:05 2013

@author: cbrams

This script is intended to solve the problem with BVP presented
in the spectral chapter
"""
#Initialization
import numpy as np
from numpy.linalg import norm
import DABISpectral1D
import matplotlib.pyplot as plt
from Diff import DP
plt.close('all')

#Generate iteration parameters
N = np.linspace(10,80,num=71)

epsilon = 0.01
error = np.zeros([len(N)])
for i in range(len(N)):
    n = int(N[i] )
    b = -1

    #Initialize polynomial
    polyLeg = DABISpectral1D.Poly1D(DABISpectral1D.JACOBI, (0.0,0.0))

    #Generate Quadrature and transformation
    [D,y] = DP(n)
    x = (y+1)/2

    #Construct L
    L = -epsilon*4*np.dot(D,D) + b*2*D

    #Implement boundary conditions
    L[0,:] = 0
    L[-1,:] = 0
    L[0,0] = 1
    L[-1,-1] = 1

    #Generate right-side function with boundary conditions
    f = np.ones((n,1))
    f[0]  = 0
    f[-1] = 0

    #Solve system and calculate exact solution
    u = np.linalg.solve(L,f)
    uExact = (np.exp((1-x)/epsilon)+np.exp(1/epsilon)*\
    (x-1)-x)/(1-np.exp(1/epsilon))

    #Calculate error
    error[i] = norm(u[:,0]-uExact,np.inf)

#Plot function
plt.figure()
plt.plot(x,u,'ro',label="Approximation")
plt.plot(x,uExact,label="Exact")
plt.xlabel("x")
plt.title("N=" + str(n) + " and epsilon="+str(epsilon))
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter4/EpsilonSolution.eps")


#Errorplot
plt.figure()
```

```python
plt.loglog(N,error[:],'r',label="Error \epsilon{}="+str(epsilon))
plt.legend()
plt.xlabel("N")
plt.savefig("../../Latex/Billeder/Chapter4/EpsilonError.eps")
```

The test script for Burgers' equation

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 21 20:05:23 2013

@author: cbrams

This script is intended to solve Burger's problem with a
pseudo-spectral method
"""

from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
from Diff import DP
from scipy.integrate import odeint


def initialValue(x):
    #Calculate initial value
    res = -np.tanh((x))
    #Adjust boundaries
    res = res/np.abs(res[0])
    return res

def dudt(u,t,v,Dx,BoundaryFixer):
    #Calculating linear operator
    unew= -u*np.dot(Dx,u)+v*np.dot(Dx,np.dot(Dx,u))
    #Adjust for boundaries
    unew = unew*BoundaryFixer
    return unew


#Initialization
xN = 80
v = 0.1
```

```python


#Generating Dx and u0
Dx,x = DP(xN)
u0 = initialValue(x)

#Configuring time needed
tEnd = 4
tNum = 1000
#We create the time-array we want values for
t = np.linspace(0,tEnd,tNum)

#Create boundary fixer to cancel the time derivatives
# at the boundaries.
BDFix = np.ones(x.shape)
BDFix[0] = 0
BDFix[-1] = 0

#Start the time-integrator
v0 = v
usol = odeint(dudt,u0,t,tuple([v0,Dx,BDFix]))

#Plot the solutions
plt.figure()
plt.title("Solution to Burgers' equation")
plt.plot(x,initialValue(x),'r--',linewidth=2,label="Initial Value")
for i in range(1,6):
    plt.plot(x,usol[i*100,:],label="t=%.2f"%t[i*100])
plt.plot(x,usol[-1,:],label="t=%2d"%t[-1])
plt.xlabel("x")
plt.ylim(-1.1,1.1)
plt.legend()

plt.savefig("../../Latex/Billeder/Chapter4/BurgersEQSpectral.eps")
```

Appendix E

Code used in chapter 5

E.1 Visualization

Visualising the first six Hermite polynomials

```python
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 12 10:39:12 2013

@author: cbrams

This script is intended for showing the hermite polynomials
"""

import DABISpectral1D as DB
import numpy as np
import matplotlib.pyplot as plt

#Initialize Polynomial
HPol = DB.Poly1D('HermitePprob',())

```

```python
#Generate grid
xlim = 3
N = 100
x = np.linspace(-xlim,xlim,N)

plt.figure()
for i in range(6):
    #Evaluate and plot polynomials
    polyval = HPol.GradEvaluate(x,i,0)
    plt.plot(x,polyval,label="HP"+str(i))

plt.legend(loc=9)
plt.savefig("../../Latex/Billeder/Chapter5/HermitePolynomials.eps")
```

E.2 Practical examples

E.2.1 Test equation

Monte Carlo mean approximation

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 10 12:52:18 2013

@author: cbrams

This script is designed for calculating the mean by the
Monte Carlo methods
"""
#Imports
from __future__ import division
import numpy as np
from scipy.integrate import odeint
from numpy.random import normal as alpha
from math import e
import matplotlib.pyplot as plt


#Define time-derivative for one random parameter
def dudt(u,t,args):
    return -args*u

#True solution
def u(t,mean,std):
    a = alpha(mean,std)
    return e**(-a*t),a

#Define mean, standard deviation and expectation
mean = 9.
std = 3.
tTrue = 0.5
TrueExp = e**(1/2*tTrue**2*std**2-tTrue*mean)

tN = 100
t = np.linspace(0,tTrue,tN)


#Number of realizations
M = 100000

NSol = np.zeros(M)
NRunningMean = np.zeros(M)
NRunningVar = np.zeros(M)
for N in range(M):
    #Calculate a random starting point
    y0,a = u(0,mean,std)

    #Integrate over time
    sol = odeint(dudt,y0,t,tuple([a]))

    #Calculate the running mean
    NSol[N] = sol[-1]
    NRunningMean[N] = np.mean(NSol[:N])

plt.figure()
plt.plot(range(M),NRunningMean,label="Running Mean");
plt.plot(range(M),np.ones(M)*TrueExp,label="Expectation")
plt.xlabel("N")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/TestEQMonteCarloRunningMean.eps")

plt.figure()
plt.loglog(range(M),abs(NRunningMean-TrueExp),label="Error");
plt.xlabel("N")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/TestEQMonteCarloRunningError.eps")
```

Collocation mean and variance approximation

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jan 11 12:46:31 2013

@author: cbrams

This script is intended to calculate the mean and variance
for the test equation by the spectral collocation method
"""
from __future__ import division
import DABISpectral1D as DB
import numpy as np
from math import e
from scipy.integrate import odeint
import matplotlib.pyplot as plt

#True solution
def u(t,a):
    return e**(-a*t)

def dudt(u,t,a):
    return -a*u
#Initialize Polynomial
HPol = DB.Poly1D('HermitePprob',())

#Set number of time steps, as well as calculating the real values
tN = 1000
mean = 9
std = 3
tTrue = 0.5
TrueExp = e**(1/2*tTrue**2*std**2-tTrue*mean)
TrueVar = (e**(tTrue**2*std**2)-1)*\
e**(-2*tTrue*mean+tTrue**2*std**2)

t = np.linspace(0,tTrue,tN)

NMax = 25
```

```python
Exp = np.zeros(NMax-1)
ExpVar = np.zeros(NMax-1)
for N in range(1,NMax):
    #Generate points
    x,w = HPol.GaussQuadrature(N)

    #Generate the random variable representation
    a = mean + (std)*x;

    #Fint initial value
    u0 = u(0,a)

    #Integrate over time
    sol = odeint(dudt,u0,t,tuple([a]))
    ufinal = sol[-1]

    #Calculate Expectation and variance
    Exp[N-1] = sum(w[:,0]*ufinal)
    ExpVar[N-1] = sum(w[:,0]*((ufinal-Exp[N-1]))**2)

plt.figure
NN = np.array(range(1,NMax))
plt.plot(NN,Exp,label="Approximated exp")
plt.plot(NN,np.ones(NMax-1)*TrueExp,label="True exp")
plt.xlabel("N")
plt.legend()

NN = np.array(range(1,NMax))
plt.plot(NN,ExpVar,label="Approximated variance")
plt.plot(NN,np.ones(NMax-1)*TrueVar,label="True variance")
plt.xlabel("N")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/TestEQCollocationMean.eps")

plt.figure()
plt.semilogy(NN,abs(Exp-TrueExp),label="Error mean")
plt.semilogy(NN,abs(ExpVar-TrueVar),label="Error Variance")
plt.xlabel("N")
plt.title("Absolute error")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/TestEQCollocationError.eps")
plt.show()

np.savez("test",Exp=Exp,ufinal=ufinal)
```

Monte Carlo mean approximation for the uniform case

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 21 10:45:51 2013

@author: cbrams

This script is intended to calculate the Monte Carlo mean
for a uniform distribution
"""

#Imports
from __future__ import division
import numpy as np
from scipy.integrate import odeint
from numpy.random import uniform as alpha
from math import e
import matplotlib.pyplot as plt


#Define time-derivative for once random parameter
def dudt(u,t,args):
    return -args*u

#True solution
def u(t,start,end):
    a = alpha(low=start,high=end)
    return e**(-a*t),a

#Define mean, standard deviation and expectation
start = -7
end = 2
tTrue = 0.5
TrueExp = (-e**(-start*tTrue)+e**(-end*tTrue))/(tTrue*(start-end))

tN = 100
t = np.linspace(0,tTrue,tN)


#Number of realizations
M = 100000

```

```
NSol = np.zeros(M)
NRunningMean = np.zeros(M)
for N in range(M):
    #Generate random start
    y0,a = u(0,start,end)

