Isogeometric Analysis

Higher-Order Differential Equations

Mathias Farvolden
Tjomsland

Master of Science in Physics and Mathematics

Submission date:	June 2013
Supervisor:	Trond Kvamsdal, MATH
Co-supervisor:	Kjetil Andre Johannessen, IMF

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

This thesis introduces Isogeometric Analysis as a potentional bridge between the Finite Element Analysis (FEA) and Computer Aided Design (CAD) communities. An introduction to B-splines and B-spline-based Isogeometric Analysis is given. Then an implemented Isogeometric Analysis solver is used to solve both the Poisson problem and the higher order Biharmonic equation on the unit square. The significant convergence results for Isogeometric Analysis is verified in both cases. Lastly, the highly non-linear, and stiff, Cahn-Hilliard equation is studied. The implemented solver is used to show good results, particularily in the Dirichlet boundary condition case. This demonstrates a major advantage of Isogeometric Analysis over traditional Finite Element Method, in that higher order partial differential equations can be solved without any major workarounds or adjustments to the solver by increasing the continuity of the basis.

Sammendrag

Denne masteroppgaven introduserer Isogeometrisk Analyse som en potensiell bro mellom Finite Element Analysis (FEA) og Computer Aided Design (CAD) - fagfeltene. En introduksjon til B-splines og Isogeometrisk Analyse basert på B-splines blir gitt. En implementasjon av en Isogeometrisk Analyse løser blir brukt for å løse både Poisson problemet og den høyere ordens Biharmoniske ligningen på enhetsfirkanten. De lovende teoretiske konvergensresultatene bekreftes i begge tilfeller. Til slutt studeres den høyst ikkelinære og stive Cahn-Hilliard ligningen. Den implementerte løseren gir gode, kvalitativt verifiserte resultater, spesielt med bruk av Dirichlet grensebetingelser. Dette demonstrerer en stor fordel som Isogeometrisk Analyse har over tradisjonell Elementmetode. Høyere ordens partielle differentialligninger kan løses uten store omformuleringer eller modifiseringer til metoden, fordi man kan øke kontinuiteten til basisen.

Preface and Acknowledgements

This project was completed as the concluding thesis of the Master of Science program called Applied Physics and Mathematics, with a specialization in Industrial Mathematics at the Norwegian University of Science and Technology (NTNU). The thesis was completed during the spring of 2013.
When the possibility to work with such a new and up-and-coming field as Isogeometric Analysis came by, the chance was instantly grasped. Isogeometric Analysis seems to be shifting the analysis community towards greater connection with CAD geometrically, and shows great results. The text assumes some knowledge of Finite Element Analysis, but assumes no prior knowledge of Splines and Isogeometric Analysis.
The project has been very rewarding, and greatly challenging. Especially, since the field is relatively new, not a lot of textbooks or instructional texts have been written yet. However, the works of Thomas J. R. Hughes and Yuri Bazilevs have been a great source of knowledge and inspiration. Furthermore, I want to direct a huge thank you to my great project supervisors, Trond Kvamsdal and associate supervisor Kjetil A. Johannessen, whom both have been great contributors and facilitators of my learning and my thesis' completion. A special thank you to Kjetil A. Johannessen for his many brilliant ideas and his availability as issues were worked on underways. I also want to direct a big thank you to Arne Morten Kvarving, who stepped in and contributed with his great knowledge of time-integration and stiff systems towards the end of the project.
Mathias F. Tjomsland
16 June 2012
Trondheim

Contents

1 Introduction

For a long time, **finite element analysis** (FEA) has been the analysis tool of choice for many engineers [24]. The **finite element method** (FEM), based on FEA, has generally proved very applicable, providing engineers with a much needed analysis tool to model many differential equations on nearly arbitrary domains. The applications of FEA are seemingly endless, playing a vital role in, amongst others; structural engineering [40], aeronautical engineering [21], biomechanical engineering [36], and the automotive industries [45]. Simulation of car crashes through the use of FEA, is saving the automotive industry large sums otherwise used on physical mock-ups for crash testing. Building a bridge, the engineers will need to know how much load it can withstand, and at what frequencies it will resonate to identify any weak spots. Dentists use FEA to simulate stress distributions in implants. Areonautical engineers use FEA to optimize their wing design on new aircraft.

All of these applications require some sort of model or computerized geometrical representation of the object on which analysis is to be performed. The vast amount of design work is now done on a computer, using **computer aided design** technology (CAD). 3D models of bridges, buildings, aircraft wings, and cars are created by the design teams in CAD, before they are handed over to engineers for analysis. However common this practice is, there has so far been a void between the FEA and CAD industries. The two industries have grown into independent fields, specifically, without an agreed-upon mutual standard; CAD and FEA use totally different geometrical representations. This has been a grave shortcoming of the industry as it is today, as nearly 80% of analysis time [25] actually is spent by engineers on creating an analysis-suitable geometric representation and meshes of the CAD model. There are in most cases no fully automatic computerized way to do this transition from CAD to FEA geometries.