    #Integrate over time
    sol = odeint(dudt,y0,t,tuple([a]))

    #Calculate running mean
    NSol[N] = sol[-1]
    NRunningMean[N] = np.mean(NSol[:N])

plt.figure()
plt.plot(range(M),NRunningMean,label="Running Mean");
plt.plot(range(M),np.ones(M)*TrueExp,label="Expectation")
plt.xlabel("N")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/TestEQMonteCarloRunningMeanUniform.eps

plt.figure()
plt.loglog(range(M),abs(NRunningMean-TrueExp),label="Error");
plt.xlabel("N")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/TestEQMonteCarloRunningErrorUniform.ep
```

Collocation mean approximation for the uniform case

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 12 10:58:40 2013

@author: cbrams
"""
from __future__ import division
import DABISpectral1D as DB
import numpy as np
from math import e
from scipy.integrate import odeint
import matplotlib.pyplot as plt

#True solution
```

```
def u(t,a):
    return e**(-a*t)

def dudt(u,t,a):
    return -a*u
#Initialize Polynomial
JPol = DB.Poly1D('Jacobi',(0.0,0.0))


start = -7
end = 2
tTrue = 0.5
TrueExp = (-e**(-start*tTrue)+e**(-end*tTrue))/(tTrue*(start-end))

t = np.linspace(0,tTrue,2)

NMax = 25
Exp = np.zeros(NMax-1)
ExpVar = np.zeros(NMax-1)
for N in range(1,NMax):
    #Generate points
    x,w = JPol.GaussQuadrature(N)

    #Generate random representation
    x = (x+1)/2*(end-start)+start

    a = x
    a[x>end] = 0
    a[x<start] = 0


    #Solve for time
    u0 = u(0,a)
    sol = odeint(dudt,u0,t,tuple([a]))

    #Employ correct weights
    w = w/2
    Exp[N-1] = sum(w[:,0]*sol[-1])
    ExpVar[N-1] = sum(w[:,0]*sol[-1]**2)

plt.figure
NN = np.array(range(1,NMax))
plt.plot(NN,Exp,label="Approximated exp")
plt.plot(NN,np.ones(NMax-1)*TrueExp,label="True exp")
plt.xlabel("N")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/TestEQCollocationMeanUniform.eps")

plt.figure()
plt.semilogy(NN,abs(Exp-TrueExp),label="Error")
plt.xlabel("N")
plt.title("Absolute error")
plt.savefig("../../Latex/Billeder/Chapter5/TestEQCollocationMeanErrorUniform.eps"
```

Stochastic Galerkin method for the mean approximation

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 21 10:33:56 2013

@author: cbrams
"""
from __future__ import division
import DABISpectral1D as DB
import numpy as np
from math import e
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from scipy.misc import factorial

#True solution
def u(t,a):
    return e**(-a*t)

def dudt(u,t,A):
    return np.dot(A.T,u)

#Calculate eijk
def ef(i,j,k,N):
    M = int(np.ceil(3/2*N))
    ePol = DB.Poly1D('HermitePprob',())
    z,wz = ePol.GaussQuadrature(M)

    Hi = ePol.GradEvaluate(z,i,0)
    Hj = ePol.GradEvaluate(z,j,0)
    Hk = ePol.GradEvaluate(z,k,0)

    E = sum(Hi*Hj*Hk*wz)

    return E
#Initialize Polynomial
HPol = DB.Poly1D('HermitePprob',())

#Initialize computational parameters
tN = 1000
mean = 9
std = 3
tTrue = 0.5
TrueExp = e**(1/2*tTrue**2*std**2-tTrue*mean)
t = np.linspace(0,tTrue,tN)

NMax = 30
Exp = np.zeros(NMax-1)
for N in range(NMax):
    #Generate points
    x,w = HPol.GaussQuadrature(N)

    #Generate representation
    a = mean + (std)*x;

    #Transforming
    u0 = u(0,a)
    v0 = HPol.DiscretePolynomialTransform(x,u0,N)

    #Generating the A matrix
    A = np.zeros([N+1,N+1])
    for j in range(N+1):
        for k in range(N+1):
                A[j,k] = -1/factorial(k)*(mean*ef(0,j,k,N)+std*ef(1,j,k,N))

    #Solve for time
    sol = odeint(dudt,v0,t,tuple([A]))

    #Inversely transform
    ufinal = HPol.InverseDiscretePolynomialTransform(x,sol[-1],N)


    #Calculating expectancy
    Exp[N-1] = sum(w[:,0]*ufinal)

```

```
plt.figure
NN = np.array(range(1,NMax))
plt.plot(NN,Exp,label="Approximated exp")
plt.plot(NN,np.ones(NMax-1)*TrueExp,label="True exp")
plt.xlabel("N")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/TestEQGalerkinMean.eps")


plt.figure()
plt.semilogy(NN,abs(Exp-TrueExp),label="Error")
plt.xlabel("N")
plt.title("Absolute error")
plt.savefig("../../Latex/Billeder/Chapter5/TestEQGalerkinError.eps")
```

Two dimensional stochastic collocation method for the mean approximation

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 20 16:52:00 2013

@author: cbrams
"""

from __future__ import division
import DABISpectral1D as DB
import numpy as np
from math import e
from scipy.integrate import odeint
import matplotlib.pyplot as plt

#True solution
def u(t,a,b):
    return b*e**(-a*t)

def dudt(u,t,a):
    return -a*u
#Initialize Polynomial
HPol = DB.Poly1D('HermitePprob',())


```

```python
#Generate both gaussian distribution coefficients
tN = 1000
mean = 4
std = 1.5

meanB = 3
stdB = 0.15

#Calculate true expectancy
tTrue = 0.5
TrueExp = meanB*e**(1/2*tTrue**2*std**2-tTrue*mean)

t = np.linspace(0,tTrue,tN)

NMax = 25
Exp = np.zeros(NMax-1)
ExpVar = np.zeros(NMax-1)
for N in range(1,NMax):
    #Generate points
    x,xw = HPol.GaussQuadrature(N)
    y,yw = HPol.GaussQuadrature(N)

    #Calculate both stochastic representations
    a = mean + (std)*x;
    b = meanB+stdB*y

    #Create a mesh of both stochastic variables and weights
    A,B = np.meshgrid(a,b)
    XW,YW = np.meshgrid(xw,yw)
    W = XW*YW
    W = W.reshape([(N+1)*(N+1)])

    #Calculate starting conditions
    V = HPol.GradVandermonde1D(x,N,0)
    u0 = u(0,A,B)

    #Reshape solution
    u0 = u0.reshape([(N+1)*(N+1)])
    A = A.reshape([(N+1)*(N+1)])

    #Integrate over time
    sol = odeint(dudt,u0,t,tuple([A]))
    ufinal = sol[-1]

    #Calculate mean and expectancy
    Exp[N-1] = sum(W*ufinal)
    ExpVar[N-1] = sum(W*ufinal**2)

    ExpRunning = np.sum(np.tile(W,[tN,1])*sol,1)
    test = np.tile(ExpRunning,[(N+1)*(N+1),1]).T

    VarRunning = np.sum(np.tile(W,[tN,1])*(sol-test)**2,1)
plt.figure
NN = np.array(range(1,NMax))
plt.plot(NN,Exp,label="Approximated exp")
plt.plot(NN,np.ones(NMax-1)*TrueExp,label="True exp")
plt.xlabel("N")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/TestEq2DGPCApproxExp.eps")

plt.figure()
plt.semilogy(NN,abs(Exp-TrueExp),label="Error")
plt.xlabel("N")
plt.title("Absolute error")
plt.savefig("../../Latex/Billeder/Chapter5/TestEq2DGPCError.eps")

plt.figure()
plt.fill_between(t,ExpRunning-np.sqrt(VarRunning),ExpRunning+np.sqrt(VarRunning),
plt.plot(t,ExpRunning+2*np.sqrt(VarRunning),'r',label="Upper")
plt.plot(t,ExpRunning,label="Mean")
plt.plot(t,ExpRunning-2*np.sqrt(VarRunning),'r',label="Lower")
plt.xlabel("t")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/TestEq2DGPCSolution.eps")
plt.show()
```

E.2.2 Burgers' equation

Solution to Burgers' with uncertainty on v

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 12 17:44:27 2013

@author: cbrams
```

```python
"""

from __future__ import division
import numpy as np
import DABISpectral1D as DB
import matplotlib.pyplot as plt
from Diff import DP
from scipy.integrate import odeint


def initialValue(x):
    return -np.tanh(x*10)

def dudt(u,t,v,Dx,BoundaryFixer):

    unew= -u*np.dot(Dx,u)+v*np.dot(Dx,np.dot(Dx,u))
    unew = unew*BoundaryFixer
    return unew


LPol = DB.Poly1D("Jacobi",(0.0,0.0))

#Define constants
start = 0.05
end = 0.2

#Represent the random variable
zN = 10
z,Zw = LPol.GaussQuadrature(zN-1)
Zw = Zw/2
xN = 60
v =(z+1)/2*(end-start)+start

#Calculate spectral differential operators
Dx,x,Lw = DP(xN)
u0 = initialValue(x)

tEnd = 50
tNum = 1000

t = np.linspace(0,tEnd,tNum)

#Prepare system for solution by dudt
#Instead of running multiple times, we tile the system
```

```
Dx = np.kron(np.identity(zN),Dx)
u0 = np.tile(u0,zN)
xlol = np.tile(x,zN)

#We modify the boundary, so the correct points get zeroed
BDFix = np.ones(x.shape)
BDFix[0] = 0
BDFix[-1] = 0
BDFix = np.tile(BDFix,zN)
v0 = v.repeat(xN)

#Solve and reshape
usol = odeint(dudt,u0,t,tuple([v0,Dx,BDFix]))
usol = np.reshape(usol,(tNum,xN,zN),order='F')

#Calculate Exp, Var and Std
Exp = np.sum(usol[-1,:,:]*np.tile(Zw,[1,xN]).T,axis=1)
Var = np.sum((usol[-1,:,:]-np.tile(Exp,[zN,1]).T)**2*np.tile(Zw,[1,xN]).T,axis=1)
Std = np.sqrt(Var)

plt.figure()
plt.fill_between(x,Exp-np.sqrt(Var),Exp+np.sqrt(Var),color="yellow")
plt.plot(x,Exp+2*Std,marker="^",label="Upper",color="purple")
plt.plot(x,Exp,linewidth=3,label="Exp",color="red")
plt.plot(x,Exp-2*Std,marker="v",label="Low",color="purple")
plt.legend()
plt.savefig("../../Latex/Billeder/Chapter5/BurgersEQStochV.eps")
```

Solution to Burgers' with gaussian uncertainty on the boundary

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 13 14:10:46 2013

@author: cbrams
"""
import numpy as np
import DABISpectral1D as DB
import matplotlib.pyplot as plt
from Diff import DP
from scipy.integrate import odeint

```

```

def initialValue(x):
    return -np.tanh(x*10)

def dudt(u,t,v,Dx,BoundaryFixer):
    unew= -u*np.dot(Dx,u)+v*np.dot(Dx,np.dot(Dx,u))
    unew = unew*BoundaryFixer
    return unew



LPol = DB.Poly1D("Jacobi",(0.0,0.0))
HPol = DB.Poly1D("HermitePprob",())

#Preparing Gaussian variable
mean = 0.1
std = 0.05


#Preparing physical and stochastic grid
zN = 20
z,Hw = HPol.GaussQuadrature(zN-1)
xN = 50

delta = mean + z*std



Dx,x,Lw = DP(xN)

u0i = initialValue(x)

tEnd = 50
tNum = 1000

t = np.linspace(0,tEnd,tNum)

#Prepare system for solution by dudt
Dx = np.kron(np.identity(zN),Dx)
u0 = u0i.copy()
u0[u0>0] = u0[u0>0]*(1+delta[0])

#Generate different starting conditions
for i in range(1,zN):
    u0temp = u0i.copy()
    u0temp[u0temp>0] = u0temp[u0temp>0]*(1+delta[i])
    u0 = np.concatenate((u0,u0temp))

#Boundary conditions
v0 = 0.05
BDFix = np.ones(x.shape)
BDFix[0] = 0
BDFix[-1] = 0
BDFix = np.tile(BDFix,zN)
#v0 = v.repeat(xN)

#Solve and reshape
usol = odeint(dudt,u0,t,tuple([v0,Dx,BDFix]))
usol = np.reshape(usol,(tNum,xN,zN),order='F')

ufin = usol[-1]

#Preparing to calculate exp
Hw = np.tile(Hw,(1,xN))
Hw = Hw.T


#Calculate exp, var and std
Exp = np.sum(Hw*ufin,axis=1)
ExpTile = np.tile(Exp,(zN,1))
ExpTile = ExpTile.T

VarCalc = ufin-ExpTile

Var =  np.sum(Hw*(VarCalc)**2,axis=1)
Std = np.sqrt(Var)

plt.figure()
plt.fill_between(x,Exp-Std,Exp+Std,color="yellow")
plt.plot(x,ufin[:,-1],'b:',label="Highest, e=%.2f"%delta[-1])
plt.plot(x,Exp+2*Std,'^-',color="purple",label="Upper",linewidth=2)
plt.plot(x,Exp,'r-',label="Expectation",linewidth=2)
plt.plot(x,Exp-2*Std,'v-',color="purple",label="Lower",linewidth=2)
plt.plot(x,ufin[:,0],'g:',label="Lowest, e=%.2f"%delta[0])
plt.legend(loc=3)
plt.savefig("../../Latex/Billeder/Chapter5/BurgersEQStochEGauss.eps")
```

Solution to Burgers' with uniform uncertainty on the boundary

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 21 13:34:35 2013

@author: cbrams
"""
from __future__ import division
import numpy as np
import DABISpectral1D as DB
import matplotlib.pyplot as plt
from Diff import DP
from scipy.integrate import odeint


def initialValue(x):
    return -np.tanh(x*10)

def dudt(u,t,v,Dx,BoundaryFixer):
    unew= -u*np.dot(Dx,u)+v*np.dot(Dx,np.dot(Dx,u))
    unew = unew*BoundaryFixer
    return unew


#Preparing for modelling of random coefficients
LPol = DB.Poly1D("Jacobi",(0.0,0.0))

start = 0.
end = 0.1

zN = 20
z,Zw = LPol.GaussQuadrature(zN-1)
Zw = Zw/2
xN = 50

delta = (z+1)/2*(end-start)+start


#Calculate for the physical domain as well
Dx,x,Lw = DP(xN)

u0i = initialValue(x)
```

```python

tEnd = 50
tNum = 1000

t = np.linspace(0,tEnd,tNum)

#Prepare system for solution by dudt
Dx = np.kron(np.identity(zN),Dx)
u0 = u0i.copy()
u0[u0>0] = u0[u0>0]*(1+delta[0])

for i in range(1,zN):
    u0temp = u0i.copy()
    u0temp[u0temp>0] = u0temp[u0temp>0]*(1+delta[i])
    u0 = np.concatenate((u0,u0temp))


v0 = 0.05

BDFix = np.ones(x.shape)
BDFix[0] = 0
BDFix[-1] = 0
BDFix = np.tile(BDFix,zN)
#v0 = v.repeat(xN)
usol = odeint(dudt,u0,t,tuple([v0,Dx,BDFix]))
usol = np.reshape(usol,(tNum,xN,zN),order='F')

ufin = usol[-1]