Furthermore, the analysis-suitable geometric representation the engineers create is only an approximation to the CAD model. This implies that if the FEA geometrc representation needs to be refined, changed, or improved; the engineers have to consult the CAD model again. The fact that the FEA geometries only are approximations is known to have a significant effect on some highly geometry-dependend situasions [13, preface], such as:

- Shell buckling analysis,
- Boundary layer phenomena in aerodynamics

- Sliding contact bweteen bodies

In 2003, after a conversation about these shortcomings of the way the industry was built up, and the lack of exactness in FEA geometries, Thomas J. R. Hughes envisioned a solution that would bring the CAD and FEA industries much closer together [13, ch.1]. The idea was to attemt to unite FEA and CAD around one common, exact model. Now, relatively speaking, CAD is a larger industry than FEA, and large sums of investments have gone into developing the standards of todays CAD technology. With this in mind, and knowing that for all intents and purposes the CAD models were already exact, Hughes found that the most rational way to implement his vision would be to suggest a new form of analysis to the engineering community. This new form of analysis should utilize the existing CAD geometrical representation standards in a way that would remove the need of generating a seperate approximate FEA geometry, and should also if possible be based on familiar analysis tools such as FEA to withold some familiarity for engineers . Hence, Hughes proposed **Isogeometric Analysis** [25], [4], [3], [12], [11], [53], [20].

Isogeometric Analysis aims to use the exact CAD representation for analysis. Hughes found that the most widely used industry standard CAD geometric representation was that of splines (NURBS based on B-splines) [13, ch. 1], [39],[17] [10]. These spline represtentations are constituted by a number of basis functions - each with a small span - very similar to the basis funtions or shape functions used in FEA. Hughes showed that these spline basis functions could replace the classical polynomial basis functions of FEA in the familiar Galerkin method. Using these spline basis functions would allow the analysis to work potentially directly on CAD models, meaning the geometric representation could be exact. Furthermore, the geometric representation would be exact at all levels of spline-mesh refinement, as the refinement of a spline based geometry does not change the geometry - just the resolution of computation [25].

This thesis aims to investigate B-spline theory and demonstrate their use as basis functions in the Galerkin method. This will be done by creating an Isogeometric Analysis solver implementation in MATLAB, that uses spline basis functions. Specifically, one very noteworthy advantage of a spline basis is that one can create a C^n-continuous basis. Whereas traditional FEA operates with a C^0 continuity across elements, Isogeometric analysis effortlessly allows one to choose higher orders of continuity. This is especially applicable to higher order partial differential equations - equations that demand such a higher-continuous basis. The implemented solver is thus tested and verified for both the Poisson equation and the higher order Biharmonic equation. The strong error results found by Hughes is verified in each case - revealing similar, if not surpassing results to that of the FEM[13]. Finally the solver is used to study the highly non-linear Cahn-Hilliard equation - a very stiff and highly-nonlinear fourth order partial differential equation [22]. This equation is not solvable by traditional FEM without performing complex workarounds. Therefore, this thesis also aims to demonstrate the power of Isogeometric Analysis as a unified system to solve both lower and higher order partial differential equations.

2 Spline Theory

2.1 Splines

The industry standard in CAD geometry representation is based on **splines** [13]. Splines are also widely used in computer graphics and general 3D-design software to represent geometries. They are smooth functions that are piecewise defined. Each piecewise part may be quite simple, but the power of spline representation is that they can be used to model arbitrary functions and geometries. Also, they are defined in a way that allows sufficient smoothness in the intersection between the piecewise parts, which shall be shown as importantto the topic of this thesis. The most commonly used spline representations in CAD are **NURBS** (Non-Uniform Rational B-Splines). NURBS are projetive transformations of B-splines from $\mathbb{R}^{d+1}$ into $\mathbb{R}^d$, and the main advantage of NURBS over B-splines is that they can represent all conic sections exactly. NURBS are based on weighted B-splines, and so electing a basis of B-splines for this thesis is a good starting point in Isogeometric Analysis. Hence, an introduction to B-splines is in place. All figures are generated from Matlab implementations of the spline theory.

2.2 B-Splines

B-splines [43] are the type of splines used in this Isogeometric Analysis implementation. The B-spline functions are composed of several basis functions defined in a parameter space. Let Ξ which is called the **knot vector** be a non-decreasing set of numbers that will serve as coordinates in the B-spline parameter space. Also, let p be the chosen polynomial order. Thus, we can define a regular knot vector as $\Xi = \{\xi_1, \xi_2, \ldots, \xi_{n+p+1}\}$, where $\xi_i \in \mathbb{R}$ is the i^{th} **knot**, and $\xi_i \leq \xi_{i+1}$. Defined in this way, n turns out to be the number of basis functions we will need to compute the B-spline curve over this knot vector.

A knot vector is called **uniform** if each of the knots are equally spaced, and **non-uniform** if they are not. Also, a knot vector is called **open** if the first and last value of ξ is repeated $p + 1$ times. Open and non-uniform knot vectors are standard in CAD [13, p. 21], and hence, will be supported and assumed throughout this thesis's implementation.

The B-splines are composed of several basis functions that are defined over $p+1$ spans in the knot vector. These basis functions $N_{i,p}(\xi)$ are defined in a recursive manner[34], starting with piecewise constants, for $p = 0$:

$$N_{i,0}(\xi) = \begin{cases} 1 & \text{if} \quad \xi_i \leq \xi < \xi_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

When $p = 1, 2, 3, \ldots$, the basis functions are defined recursively by the **Cox-de Boor recursion formula** ([13, chapter 2.1.2] [5]):

$$N_{i,p}(\xi) = \frac{\xi - \xi_i}{\xi_{i+p} - \xi_i} N_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{\xi_{i+p+1} - \xi_{i+1}} N_{i+1,p-1}(\xi) \tag{2}$$

The graphs of the first few basis functions on a uniform knot vector $\Xi = \{0, 1, 2, 3, 4, 5, \ldots\}$, when the polynomial degree $p = 1, 2, 3$ are shown in figure 1, 2, 3 respectively.

For the purpose of this thesis, it will be worthwhile to note that the basis

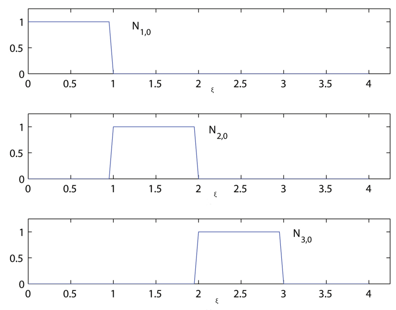


Figure 1: First three basis functions, $p = 0$ on knot vector $\Xi = \{0, 1, 2, 3, 4, 5, \ldots\}$

functions in figures 1, 2 and 3 are on a non-open knot vector. On an open knot vector, things look a little different at the end-points, because of the $p + 1$ identical knots. This is because of an important property of B-splines [34, Theorem 1.4], which states: If a knot ξ_i occurs m times in the knot vector, then the spline function will have continuous derivatives up to order $p - m$ at the ith knot. This means that by repeating knots, one can control the behaviour of the spline function. If a knot is repeated p times, the spline function will have C^0-continuity at that knot value. Furthermore, the spline function will, when it has C^0-continuity, be interpolatory. This means, using open knot vectors, the spline function will always be interpolatory at the endpoints, but not neccecarily at the internal knot values unless repeated knots are present, or $p = 1$. This property is visible in the arbitrary spline function interpolating the circled points in the top half of figure. 4; the endpoints are interpolated, but not the internal points. The

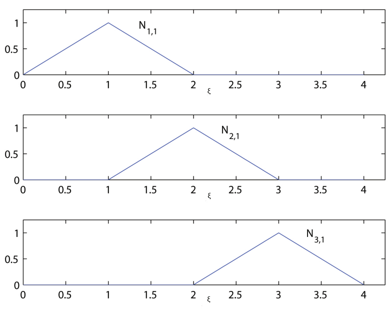


Figure 2: First three basis functions, $p = 1$ on knot vector $\Xi = \{0, 1, 2, 3, 4, 5, \ldots\}$

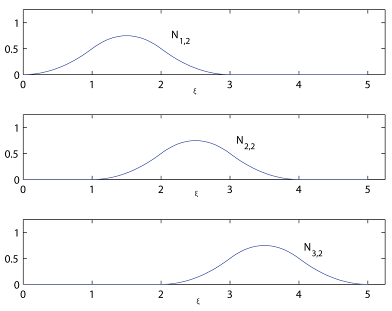


Figure 3: First three basis functions, $p = 2$ on knot vector $\Xi = \{0, 1, 2, 3, 4, 5, \ldots\}$

effect of adding a repeated knot is shown in the bottom half of figure. 4. Here $p = 2$ and $\Xi = \{0, 0, 0, 1, 2, 3, 4, 5, 5, 5\}$. The graph on the bottom has a double knot at $\xi = 2$, giving $\Xi = \{0, 0, 0, 1, 2, 2, 3, 4, 5, 5, 5\}$ making the spline function interpolatory in that point.

With these properties in place, a full set of basis functions for $p = 2$ on a non-uniform, open knot vector $\Xi = \{0, 0, 0, 1, 2, 3, 3, 4, 5, 5, 5\}$, with a double knot at $\xi = 3$ is shown in figure 5. One more important feature of the B-spline basis functions that now can be seen is that the sum of the nonzero basis functions in