#Adjusting weigts for quadrature
Zw = np.tile(Zw,(1,xN))
Zw = Zw.T

Exp = np.sum(Zw*ufin,axis=1)
ExpTile = np.tile(Exp,(zN,1))
ExpTile = ExpTile.T

VarCalc = ufin-ExpTile

Var =  np.sum((Zw*VarCalc)**2,axis=1)
Std = np.sqrt(Var)

```

```
plt.figure()
plt.fill_between(x,Exp-Std,Exp+Std,color="yellow")
plt.plot(x,ufin[:,-1],'b:',label="Highest, e=%.2f"%delta[-1])
plt.plot(x,Exp+2*Std,'^-',color="purple",label="Upper",linewidth=2)
plt.plot(x,Exp,'r-',label="Expectation",linewidth=2)
plt.plot(x,Exp-2*Std,'v-',color="purple",label="Lower",linewidth=2)
plt.plot(x,ufin[:,0],'g:',label="Lowest, e=%.2f"%delta[0])
plt.legend(loc=3)
plt.savefig("../../Latex/Billeder/Chapter5/BurgersEQStochEUniform.eps")
```

Appendix F

Code used in the lid driven cavity problem

F.1 Approximating solution

Approximating Ghia et al

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 26 11:04:08 2013

@author: cbrams

This script is intended to approximate a steady state from
scratch for the lid driven cavity problem.
"""

from __future__ import division
import numpy as np
import DABISpectral1D as DB
from Diff import DP
import scipy.sparse as sp
```

```python


def dTime(inputVec,t,beta2,Re,Nx,Ny,DX,DY,V1,VInv,DXP,DYP,index):
    #Split the input
    u = inputVec[:(Nx+1)*(Ny+1)]
    v = inputVec[(Nx+1)*(Ny+1):(Nx+1)*(Ny+1)*2]
    p = inputVec[(Nx+1)*(Ny+1)*2:]

    #Generate inner differentials
    pxd = DXP.dot(p)
    pyd = DYP.dot(p)

    #Transform inner differentials
    pxd = V1.dot(VInv.dot(pxd))
    pyd = V1.dot(VInv.dot(pyd))

    #Calculate outer differentials
    ux = DX.dot(u)
    uy = DY.dot(u)
    vx = DX.dot(v)
    vy = DY.dot(v)

    #Calculate outer double differentials
    uxx = DX.dot(ux)
    vyy = DY.dot(vy)
    uyy = DY.dot(uy)
    vxx = DX.dot(vx)

    #Calculate the change en pressure
    dpdt = -beta2*(ux+vy)
    dpdt = dpdt[index[1:-1,1:-1]].flatten("F")

    #Calculate the change in velocities
    dudt = -(u*ux+v*uy)-pxd+Re**(-1)*(uxx+uyy)
    dvdt = -(u*vx+v*vy)-pyd+Re**(-1)*(vxx+vyy)


    outputVec = np.zeros((Nx+1)*(Ny+1)*2+(Nx-1)*(Ny-1))

    #Impose boundary conditions
    dudt[index[0,:]] = 0
    dudt[index[-1,:]] = 0
    dudt[index[:,0]] = 0
    dudt[index[:,-1]] = 0

    dvdt[index[0,:]] = 0
    dvdt[index[-1,:]] = 0
    dvdt[index[:,0]] = 0
    dvdt[index[:,-1]] = 0

    #Collocting output
    outputVec[:(Nx+1)*(Ny+1)] = dudt
    outputVec[(Nx+1)*(Ny+1):(Nx+1)*(Ny+1)*2] = dvdt
    outputVec[(Nx+1)*(Ny+1)*2:] = dpdt


    return outputVec

#Creating mesh
LegPol = DB.Poly1D(DB.JACOBI,(0.0,0.0))
Nx = 35
Ny = Nx
x,wx = LegPol.GaussLobattoQuadrature(Nx)
y,wy = LegPol.GaussLobattoQuadrature(Ny)
X,Y = np.meshgrid(x,y)

#Creating mesh of real values
xreal = (x+1)/2
yreal = (y+1)/2
XREAL,YREAL = np.meshgrid(xreal,yreal)

#Creater inner meshes
xp = x[1:-1]
yp = y[1:-1]
xpreal = xreal[1:-1]
ypreal = yreal[1:-1]

XPREAL,YPREAL = np.meshgrid(xpreal,ypreal)

#Create index functions
index = np.arange((Nx+1)*(Ny+1)).reshape(Ny+1,Nx+1,\
order='F').copy()
indexP = np.arange((Nx-1)*(Ny-1)).reshape(Ny-1,Nx-1,\
order='F').copy()

#Initial conditions
u = np.zeros(X.shape)
v = np.zeros(X.shape)
```

```
p = np.ones(XPREAL.shape)


#Flatten the matrices
u = u.flatten("F")
v = v.flatten("F")
p = p.flatten("F")

#Initial condition
u[index[-1,:]] = 1

#Generate and scale differential operators
[Dx,_,_] = DP(Nx+1)
[Dy,_,_] = DP(Ny+1)
Dx = 2*Dx
Dy = 2*Dy

DY = np.kron(np.identity(Nx+1),Dy)
DX = np.kron(Dx,np.identity(Ny+1))

#Sparsifying matrices
DY = sp.csr_matrix(DY)
DX = sp.csr_matrix(DX)

#Scale inner grid
cx = xp[-1]
cy = yp[-1]

xp = xp/cx
yp = yp/cy

XP,YP = np.meshgrid(xp,yp)

#Calculate Vandermondes for transformation and
#differentiation
Vxp = LegPol.GradVandermonde1D(xp,Nx-2,0)
Vyp = LegPol.GradVandermonde1D(yp,Ny-2,0)

V = np.kron(Vxp,Vyp)

VxpD = LegPol.GradVandermonde1D(xp,Nx-2,1)
VypD = LegPol.GradVandermonde1D(yp,Ny-2,1)

#Calculate and scale inner differential matrices
```

```
Dxp = np.linalg.solve(Vxp.T,VxpD.T).T
Dyp = np.linalg.solve(Vyp.T,VypD.T).T
Dxp = 1/cx*2*Dxp
Dyp = 1/cy*2*Dyp

DXP = np.kron(Dxp,np.identity(Ny-1))
DYP = np.kron(np.identity(Nx-1),Dyp)

#Sparsify
DXP = sp.csr_matrix(DXP)
DYP = sp.csr_matrix(DYP)


#Creating final transformation matrices
Vx1 = LegPol.GradVandermonde1D(x*1/cx,Nx-2,0)
Vy1 = LegPol.GradVandermonde1D(y*1/cy,Ny-2,0)
V1 = np.kron(Vx1,Vy1)
V1 = sp.csr_matrix(V1)

VInv= np.linalg.inv(V)
VInv = sp.csr_matrix(VInv)


"""
TEST FOR DIFFERENTIATION

T1_FULLF = np.cos(XREAL)*np.sin(YREAL)
T1_FULLSOL = -np.sin(XREAL)*np.cos(YREAL)

T1_INNERF = np.cos(XPREAL)*np.sin(YPREAL)
T1_INNERSOL = -np.sin(XPREAL)*np.cos(YPREAL)

T1_FULLAPPROX = DX.dot(DY.dot(T1_FULLF.flatten("F")))
T1_INNERAPPROX = DXP.dot(DYP.dot(T1_INNERF.flatten("F")))

T1_FULLERROR = np.abs(T1_FULLSOL - \
T1_FULLAPPROX[index[:,:]])
T1_INNERERROR = np.abs(T1_INNERSOL - \
T1_INNERAPPROX[indexP[:,:]])

plt.figure()
plt.title("Differentiation test, error for full grid")
plt.imshow(T1_FULLERROR,origin="lower",extent=[0,1,0,1])
plt.colorbar()
plt.xlabel('x')
plt.ylabel('y')
plt.axis('normal')
plt.savefig("../../Latex/Billeder/LidDrivenCavity/\
DiffTestFullError.eps")

plt.figure()
plt.title("Differentiation test, error for inner grid")
plt.imshow(T1_INNERERROR,origin="lower",extent=[0,1,0,1])
plt.colorbar()
plt.xlabel('x')
plt.ylabel('y')
plt.axis('normal')
plt.savefig("../../Latex/Billeder/LidDrivenCavity/\
DiffTestInnerError.eps")

END OF TEST FOR DIFFERENTIATION
"""

"""
TEST FOR INTERPOLATION

T2_INNER = np.exp(XPREAL*YPREAL)
T2_OUTERTRUE = np.exp(XREAL*YREAL)
T2_OUTERAPPROX = V1.dot(VInv.dot(T2_INNER.flatten("F")))

plt.figure()
plt.title("Interpolation test, error")
plt.imshow(T2_OUTERAPPROX[index[:,:]]-T2_OUTERTRUE,\
origin="lower",extent=[0,1,0,1])
plt.colorbar()
plt.xlabel('x')
plt.ylabel('y')
plt.axis('normal')
plt.savefig("../../Latex/Billeder/\
LidDrivenCavity/IntTestError.eps")

END OF TEST FOR INTERPOLATION
"""

#Gatherring input
inputVec = np.zeros((Nx+1)*(Ny+1)*2+(Nx-1)*(Ny-1))
inputVec[:(Nx+1)*(Ny+1)] = u
inputVec[(Nx+1)*(Ny+1):(Nx+1)*(Ny+1)*2] = v
```

```
inputVec[(Nx+1)*(Ny+1)*2:] = p


"""
Start Reproducing Ghia et Al results
"""

#Iterate over different values of Re
nG = 3
R = np.array([100,400,1000])

totalU = np.zeros([u.size,nG])
totalV = np.zeros([v.size,nG])
totalP = np.zeros([p.size,nG])

initialVec = inputVec.copy()
for iG in range(nG):
    Re = R[iG]
    """
    Start time-derivative calculation
    """
    beta2 = 5
    tend = 200
    t = 0
    t0 = 0

    #Calculating time-steps
    CFL = 0.5
    deltax = np.min(np.abs(x[1:]-x[:-1]))
    deltay = np.min(np.abs(y[1:]-y[:-1]))
    lambdax = (np.abs(np.max(u))+np.sqrt(np.max(u)**2+beta2))/deltax \
    + 1/(Re*deltax**2)
    lambday = (np.abs(np.max(v))+np.sqrt(np.max(v)**2+beta2))/deltay \
    + 1/(Re*deltay**2)

    inputVec = initialVec.copy()
    tstep = CFL/(lambdax+lambday)

    #Time each iteration for comparison
    import time
    T = time.clock()
    while t<=tend:
        #Initiate Runge-Kutta
        sol1 = inputVec + 1/4*tstep*dTime(inputVec,t,beta2,Re,Nx,Ny\
        ,DX,DY,V1,VInv,DXP,DYP,index)
        sol2 = inputVec + 1/3*tstep*dTime(sol1,t,beta2,Re,Nx,Ny,DX,\
        DY,V1,VInv,DXP,DYP,index)
        sol3 = inputVec + 1/2*tstep*dTime(sol2,t,beta2,Re,Nx,Ny,DX,\
        DY,V1,VInv,DXP,DYP,index)

        #Calculate new vector
        inputVec = inputVec + tstep*dTime(sol3,t,beta2,Re,Nx,Ny,DX,\
        DY,V1,VInv,DXP,DYP,index)
        #Save old result
        uold = u.copy()
        vold = v.copy()
        pold = p.copy()

        u = inputVec[:(Nx+1)*(Ny+1)]
        v = inputVec[(Nx+1)*(Ny+1):(Nx+1)*(Ny+1)*2]
        p = inputVec[(Nx+1)*(Ny+1)*2:]

        #Calculate test-values
        testValu = np.linalg.norm(u-uold,ord=2)\
        /np.linalg.norm(uold,ord=2)
        #testValv = np.linalg.norm(v-vold,ord=2)\
        #/np.linalg.norm(vold,ord=2)
        #testValp = np.linalg.norm(p-pold,ord=2)\
        #/np.linalg.norm(pold,ord=2)

        #Break if satisfyingly stable
        if testValu<1e-6:
            print t
            break

        #Recalculate time-steps
        lambdax = (np.abs(np.max(u))+np.sqrt(np.max(u)**2\
        +beta2))/deltax + 1/(Re*deltax**2)
        lambday = (np.abs(np.max(v))+np.sqrt(np.max(v)**2\
        +beta2))/deltay + 1/(Re*deltay**2)
        tstep = CFL/(lambdax+lambday)

        #Break if time-step becomes too small
        if tstep < 1e-15 or np.isnan(tstep):
            print t
            print tstep
            break

            #Time step forward
            t = t + tstep
            t0 = t0 + 1

        #Print elapsed time
        T2 = time.clock()

        print T2-T

        #Save results
        totalU[:,iG] = inputVec[:(Nx+1)*(Ny+1)]
        totalV[:,iG] = inputVec[(Nx+1)*(Ny+1):(Nx+1)*(Ny+1)*2]
        totalP[:,iG] = inputVec[(Nx+1)*(Ny+1)*2:]


#Save results for another script to plot
np.savez("GhiaEtAlApprox",totalU=totalU,totalV=totalV,\
totalP=totalP,R=R,Nx=Nx,Ny=Ny,index=index,indexP=indexP,\
XREAL=XREAL,YREAL=YREAL,XPREAL=XPREAL,YPREAL=YPREAL)
```

Example of stochastic collocation

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 28 14:43:36 2013

@author: cbrams

This script is using a stochastic collocation method to
calculate the mean and spread for uncertainty on Re

It will be the only of the 8 used scripts present in the appendix
since the others are simply duplicates of this with the
uniform distribution and the initial condition changed.
"""
from __future__ import division
import numpy as np
import DABISpectral1D as DB
from Diff import DP
import scipy.sparse as sp


```

```
def dTime(inputVec,t,beta2,Re,Nx,Ny,DX,DY,V1,VInv,DXP,DYP,index):
    #Split the input
    u = inputVec[:(Nx+1)*(Ny+1)]
    v = inputVec[(Nx+1)*(Ny+1):(Nx+1)*(Ny+1)*2]
    p = inputVec[(Nx+1)*(Ny+1)*2:]

    #Generate inner differentials
    pxd = DXP.dot(p)
    pyd = DYP.dot(p)

    #Transform inner differentials
    pxd = V1.dot(VInv.dot(pxd))
    pyd = V1.dot(VInv.dot(pyd))

    #Calculate outer differentials
    ux = DX.dot(u)
    uy = DY.dot(u)
    vx = DX.dot(v)
    vy = DY.dot(v)

    #Calculate outer double differentials
    uxx = DX.dot(ux)
    vyy = DY.dot(vy)
    uyy = DY.dot(uy)
    vxx = DX.dot(vx)

    #Calculate the change en pressure
    dpdt = -beta2*(ux+vy)
    dpdt = dpdt[index[1:-1,1:-1]].flatten("F")

    #Calculate the change in velocities
    dudt = -(u*ux+v*uy)-pxd+Re**(-1)*(uxx+uyy)
    dvdt = -(u*vx+v*vy)-pyd+Re**(-1)*(vxx+vyy)


    outputVec = np.zeros((Nx+1)*(Ny+1)*2+(Nx-1)*(Ny-1))

    #Impose boundary conditions
    dudt[index[0,:]] = 0
    dudt[index[-1,:]] = 0
    dudt[index[:,0]] = 0
    dudt[index[:,-1]] = 0

    dvdt[index[0,:]] = 0
    dvdt[index[-1,:]] = 0
    dvdt[index[:,0]] = 0
    dvdt[index[:,-1]] = 0

    #Collocting output
    outputVec[:(Nx+1)*(Ny+1)] = dudt
    outputVec[(Nx+1)*(Ny+1):(Nx+1)*(Ny+1)*2] = dvdt
    outputVec[(Nx+1)*(Ny+1)*2:] = dpdt


    return outputVec

#Creating mesh
LegPol = DB.Poly1D(DB.JACOBI,(0.0,0.0))
Nx = 35
Ny = Nx
x,wx = LegPol.GaussLobattoQuadrature(Nx)
y,wy = LegPol.GaussLobattoQuadrature(Ny)
X,Y = np.meshgrid(x,y)

#Creating mesh of real values
xreal = (x+1)/2
yreal = (y+1)/2
XREAL,YREAL = np.meshgrid(xreal,yreal)

#Creater inner meshes
xp = x[1:-1]
yp = y[1:-1]
xpreal = xreal[1:-1]
ypreal = yreal[1:-1]

XPREAL,YPREAL = np.meshgrid(xpreal,ypreal)

#Create index functions
index = np.arange((Nx+1)*(Ny+1)).reshape(Ny+1,Nx+1,\
order='F').copy()
indexP = np.arange((Nx-1)*(Ny-1)).reshape(Ny-1,Nx-1,\
order='F').copy()

#Initial conditions
u = np.zeros(X.shape)
v = np.zeros(X.shape)
p = np.ones(XPREAL.shape)

```

```

#Flatten the matrices
u = u.flatten("F")
v = v.flatten("F")
p = p.flatten("F")

#Initial condition
u[index[-1,:]] = 1

#Generate and scale differential operators
[Dx,_,_] = DP(Nx+1)
[Dy,_,_] = DP(Ny+1)
Dx = 2*Dx
Dy = 2*Dy

DY = np.kron(np.identity(Nx+1),Dy)
DX = np.kron(Dx,np.identity(Ny+1))

#Sparsifying matrices
DY = sp.csr_matrix(DY)
DX = sp.csr_matrix(DX)

#Scale inner grid
cx = xp[-1]
cy = yp[-1]

xp = xp/cx
yp = yp/cy

XP,YP = np.meshgrid(xp,yp)

#Calculate Vandermondes for transformation and
#differentiation
Vxp = LegPol.GradVandermonde1D(xp,Nx-2,0)
Vyp = LegPol.GradVandermonde1D(yp,Ny-2,0)

V = np.kron(Vxp,Vyp)

VxpD = LegPol.GradVandermonde1D(xp,Nx-2,1)
VypD = LegPol.GradVandermonde1D(yp,Ny-2,1)

#Calculate and scale inner differential matrices
Dxp = np.linalg.solve(Vxp.T,VxpD.T).T
Dyp = np.linalg.solve(Vyp.T,VypD.T).T
```

```
Dxp = 1/cx*2*Dxp
Dyp = 1/cy*2*Dyp

DXP = np.kron(Dxp,np.identity(Ny-1))
DYP = np.kron(np.identity(Nx-1),Dyp)

#Sparsify
DXP = sp.csr_matrix(DXP)
DYP = sp.csr_matrix(DYP)


#Creating final transformation matrices
Vx1 = LegPol.GradVandermonde1D(x*1/cx,Nx-2,0)
Vy1 = LegPol.GradVandermonde1D(y*1/cy,Ny-2,0)
V1 = np.kron(Vx1,Vy1)
V1 = sp.csr_matrix(V1)

VInv= np.linalg.inv(V)
VInv = sp.csr_matrix(VInv)



#Loading values from Ghia et al approximation
CalcValues = np.load("GhiaEtAlApprox.npz")
u = CalcValues['totalU'][:,0]
v = CalcValues['totalV'][:,0]
p = CalcValues['totalP'][:,0]

#Gatherring input
inputVec = np.zeros((Nx+1)*(Ny+1)*2+(Nx-1)*(Ny-1))
inputVec[:(Nx+1)*(Ny+1)] = u
inputVec[(Nx+1)*(Ny+1):(Nx+1)*(Ny+1)*2] = v
inputVec[(Nx+1)*(Ny+1)*2:] = p


"""
Start Uncertainty Quantification
"""

#Defining the rangeof uncertainty
JPol = DB.Poly1D(DB.JACOBI,(0.0,0.0))
start = 95
end = 105

```

```python
#Number of iterations
nUQ = 10

#Calculate grid and weights
R,Rw = JPol.GaussLobattoQuadrature(nUQ-1)
Rw = Rw/2

#Tile weights for later use
oW = np.tile(Rw,[u.size,1])
iW = np.tile(Rw,[p.size,1])

#Generate space for each iteration
totalU = np.zeros([u.size,nUQ])
totalV = np.zeros([v.size,nUQ])
totalP = np.zeros([p.size,nUQ])

#Iterate over deterministic solver
for iUQ in range(nUQ):
    Re = (R[iUQ]+1)/2*(end-start)+start
    """
    Start time-derivative calculation
    """
    beta2 = 5
    tend = 100
    t = 0
    t0 = 0

    #Calculate time-steps
    CFL = 0.5
    deltax = np.min(np.abs(x[1:]-x[:-1]))
    deltay = np.min(np.abs(y[1:]-y[:-1]))
    lambdax = (np.abs(np.max(u))+np.sqrt(np.max(u)\
    **2+beta2))/deltax + 1/(Re*deltax**2)

    lambday = (np.abs(np.max(v))+np.sqrt(np.max(v)\
    **2+beta2))/deltay + 1/(Re*deltay**2)

    tstep = CFL/(lambdax+lambday)
    #Start clock
    import time
    T = time.clock()
    while t<=tend:

        #Runge-Kutta iteration
        sol1 = inputVec + 1/4*tstep*dTime(inputVec,t,beta2,Re\
        ,Nx,Ny,DX,DY,V1,VInv,DXP,DYP,index)

        sol2 = inputVec + 1/3*tstep*dTime(sol1,t,beta2,Re,Nx,\
        Ny,DX,DY,V1,VInv,DXP,DYP,index)

        sol3 = inputVec + 1/2*tstep*dTime(sol2,t,beta2,Re,Nx,Ny,\
        DX,DY,V1,VInv,DXP,DYP,index)

        inputVec = inputVec + tstep*dTime(sol3,t,beta2,Re,Nx,Ny,\
        DX,DY,V1,VInv,DXP,DYP,index)


        u_old = u.copy()

        u = inputVec[:(Nx+1)*(Ny+1)]
        v = inputVec[(Nx+1)*(Ny+1):(Nx+1)*(Ny+1)*2]
        p = inputVec[(Nx+1)*(Ny+1)*2:]