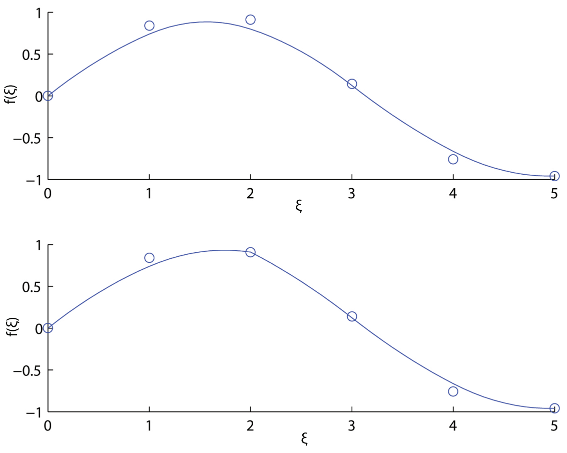


Figure 4: Spline function modelling $sin(\xi)$ based on the circled points, $p = 2$ and $\Xi = \{0,0,0,1,2,3,4,5,5,5\}$, revealing the interpolatory nature of splines at the endpoints (top), and the effect of inserting a double knot at 2, giving $\Xi = \{0,0,0,1,2,2,3,4,5,5,5\}$ (bottom).

a arbitrary point ξ is unity:

$$\sum_{i=1}^{n} = N_{i,p}(\xi) = 1 \tag{3}$$

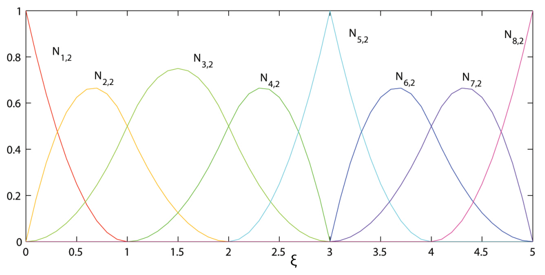


Figure 5: A complete set of basis functions for $p = 2$ and the knot vector $\Xi = \{0,0,0,1,2,3,3,4,5,5,5\}$

The last concept needed before we can formally define the spline curve, is that of **control points**. The control points are the real number coefficients of the basis functions at each knot value ξ. The control points will be stored as points in a vector called **B**. B-spline curves are made by taking linear combinations of the basis functions, and hence we can finally define a **B-spline curve**, $C(\xi)$:

$$C(\xi) = \sum_{i=1}^{n} N_{i,p}(\xi)B_i, \tag{4}$$

where $N_{i,p}, i = 1, 2, \ldots, n$ are the basis functions as defined in equation (2) on a given knot vector, and B_i are the corresponding control points.

2.3 Variation Diminishing Spline Approximation

The B-spline curves can be applied in many ways and they are a powerful tool in representing curves, surfaces and solids - in general; most geometric form or shape that would be wanted as a domain for the Isogeometric Analysis of this thesis. Firstly, splines can be used to approximate a function f defined on an interval $[a, b]$. A simple but very useful method is called the **Variation Diminishing Spline Approximation** [34, p.110]. The variation diminishing approximation V of a function $f(x)$ on $[a, b]$ is defined by;

$$(Vf)(x) = \sum_{i=1}^{n} N_{i,p}(x)f(\xi_i^*), \tag{5}$$

where $\xi_i^* = (\xi_{i+1} + \ldots + \xi_{i+p})/p$ are the averages of each knot interval, and the knot vector spans $[a, b]$ with a and b being the endpoints of the knot vector. Figure 6 shows the variation diminishing approximation to $sin(x), x \in [0, 5]$ using $p = 1$ and $p = 2$ respectively in the upper and middle plot, whereas the bottom plot uses $p = 2$ and a knot vector with halved knot spans, showing significantly improved results.

2.4 Parametric Spline Curves

By letting the control points be points in $\mathbb{R}^d$,instead of real values as before, we can define **parametric spline curves**:

$$g(u) = \sum_{i=1}^{n} N_{i,p}(x)\mathbf{B}(u), \tag{6}$$

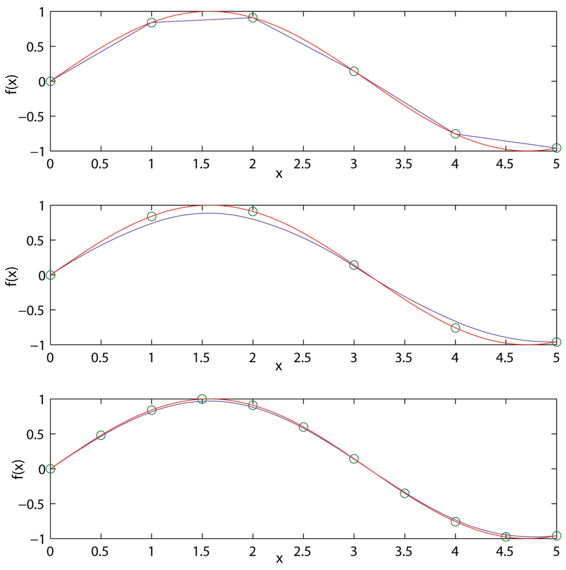


Figure 6: Variation diminishing spline approximation of $sin(x)$, $p = 1, 2$, on knot vector $\Xi = \{0, 0, 0, 1, 2, 3, 4, 5, 5, 5\}$,(top, middle respectively) and $p = 2, \Xi = \{0, 0, 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5, 5\}$ (bottom)

where, in $\mathbb{R}^d$, each entry in $\mathbf{B}_i$ will be a d-component coordinate in $\mathbb{R}^d$. In $\mathbb{R}^3$, if we define each control point $B_i = (x_i, y_i, x_i)$, the parametric spline curve will turn out to be:

$$g(u) = (\sum_{i=1}^{n} N_{i,p}(u)x_i, \sum_{i=1}^{n} N_{i,p}(u)y_i, \sum_{i=1}^{n} N_{i,p}(u)z_i) \tag{7}$$