        #Test for satisfaction
        testVal = np.linalg.norm(u-u_old,ord=2)/\
        np.linalg.norm(u_old,ord=2)

        if testVal<1e-6:
            print t
            break


        #Recalculate time-steps
        lambdax = (np.abs(np.max(u))+np.sqrt(np.max(u)**2+beta2))\
        /deltax + 1/(Re*deltax**2)

        lambday = (np.abs(np.max(v))+np.sqrt(np.max(v)**2+beta2))\
        /deltay + 1/(Re*deltay**2)

        tstep = CFL/(lambdax+lambday)
        if tstep < 1e-16 or np.isnan(tstep):
            print t
            print tstep
            break

        t = t + tstep
        t0 = t0 + 1
```

```

    T2 = time.clock()

    print T2-T


    #Save the output for each iteration
    totalU[:,iUQ] = inputVec[:(Nx+1)*(Ny+1)]
    totalV[:,iUQ] = inputVec[(Nx+1)*(Ny+1):(Nx+1)*(Ny+1)*2]
    totalP[:,iUQ] = inputVec[(Nx+1)*(Ny+1)*2:]


#Calculate mean
meanU = np.sum(totalU*oW,axis=1)
meanV = np.sum(totalV*oW,axis=1)
meanP = np.sum(totalP*iW,axis=1)

#Tile mean for calculation of variance
meanUT = np.tile(meanU,[nUQ,1]).T
meanVT = np.tile(meanV,[nUQ,1]).T
meanPT = np.tile(meanP,[nUQ,1]).T

#Calculate variance
varU = np.sum(((totalU-meanUT))**2*oW,axis=1)
varV = np.sum(((totalV-meanVT))**2*oW,axis=1)
varP = np.sum(((totalP-meanPT))**2*iW,axis=1)

stdU = np.sqrt(varU)
stdV = np.sqrt(varV)
stdP = np.sqrt(varP)

#Save data for display
np.savez("UQRe100",start = start,end = end,meanU = meanU,meanV = meanV,meanP = meanP,
```

F.2 Visualizing

Visualizig Ghia et al

```
# -*- coding: utf-8 -*-
"""
```

```
Created on Sat Jan 26 11:57:08 2013

@author: cbrams

This script is intended to simply plot the values calculated when
approximating the results of Ghia Et al.
"""

import numpy as np
import matplotlib.pyplot as plt
import DABISpectral1D as DB
import scipy.sparse as sp

"""
GHIA ET AL VALUES
"""

TESTY = np.array([0.0000, 0.0547, 0.0625, 0.0703,\
 0.1016, 0.1719, 0.2813,\
0.4531, 0.500,0.6172, 0.7344, 0.8516, 0.9531, 0.9609,\
 0.9688, 0.9766, 1.0000])

TESTU100 = np.array([0.0000, -0.03717, -0.04192,\
 -0.04775, -0.06434, -0.10150,\
-0.15662, -0.21090,-0.20581, -0.13641, 0.00332, \
0.23151, 0.68717, 0.73722,\
0.78871, 0.84123, 1.00000])

TESTU400 = np.array([0,-0.08186,-0.09266,-0.10338,\
-0.14612,-0.24299,-0.32726,\
-0.17119,-0.11477,0.02135,0.16256,0.29093,0.55892,\
0.61756,0.68439,0.75837,1])

TESTU1000 = np.array([0,-0.18109,-0.20196,-0.22220,\
-0.29730,-0.38289,-0.27805,\
-0.10648,-0.06080,0.05702,0.18719,0.33304,0.46604,\
0.51117,0.57492,0.65928,1])


"""
END GHIA ET AL VALUES
"""

#Load previously calculated values
```

```python
CalcValues = np.load("GhiaEtAlApprox.npz")

u100 = CalcValues['totalU'][:,0]
u400 = CalcValues['totalU'][:,1]
u1000 = CalcValues['totalU'][:,2]
index = CalcValues['index']
YREAL = CalcValues['YREAL']
Nx = CalcValues['Nx']

"""
START COMPARING VELOCITY PROFILES
"""


plt.figure()
plt.plot(u100[index[:,int(Nx/2)]],YREAL[:,int(Nx/2)],label="Approx")
plt.plot(TESTU100,TESTY,'ro',label="Ghia et al")
plt.legend()
plt.xlabel("u")
plt.ylabel("y")
plt.title("Velocity profile of u along middle axis for Re=100")
plt.savefig("../../Latex/Billeder/LidDrivenCavity/Ghia100.eps")


plt.figure()
plt.plot(u400[index[:,int(Nx/2)]],YREAL[:,int(Nx/2)],label="Approx")
plt.plot(TESTU400,TESTY,'ro',label="Ghia et al")
plt.legend()
plt.xlabel("u")
plt.ylabel("y")
plt.title("Velocity profile of u along middle axis for Re=400")
plt.savefig("../../Latex/Billeder/LidDrivenCavity/Ghia400.eps")

plt.figure()
plt.plot(u1000[index[:,int(Nx/2)]],YREAL[:,int(Nx/2)],label="Approx")
plt.plot(TESTU1000,TESTY,'ro',label="Ghia et al")
plt.legend()
plt.xlabel("u")
plt.ylabel("y")
plt.title("Velocity profile of u along middle axis for Re=1000")
plt.savefig("../../Latex/Billeder/LidDrivenCavity/Ghia1000.eps")


"""
```

```
END COMPARING VELOCITY PROFILES
"""


"""
Plot streamplots
"""
v100 = CalcValues['totalV'][:,0]
v400 = CalcValues['totalV'][:,1]
v1000 = CalcValues['totalV'][:,2]


#Calculate transformation matrices
EQUI = np.linspace(-1,1,Nx+1)
EQUIREAL = np.linspace(0,1,Nx+1)
LegPol = DB.Poly1D(DB.JACOBI,(0.0,0.0))
Ny = CalcValues['Ny']
x,wx = LegPol.GaussLobattoQuadrature(Nx)
y,wy = LegPol.GaussLobattoQuadrature(Ny)
SVx = LegPol.GradVandermonde1D(x,Nx,0)
SVy = LegPol.GradVandermonde1D(y,Ny,0)

SV = np.kron(SVx,SVy)
SVInv = np.linalg.inv(SV)
SVInv = sp.csr_matrix(SVInv)

SV2x = LegPol.GradVandermonde1D(EQUI,Nx,0)
SV2y = LegPol.GradVandermonde1D(EQUI,Ny,0)
SV2 = np.kron(SV2x,SV2y)
SV2 = sp.csr_matrix(SV2)


#Transform the data
SU100 = SV2.dot(SVInv.dot(u100))
SV100 = SV2.dot(SVInv.dot(v100))

SU400 = SV2.dot(SVInv.dot(u400))
SV400 = SV2.dot(SVInv.dot(v400))

SU1000 = SV2.dot(SVInv.dot(u1000))
SV1000 = SV2.dot(SVInv.dot(v1000))


#Plot the streamplots
```

```
plt.figure()
plt.streamplot(EQUIREAL,EQUIREAL,SU100[index[:,:]],SV100[index[:,:]])
plt.title("Stream plot for Re=100")
plt.xlabel("x")
plt.ylabel("y")
plt.savefig("../../Latex/Billeder/LidDrivenCavity/GhiaStream100.eps")

plt.figure()
plt.streamplot(EQUIREAL,EQUIREAL,SU400[index[:,:]],SV400[index[:,:]])
plt.title("Stream plot for Re=400")
plt.xlabel("x")
plt.ylabel("y")
plt.savefig("../../Latex/Billeder/LidDrivenCavity/GhiaStream400.eps")

plt.figure()
plt.streamplot(EQUIREAL,EQUIREAL,SU1000[index[:,:]],SV1000[index[:,:]])
plt.title("Stream plot for Re=1000")
plt.xlabel("x")
plt.ylabel("y")
plt.savefig("../../Latex/Billeder/LidDrivenCavity/GhiaStream1000.eps")


plt.show()
```

Example of visualizing uncertainty from stochastic collocation

Appendix G

Code used in the 2D wave tank problem

G.1 Time-integration functions

The RHS function

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 30 12:52:54 2013

@author: cbrams

This script is merely here to make it easier to comprehend what the
time-stepping function does
"""

#Initialization
from __future__ import division
import numpy as np
from laplacefinitediffFunSparse import laplace_finite_diffFun

```

```python
#Function
def dTime_finite(tmp,t,WLAST,tstep,CalcPressure,Nx,Ny,Dx,Dsigma,\
DX,DSIGMA):

    #Calculate N
    N = int(tmp.shape[0]/2)

    #Split array
    phi = tmp[:N]
    eta = tmp[N:]

    #Calculate dfuns
    dphi,deta,W2Z,p,Force= laplace_finite_diffFun(phi,eta,WLAST,\
    tstep,CalcPressure,Nx,Ny,Dx,Dsigma,DX,DSIGMA)

    #Join array
    res = np.zeros(N*2)
    res[:N] = dphi
    res[N:] = deta
    return res,W2Z,p,Force
```

The Laplacian solver

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 29 17:54:35 2013

@author: cbrams

This is a script for computing the Laplacian for the Wave model
and the associated values (Force, time-derivatives)
"""
#Initialization
from __future__ import division
import numpy as np
import DABISpectral1D
import DABISpectral1D as DB
import scipy.sparse as sp
import scipy.sparse.linalg as spl

def laplace_finite_diffFun(phi,zeta,WLAST,tstep,CalcPressure,\
Nx,Ny,Dx,Dsigma,DX,DSIGMA):
    h = 2
    g = 9.82    #Gravitational acceleration

    #Storing phi as u0
    u0 = phi

    #Initializing quadratures
    polyLeg = DABISpectral1D.Poly1D(DABISpectral1D.JACOBI, (0.0,0.0))
    (y,w) = polyLeg.GaussLobattoQuadrature(Ny-1)
    (x,wx) = polyLeg.GaussLobattoQuadrature(Nx-1)
    X , SIGMA = np.meshgrid((x+1)/2,(y+1)/2)

    sigma = (y+1)/2
    #Calculating and diagonalising constants
    dZeta = np.dot(Dx,zeta)
    ddZeta = np.dot(Dx,dZeta)
    d = zeta + h

    Dsigmadx = np.zeros((Ny,Nx))
    DDsigmadxx = np.zeros((Ny,Nx))
    DsigmadZ = np.zeros((Ny,Nx))
    for i in range(Nx):
        d = zeta[i]+h
        for j in range(Ny):
            Dsigmadx[j,i] = -SIGMA[j,i] * dZeta[i]/d

            DDsigmadxx[j,i] = -SIGMA[j,i]/d *(ddZeta[i]-\
            (dZeta[i])**2/d)-Dsigmadx[j,i]/d*dZeta[i]

            DsigmadZ[j,i] = 1/d
    DSIGMADX =  np.diag(Dsigmadx.flatten(1))
    DDSIGMADXX =  np.diag(DDsigmadxx.flatten(1))
    DSIGMADZ =  np.diag(DsigmadZ.flatten(1))

    DSIGMADX = sp.csr_matrix(DSIGMADX)
    DDSIGMADXX = sp.csr_matrix(DDSIGMADXX)
    DSIGMADZ = sp.csr_matrix(DSIGMADZ)

    #Calculating L
    DTEMP = DSIGMADX**2 + DSIGMADZ**2
    Lopperator = np.dot(DX,DX) + np.dot(DDSIGMADXX,DSIGMA) \
    + 2*np.dot(DSIGMADX,np.dot(DX,DSIGMA)) + np.dot(DTEMP,np.dot(DSIGMA,DSIGMA))

    #Constructing f
    f = np.zeros(Nx*Ny)

    #Imposing boundary conditions
    index = np.arange(Nx*Ny).reshape(Ny,Nx,order='F').copy()
    LBCsigma = np.dot(DSIGMADZ,DSIGMA)
    Lopperator = Lopperator.todense()
    LBCsigma = LBCsigma.todense()

    DX = DX.todense()
    for i in range(Ny):
        Lopperator[index[i,0],:]        =       -DX[index[i,0],:]
        Lopperator[index[i,-1],:]        =       -DX[index[i,-1],:]
    for i in range(Nx):
        Lopperator[index[0,i],:]        =     -  LBCsigma[ index[0,i]  , : ]
        Lopperator[index[-1,i],:] = 0
        Lopperator[index[-1,i],index[-1,i]] = 1
    f[index[-1,:]] = u0
    Lopperator = sp.csr_matrix(Lopperator)

    #Solve for u
    u = spl.spsolve(Lopperator,f)

    #Calculating d
    d = np.zeros(Nx)
    for i in range(Nx):
        d[i] = zeta[i]+h


    #Generating W
    W=DSIGMA.dot(u)[index[-1,:]]
    W = 1/d*W

    #Calculating deta and dphi
    on = np.ones(Nx)
    deta = -dZeta*np.dot(Dx,phi) + W *(on+(dZeta)**2)
    dphi = -g*zeta - (np.dot(Dx,phi)**2)/2 +  (W**2)/2  + (W**2 * (dZeta**2))/2

     #Calculating pressure and Force
    if CalcPressure:
        U = DX.dot(u)
        U = np.array(U).flatten("F")

        W2 = DSIGMADZ.dot(DSIGMA.dot(u) )

        U = U[index[:,-1]]
        W2 = W2[index[:,-1]]
        dens = .9982071
        #Initializing pressure calculations
        LegPol = DB.Poly1D(DB.JACOBI,(0.0,0.0))
        Z = np.zeros([Ny])
        WZ = np.zeros([Ny])
        p = np.zeros([Ny])
        d = h+zeta[-1]
        ZREAL = sigma*(h+zeta[-1])-h
        p = g*(zeta[-1]-ZREAL)
        p = p + 1/2*(U[-1]**2-U**2+W2[-1]**2-W2**2)
        for i in range(Ny):
            Z,WZ = LegPol.GaussLobattoQuadrature(Ny-1)

            #Transoforming for each iteration
            V = LegPol.GradVandermonde1D(Z[Ny-1-i:],i,0)
            V2 = LegPol.GradVandermonde1D(Z,i,0)
            VInv = np.linalg.inv(V)
            TRANS = np.dot(V2,VInv)
            W2Z = np.dot(TRANS,W2[Ny-1-i:])
            WLASTZ = np.dot(TRANS,WLAST[Ny-1-i:])
            WDT = (W2Z-WLASTZ)/tstep
            integral = 0
            for j in range(Ny):
                integral = integral + WZ[Ny-1-j]/2*WDT[Ny-1-j]

            #Scaled integral
            p[Ny-1-i] = p[Ny-1-i] + (zeta[-1]-ZREAL[Ny-1-i])/2*integral

        p = p*dens
        Force = 0
        #Scaled integral
        for j in range(Ny):
            Force = Force + p[j]*WZ[j]
        Force = Force*2/(d)

    else:
        W2 = np.zeros([Ny])
        p = np.zeros([Ny])
        Force = 0

    return  dphi,deta,W2,p,Force

```

G.2 Problem implementations

The standing wave

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 31 09:58:43 2013

@author: cbrams

Script intended to create a standing wave. Will also serve as
script for zero-movement wave by uncommenting the #Nu = Nu*0 after
initial condition calculation
"""

from __future__ import division
#import scipy.io as io
import numpy as np
import DABISpectral1D
import matplotlib.pyplot as plt
from dTimefinite import dTime_finite
import matplotlib.animation as animation
import scipy.sparse as sp
plt.close('all')

L = 2
Nx = 45
Ny = 20

polyLeg = DABISpectral1D.Poly1D(DABISpectral1D.JACOBI, (0.0,0.0))
#Generate x
x,xw = polyLeg.GaussLobattoQuadrature(Nx-1)
xreal = (x+1)/2*L


#Generate standing wave
a = np.array([0.8867*10**(-1),0.5243*10**(-2),0.4978*10**(-3),0.6542*10**(-4),\
0.1007*10**(-4),0.1653*10**(-5),0.2753*10**(-6),0.4522*10**(-7)])
Nu = 0;
for i in range(8):
    Nu = Nu + (i+1)**(1/4)*a[i]*np.cos((i+1)*np.pi*x)
Nu = Nu*1/2
```

```

#Nu = Nu*0



"""
CALCULATING DIFFERENTIAL OPERATORS
"""
L = 2.
cx = (L)/(2.)

h = 2
csigma = h/2
polyLeg = DABISpectral1D.Poly1D(DABISpectral1D.JACOBI, (0.0,0.0))
(y,w) = polyLeg.GaussLobattoQuadrature(Ny-1)
V = polyLeg.GradVandermonde1D(y,Ny-1,0,norm=True)


Vx =polyLeg.GradVandermonde1D(y,Ny-1,1,norm=True)

(x,wx) = polyLeg.GaussLobattoQuadrature(Nx-1)
xV = polyLeg.GradVandermonde1D(x,Nx-1,0,norm=True)
xVx =polyLeg.GradVandermonde1D(x,Nx-1,1,norm=True)

#Generating Ds
Dsigma = np.linalg.solve(V.T,Vx.T).T/csigma
Dx = np.linalg.solve(xV.T,xVx.T).T/cx
X , SIGMA = np.meshgrid((x+1)/2,(y+1)/2)

#Calculating kronecker Ds
DX = np.kron(Dx,np.identity(Ny))
DSIGMA = np.kron(np.identity(Nx),Dsigma)

DX = sp.csr_matrix(DX)
DSIGMA = sp.csr_matrix(DSIGMA)


#Prepare time-integrator
tmp = np.zeros(Nx*2)

nT = 20
t = np.linspace(0,1,nT)
tmp[:Nx] = np.zeros(Nu.shape)
tmp[Nx:] = Nu
```

```
t = 0
tEnd = 1.13409
tstep = 0.01
tN = int(tEnd/tstep)

#Initialize room for results
yt = np.zeros([tN+1,tmp.size])
p = np.zeros([tN+1,Ny])
ForceArr = np.zeros([tN+1])
yt[0,:] = tmp
t0 = 0
WLAST = np.zeros(Ny)
for t0 in range(tN+1):
    #Runge kutta where only the last step computes force and pressure
    set1,_,_,_ = dTime_finite(tmp,t,WLAST,tstep,False,Nx,Ny\
    ,Dx,Dsigma,DX,DSIGMA)
    sol1 = tmp + 1/4*tstep*set1
    set2,_,_,_ = dTime_finite(sol1,t,WLAST,tstep,False,Nx,Ny\
    ,Dx,Dsigma,DX,DSIGMA)
    sol2 = tmp + 1/3*tstep*set2
    set3,_,_,_ = dTime_finite(sol2,t,WLAST,tstep,False,Nx,Ny\
    ,Dx,Dsigma,DX,DSIGMA)
    sol3 = tmp + 1/2*tstep*set3
    set4,WNEW,press,Force = dTime_finite(sol3,t,WLAST,tstep\
    ,True,Nx,Ny,Dx,Dsigma,DX,DSIGMA)

    #Storing values
    sol4 = tmp + tstep*set4
    yt[t0,:] = sol4
    p[t0,:] = press
    ForceArr[t0] = Force
    WLAST = WNEW
    tmp = sol4
    t = t + tstep

#Create grid
(y,w) = polyLeg.GaussLobattoQuadrature(Ny-1)
Xs , SIGMA = np.meshgrid(x,(y+1)/2)

#Find phi1 and eta1
phi1 = yt[:,:Nx]
eta1 = yt[:,Nx:]
```

```python
#Plot eta (animation)
fig = plt.figure()
ax = fig.add_subplot(111)
plt.ylim(-np.max(np.abs(eta1)),np.max(np.abs(eta1)))
def animate(i):
    line.set_ydata(eta1[i,:])  # update the data
    return line,

##Init only required for blitting to give a clean slate.
def init():
    line.set_ydata(np.ma.array(x, mask=True))
    return line,

line, = ax.plot(x,eta1[0,:])

ani = animation.FuncAnimation(fig, animate, np.arange(1, tN+1), init_func=init,
#End animation

#Plot initial condition
plt.figure()
plt.plot(x,Nu)
plt.title("Initial condition")
plt.xlabel("x")
plt.ylabel("$\eta$")
TT = np.linspace(0+tstep,tEnd,tN)
plt.savefig("../../Latex/Billeder/WaveModel/TestStandingInitial.eps")


#Plot the period plot
plt.figure()
plt.plot(x,eta1[0,:],color="blue",label="Initial")
for i in range(1,tN-1,int(tN/20)):
    plt.plot(x,eta1[i,:],color="gray")

plt.plot(x,eta1[i,:],color="red",label="End")
plt.title("Period of the standing wave with 20 steps")
plt.xlabel("x")
plt.ylabel("$\eta$")
plt.legend()

plt.savefig("../../Latex/Billeder/WaveModel/TestStandingPeriod.eps")

#Plot the Force
plt.figure()
plt.plot(TT,ForceArr[1:])
plt.xlabel("t")
plt.ylabel("F")
plt.title("Force on the right boundary")
plt.savefig("../../Latex/Billeder/WaveModel/TestStandingForce.eps")


##Needed for export of PNGs in order to create .gif
#for i in range(1,tN):
#    plt.figure()
#    plt.plot(x,eta1[i,:])
#    plt.ylim(-0.01,0.01)
#    plt.savefig("fig%03d.png"%i)
plt.show()
```