An example of a parametric spline curve using the variational diminishing approach in $\mathbb{R}^3$, where $x_i = sin(u), y_i = cos(u)$ and $z_i = u$ is shown in figure 7

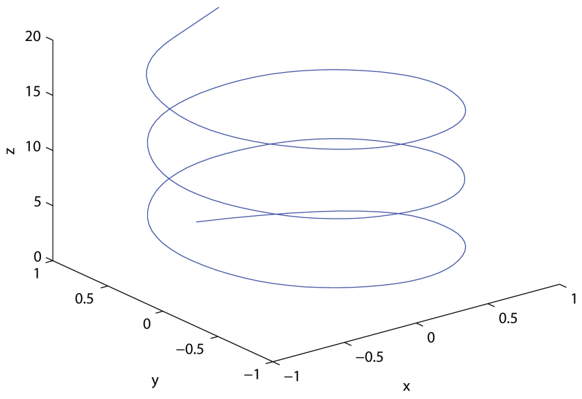


Figure 7: Parametric spline curve, variational diminishing approach, $p = 2$, $\Xi = \{0,0,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,19,19\}$, $x_i = sin(u), y_i = cos(u)$ and $z_i = u$

2.5 Tensor Product Spline Surfaces

The idea of parametric curves can be generalized to **parametric spline surfaces**, or tensor product spline surfaces over a rectangular domain. The idea is to use two, linearly independent, sets of spline basis functions - each representing one parametric dimension, and letting them span the whole rectangular parametric domain [34, Definition 7.1]. This is very similar to the idea that will be used to develop the basis functions used in this thesis shortly. The two sets of spline basis functions will be denoted $N_{i,p_1}(\xi)$ and $N_{j,p_2}(\eta)$, and will each be defined just as in equation (2). The control points will now be a matrix $\mathbf{B}$ holding one coordinate point (x, y) for every index (i, j) A surface can then be represented as

$$f(\xi, \eta) = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} N_{i,p}(\xi) N_{j,p}(\eta) \mathbf{B}_{i,j} \tag{8}$$

If we allow the two parametric directions to be the regular coordinates, x and y, we can develop a variational diminishing spline approximation to a function defined on a rectangular domain [34, chapter 7.2.1]:

$$(Vf)(x, y) = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} N_{i,p_1}(x) N_{j,p_2}(y) f(\xi_i^*, \eta_j^*) \tag{9}$$

where

$$\xi_i^* = (\xi_{i+1} + \ldots + \xi_{i+p_1})/p_1 \tag{10}$$

$$\eta_j^* = (\eta_{j+1} + \ldots + \eta_{j+p_2})/p_2 \tag{11}$$

as before. The variational diminishing spline approximation to the function

$$f(x, y) = g(x)g(y), \tag{12}$$

where

$$g(x) = \begin{cases} 1, & 0 \leq x \leq 0.5 \\ e^{-10(x-0.5)}, & 0.5 < x \leq 1 \end{cases} \tag{13}$$

on the unit square, is shown in figure 8. Here, a double knot is used at $x, y = 0.5$, to preserve the fact that the function has discontinuities in its partial derivatives across the lines $x = 0.5$ and $y = 0.5$. The results show a very good approximation.

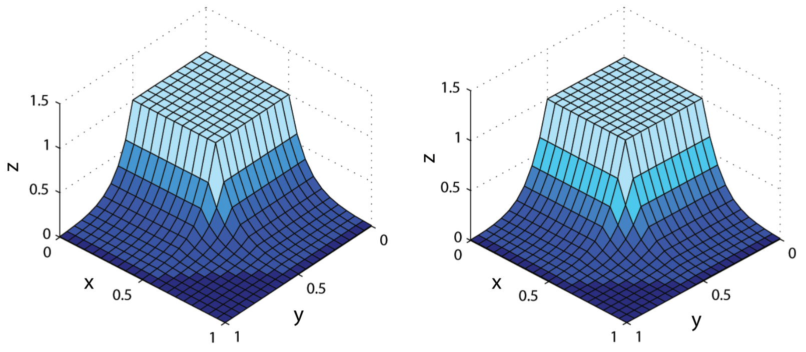


Figure 8: variational diminishing spline approximation to the equation (13). Exact solution (left), and spline approximation (right),both knot vectors are equal to $\{0, 0, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1, 1\}$

2.6 Matrix Representation of B-splines

B-splines are defined as a recurrence relation; the Cox-de Boor algorithm as stated earlier. Recurrence relations can get a little difficult to work with, even if the relation is simple. However, it turns out that in the case of B-splines, this recurrence relation can be represented as products of simple matrices, as defined in Theorem 2.13 in [34]:

Let $t = (tj)_{j=1}^{n+p+1}$ be a knot vector of B-splines with degree p, and let μ be an integer that satisfies $t_\mu < t_{\mu+1}$, and let $p+1 \leq \mu \leq n$. Then, for each positive integer k with $k \leq p$ we can define the matrix $\mathbf{R}_k^\mu(x) = \mathbf{R}_k(x)$ as

$$\mathbf{R}_k(x) = \begin{pmatrix} \frac{t_{\mu+1}-x}{t_{\mu+1}-t_{\mu+1-k}} & \frac{x-t_{\mu+1-k}}{t_{\mu+1}-t_{\mu+1-k}} & 0 & \cdots & 0 \\ 0 & \frac{t_{\mu+2}-x}{t_{\mu+2}-t_{\mu+2-k}} & \frac{x-t_{\mu+2-k}}{t_{\mu+2}-t_{\mu+2-k}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{t_{\mu+k}-x}{t_{\mu+k}-t_\mu} & \frac{x-t_\mu}{t_{\mu+k}-t_\mu} \end{pmatrix} \tag{14}$$

Then, with this definition of $\mathbf{R}_k(x)$, for any x in the interval $[t_\mu, t_{\mu+1})$ the values for the $p+1$ B-splines of degree p that are nonzero in the interval can be written as

$$\mathbf{N}_p^T = (N_{\mu-p,p} N_{\mu-p+1,p \cdots N_{\mu,p}} = \mathbf{R}_1(x)\mathbf{R}_2(x)\cdots\mathbf{R}_p(x). \tag{15}$$

Thus, the spline curve's value in a point x can be given as

$$f(x) = \mathbf{R}_1(x)\mathbf{R}_2(x)\cdots\mathbf{R}_p(x)\tilde{\mathbf{b}} \tag{16}$$

where $\tilde{\mathbf{b}} = (b_{\mu-p}, b_{\mu-p+1}, \cdots, b_\mu)^T$ are the control points.
This matrix representation allows for further analysis of the derivatives in the next section, and give birth to the following algorithm [34, 2.21] for finding the values of each nonzero B-spline at a point x.

```
Data: Let $p$ be the polynomial degree, the knots
      $t_{\mu-p+1} \leq t_\mu < t_{\mu+1} \leq t_{\mu+p}$, and a number $x$ in $[t_\mu, t_{\mu+1})$
Result: The vector $\mathbf{N}_p$ will contain the values of the $p+1$ nonzero splines
        at $x$
Set $\mathbf{N}_0 = 1$ for $k = 1, ..., p$ do
    $\mathbf{N}_k^T(x) = \mathbf{N}_{k-1}^T(x)\mathbf{R}_k(x)$
end
```

Algorithm 1: evaluating nonzero B-splines at point x

This algorithm, however, is slightly streamlined (as in [34, Algorithm 2.22]) for the use in this thesis. The $\mathbf{R}_k(x)$ matrices are not formed explicitly, and the lower order spline values $\mathbf{N}_{p-1}, \mathbf{N}_{p-2}, ...$ are stored. These lower order spline values are needed for calculating spline derivatives, as we shall see in the next section.

2.7 Derivatives of B-splines

When building the numerical system to be solved, we will need the derivatives of the spline basis functions. Two important properties are used to obtain the expression for the derivatives from equation (15). Firstly, the $\mathbf{R}_k(x)$ matrices have the property [34, lemma 3.14] that when $2 \leq p$, and $x, z \in \mathbb{R}$,

$$\mathbf{R}_{p-1}(z)\mathbf{R}_p(x) = \mathbf{R}_{p-1}(x)\mathbf{R}_p(z). \tag{17}$$

Secondly, for the derivatives of two matrices $\mathbf{A}$ and $\mathbf{B}$, whose entries are functions of x, and whose dimensions are such that the matrix product $\mathbf{AB}$ makes sense, the following holds[34, Lemma 3.13]

$$D(\mathbf{AB}) = (D\mathbf{A})\mathbf{B} + \mathbf{A}(D\mathbf{B}), \tag{18}$$

Where D represent differentiating each entry with respect to x. Applying equation (18) to equation (15) we obtain

$$D\mathbf{N}_p(x) = \sum_{k=1}^{p} \mathbf{R}_1(x)\cdots\mathbf{R}_{k-1}(x)D\mathbf{R}_k(x)\mathbf{R}_{k+1}(x)\cdots\mathbf{R}_p(x), \tag{19}$$

where $D\mathbf{R}_k(x)$ represents the differentiation of each term with respect to x in $\mathbf{R}_k(x)$. Now, using equation (17), we can change the expression so that the differentiating operator D is moved from $\mathbf{R}_k(x)$ to $\mathbf{R}_p(x)$. This gives the derivatives as:

$$D\mathbf{N}_p(x) = p\mathbf{R}_1(x)\cdots\mathbf{R}_{p-1}(x)D\mathbf{R}_p(x) = p\mathbf{N}_{p-1}(x)D\mathbf{R}_p. \tag{20}$$

As mentioned in the discussion of the algorithm for computing values of nonzero B-splines, the lower order splines $\mathbf{N}_{p-1}$ are already being found when calculating $\mathbf{N}_p$ and thus only need to be stored to easily be able to also compute the derivative values with very little added computational cost.