The Gauss pulse

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 31 00:45:13 2013

@author: cbrams

Script for calculating uncertainty quantification in our wave problem
"""

from __future__ import division
import numpy as np
import DABISpectral1D
import DABISpectral1D as DB
import matplotlib.pyplot as plt
from dTimefinite import dTime_finite
import scipy.sparse as sp
plt.close('all')

L = 2
Nx = 70
Ny = 25

polyLeg = DABISpectral1D.Poly1D(DABISpectral1D.JACOBI, (0.0,0.0))
#Generate x
x,xw = polyLeg.GaussLobattoQuadrature(Nx-1)
```

```
xreal = (x+1)/2*L
Nu = np.exp(-((x-0.01)**2)/(2*(0.08)**2))/100
Nu[Nu<1e-06] = 0




"""
CALCULATING DIFFERENTIAL OPERATORS
"""
L = 2.
cx = (L)/(2.)

h = 2
csigma = h/2
polyLeg = DABISpectral1D.Poly1D(DABISpectral1D.JACOBI, (0.0,0.0))
(y,w) = polyLeg.GaussLobattoQuadrature(Ny-1)
V = polyLeg.GradVandermonde1D(y,Ny-1,0,norm=True)


Vx =polyLeg.GradVandermonde1D(y,Ny-1,1,norm=True)

(x,wx) = polyLeg.GaussLobattoQuadrature(Nx-1)
xV = polyLeg.GradVandermonde1D(x,Nx-1,0,norm=True)
xVx =polyLeg.GradVandermonde1D(x,Nx-1,1,norm=True)

#Generating Ds
Dsigma = np.linalg.solve(V.T,Vx.T).T/csigma
Dx = np.linalg.solve(xV.T,xVx.T).T/cx
X , SIGMA = np.meshgrid((x+1)/2,(y+1)/2)

#Calculating kronecker Ds
DX = np.kron(Dx,np.identity(Ny))
DSIGMA = np.kron(np.identity(Nx),Dsigma)

DX = sp.csr_matrix(DX)
DSIGMA = sp.csr_matrix(DSIGMA)

"""
Start Uncertainty Quantification
"""
#Initiate limits for stochastic variable
JPol = DB.Poly1D(DB.JACOBI,(0.0,0.0))
start = 0.9
```

```python
end = 1.1

#Number of iterations
nUQ = 10

#Generate grid and weights
R,Rw = JPol.GaussLobattoQuadrature(nUQ-1)
Rw = Rw/2
nT = 20

#Setting tEnd
tEnd = 2.5
tstep = 0.01

#Making room for all results
tN = int(tEnd/tstep)
ytTOTAL = np.zeros([tN+1,Nx*2,nUQ])
ForceArrTOTAL = np.zeros([tN+1,nUQ])
Weights = np.tile(Rw,[tN+1,1])
for iUQ in range(nUQ):
    #Prepare time-integrator
    tmp = np.zeros(Nx*2)

    #Modify initial condition
    t = np.linspace(0,1,nT)
    tmp[:Nx] = np.zeros(Nu.shape)
    tmp[Nx:] = Nu*(R[iUQ]+1)/2*(end-start)+start
    t = 0


    #Prepare for time integration
    yt = np.zeros([tN+1,tmp.size])
    p = np.zeros([tN+1,Ny])
    ForceArr = np.zeros([tN+1])
    yt[0,:] = tmp
    t0 = 0
    WLAST = np.zeros(Ny)

    for t0 in range(tN+1):
        set1,_,_,_ = dTime_finite(tmp,t,WLAST,tstep,False,Nx\
        ,Ny,Dx,Dsigma,DX,DSIGMA)

        sol1 = tmp + 1/4*tstep*set1
        set2,_,_,_ = dTime_finite(sol1,t,WLAST,tstep,False,Nx\
        ,Ny,Dx,Dsigma,DX,DSIGMA)

        sol2 = tmp + 1/3*tstep*set2
        set3,_,_,_ = dTime_finite(sol2,t,WLAST,tstep,False,Nx,\
        Ny,Dx,Dsigma,DX,DSIGMA)

        sol3 = tmp + 1/2*tstep*set3
        set4,WNEW,press,Force = dTime_finite(sol3,t,WLAST,tstep\
        ,True,Nx,Ny,Dx,Dsigma,DX,DSIGMA)

        sol4 = tmp + tstep*set4

        #Save data
        yt[t0,:] = sol4
        p[t0,:] = press
        ForceArr[t0] = Force
        WLAST = WNEW
        tmp = sol4
        t = t + tstep

    #Save data for UQ iteration
    ytTOTAL[:,:,iUQ] = yt
    ForceArrTOTAL[:,iUQ] = ForceArr

#Calculate mean
ForceMean = np.sum(Weights*ForceArrTOTAL,axis=1)

ForceMeanTiled = np.tile(ForceMean,[nUQ,1]).T

#Calculate variance
ForceVar = np.sum(((ForceArrTOTAL-ForceMeanTiled))**2*Weights,axis=1)

ForceStd = np.sqrt(ForceVar)

#Save data
np.savez("ForceU10Nx70",Weights = Weights,ForceMean = ForceMean,ForceMeanTiled = 


#Visualize
plt.figure()
TT = np.linspace(0+tstep,tEnd,tN)
plt.fill_between(TT,ForceMean[1:]-ForceStd[1:],ForceMean[1:]+ForceStd[1:],color="
plt.plot(TT,ForceMean[1:],'r')
plt.plot(TT,ForceMean[1:]+2*ForceStd[1:],'b')
plt.plot(TT,ForceMean[1:]-2*ForceStd[1:],'g')
plt.xlabel("t")
plt.ylabel("F")
plt.savefig("../../Latex/Billeder/WaveModel/GaussianUQU10.eps")

plt.show()
```

Bibliography

[Big12] Daniele Bigoni. Implementation of spectral methods in 1 dimension in python., 2012.

[EK06] Allan Peter Engsig-Karup. Unstructured nodal dg-fem solution of high-order boussinesq-type equations, 2006.

[EK11a] Allan P. Engsig-Karup. Slides for 02689 – polynomial methods. 2011.

[EK11b] Allan Peter Engsig-Karup. Advanced numerical methods for differential equations, assignment 3, Autumn 2011.

[Hun07] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun 2007.

[JOP$^+$] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[Kop09] David A. Kopriva. *Implementing Spectral Methods for Partial Differential Equations.* Springer, 2009.

[UG82] C. T. Shin U. Ghia, K. N. Ghia. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of Computational Physics*, 1982.

[WZ10] G.Xi W. Zhang, C.H. Zhang. An explicit chebyshev pseudospectral multigrid method for incompressible navier-stokes equations. *Computers & Fluids 39*, 2010.

[Xiu10] Dongbin Xiu. *Numerical Methods for Stochastic Computations – A Spectral Method Approach.* Princeton University Press, 41 William Street, Princeton, New Jersey, USA, 2010.

[YA96] M. Glozman Y. Agnon. Periodic solutions for a complex hamiltonian system: New standing water-waves. *Wave Motion*, 1996.