To find second derivatives, we differentiate equation (20). $D(D\mathbf{R}_p) = 0$, and thus get

$$D^2\mathbf{N}_p(x)^T = pD\mathbf{N}_{p-1}(x)^T D\mathbf{R}_p. \tag{21}$$

Then, using equation (20), we get

$$D^2\mathbf{N}_p(x)^T = p(p-1)\mathbf{N}_{p-2}(x)^T D\mathbf{R}_{p-1}D\mathbf{R}_p. \tag{22}$$

The first and second derivatives willl be used in this thesis, however, it is noted that for the rth derivative, we find [34, Theorem 3.15]

$$D^r\mathbf{N}_p(x)^T = \frac{p!}{(p-r)!}\mathbf{N}_{p-r}(x)^T D\mathbf{R}_{p-r+1}\cdots D\mathbf{R}_p. \tag{23}$$

Finally, we observe that the derivative of the jth B-spline of degree p on knot vector t can be formulated as a recurrence relation [34, Theorem 3.16];

$$D\mathbf{N}_{j,p}(x) = p(\frac{\mathbf{N}_{j,p-1}(x)}{t_{j+p} - t_j} - \frac{\mathbf{N}_{j+1,p-1}(x)}{t_{j+p+1} - ti_(j+1)}), \tag{24}$$

2.8 Knot Insertion and Order Elevation

In regular FEA there are two main types of refinements used to enrich the basis [54]. h-refinement refers to decreasing the element size, or in other words increasing the number of elements to achieve higher resolution for computations in the basis. p-refinement involves increasing the polynomial degree of the basis functions defined on the elements. In this section, the refinement possibilities available in Isogeometric Analysis will be presented. The main difference from FEA is that the geometry remains constant under each type of refinement, and that whereas the FEM has C^0 continuity across elements at all times, this continuity is much more controllable in Isogeometric Analysis.

2.8.1 Knot Insertion

The Isogeometric refinement called knot insertion [13, 2.1.4.1] is in some ways similar to the h-refinement in FEA. However, in Isogeometric Analysis it does not alter the geometry. A knot insertion is performed when a new (or existing) knot value is added to the knot vector. However, as one adds a new knot to the knot vector, one has to update the control points. This can be accomplished rather simply in the following manner: If one has two knot vectors $\hat{\mathbf{t}} = (\hat{t}_j)_{j=1}^{n+p+1}$ and $\mathbf{t} = (t_i)_{i=1}^{m+p+1}$ that have common knots at the end and $\hat{\mathbf{t}} \subseteq \mathbf{t}$. Suppose for a fixed integer i, $1 \leq i \leq m$ that the integer μ is such that $\hat{t}_\mu \leq t_i < \hat{t}_{\mu+1}$. Then we have, from theorem 4.6 in [34, p.87],

$$\alpha_p(i)^T = (\alpha_{\mu-p,p}(i), ..., \alpha_{\mu,p}(i) = \{ \begin{matrix} 1 & \text{if } p = 0 \\ \mathbf{R}_1(t_i+1) \cdots \mathbf{R}_p(t_{i+p}) & \text{if } p > 0, \end{matrix} \tag{25}$$

where the $\mathbf{R}_{k,\hat{t}}$ are the B-spline matrices defined in section 2.6. Now, if

$$f = \sum_j c_j N_{j,p,\hat{t}} \tag{26}$$

is a spline on $\hat{\mathbf{t}}$, with spline coefficients $\mathbf{c}$, the corresponding B-spline coefficients $\mathbf{b}$ on $\mathbf{t}$ is given by

$$b_i = \sum_{j=u-p}^{\mu} \alpha_{j,p}(i)c_j = \mathbf{R}_1(t_{i+1}) \cdots \mathbf{R}_p(t_{i+p})\mathbf{c}_p, \tag{27}$$

where $\mathbf{c}_p = (c_{\mu-p}, ..., c_\mu)^T$. Then f can be written in terms of the new knot vector t and coefficients $\mathbf{b}$,

$$f = \sum_j b_j N_{j,p,t}. \tag{28}$$

If the inserted knot is not already present in the knot, the continuity of C^{p-1} is maintained everywhere. If the inserted knot is present in the original knot vector, the multiplicity of that knot increases by one, and the continuity is reduced by one at that point. This has no analouge in FEA. In general knot insertion is much more flexible than h-refinement, as one can control the continuity as well.

2.8.2 Order Elevation

Order elevation [13, 2.1.4.2] is related to the p-refinement in FEA. During order elevation, one increases the polynomial degree of the basis functions. Again, there is a bit more flexibility in the Isogeometric case. Say that one order elevates from p to q. If it is neccecary to preserve exatly the C^{p-m-1} continuity, where m is the knot multiplicity, that existed before the order elevation, one has to increase the multiplicity of each knot by $q-p$. However, if the multiplicity is left unchanged, the order-elevated basis will have C^{q-m-1} continuity. The main difference between order elevation and regular p-refinement is that p-refinement always has C^0 continuity in its basis.

2.8.3 K-refinement

A combination of knot insertion and order elevation will often be preferable, similar to combinations of h and p refinement in FEA. It turns out that knot insertion and order elevation do not commute. If, for example, a knot insertion is performed before an order elevation from p to q, one will obtain a basis with $p-m-1$ continuous derivatives. However, if one order elevates from p to q first, then inserts knots, one will end up with a basis with $q-m-1$ continous derivatives. It turns out [13, 2.1.4.3] that the second option also creates much fewer basis functions than the first. This refinement scheme was coined k-refinement by Cottrel, Hughes and Bazilevs [13]. There is no equivelant to this refinement in FEA. The fact that both knot insertion and order elevation are more flexible than their FEA relatives, and the fact that different sequenses of refinement give different results opens up to a wide variety of possible refinement schemes in Isogeometric Analysis.

3 Isogeometric Analysis Implementation

3.1 A Model Problem

To show the workings of Isogeometric Analysis, we will in this section perform the analysis on a simple, yet powerful model problem; namely the Poisson problem. The development will highlight the similarities and differences between FEA and Isogeometric Analysis.

3.1.1 The Poisson Problem

Th **Poisson problem** [35, p.103] is an elliptic partial differential equation with many applications in the sciences. On a domain Ω in two dimensions it is defined as;

$$\begin{aligned} \nabla^2 u(x,y) &= -f(x,y) \\ u(x,y) &= g \text{ on } \Gamma_D, \\ \nabla u(x,y)\cdot \mathbf{n} &= h \text{ on } \Gamma_N, \\ \beta u(x,y) + \nabla u(x,y)\cdot \mathbf{n} &= r \text{ on } \Gamma_R. \end{aligned} \tag{29}$$

where f is a given function defined on Ω, $\partial\Omega = \overline{\Gamma_D \cap \Gamma_N \cap \Gamma_R}$ is the boundary of the domain, and $\mathbf{n}$ is the outwards unit normal vector on the boundary. Γ_D represents the part of the boundary with Dirichlet boundary conditions, Γ_N represents the Neumann boundary conditions, and Γ_R the Robin conditions.

3.1.2 The Weak Form

Formulating this problem in a numerically solvable way follows the **Galerkin method** [27]. The Galerkin method is at the very heart of FEA. A thorough introduction to the Galerkin method and FEA is not presented here, but can be found in many texts, amongst others: [44, ch. 2.3], [38, ch. 6], [30], [42], [47] and [33]. However, in general terms, equation (29) is reformulated into a weak form by multiplying by a test function v and integrating over Ω

$$\iint\limits_{\Omega} \nabla^2 u \cdot v \,\mathrm{d}x\,\mathrm{d}y \quad = -\iint\limits_{\Omega} f \cdot v \,\mathrm{d}x\,\mathrm{d}y \tag{30}$$

integrating by parts on the left hand side and rearranging we obtain

$$\iint_{\Omega} \nabla u \cdot \nabla v \,\mathrm{d}x\,\mathrm{d}y = -\iint_{\Omega} fv \,\mathrm{d}x\,\mathrm{d}y + \iint_{\Gamma_N} vh \,\mathrm{d}\Gamma + \iint_{\Gamma_R} vr \,\mathrm{d}\Gamma - \beta \iint_{\Gamma_R} v \cdot u \,\mathrm{d}\Gamma \tag{31}$$

Boundary conditions will be discussed shortly, but in this thesis Dirichlet boundary conditions are used. Thus the integrals concerning the Robin and Neumann boundary conditions are set equal to zero, and we are left with

$$\iint_{\Omega} \nabla u \cdot \nabla v \,\mathrm{d}x\,\mathrm{d}y = \iint_{\Omega} f \cdot v \,\mathrm{d}x\,\mathrm{d}y. \tag{32}$$

This is called the **weak form**, and can be rewritten as

$$a(v,u) = L(v), \tag{33}$$

where

$$a(v,u) = \iint_{\Omega} \nabla u \cdot \nabla v \,\mathrm{d}x\,\mathrm{d}y, \tag{34}$$

and

$$L(v) = \iint_{\Omega} f \cdot v \,\mathrm{d}x\,\mathrm{d}y. \tag{35}$$

Obviously there are some restrictions to precicely which functions u and v are allowed to be. It turns out that they are both in the **Sobolev space** $H^1(\Omega)$ [16, p. 258], which implies that they are once differential, and

$$\int_{\Omega} \nabla u \cdot \nabla u \mathrm{d}\Omega < +\infty. \tag{36}$$

The Dirichlet boundary conditions are worked into these functions, and we choose the test function v to vanish on the boundary, whilst $u = g$ on the boundary. The collection of u - which are called trial solutions, $\mathcal{S}$, is then:

$$\mathcal{S} = \{u | u \in H^1(\Omega), u|_{\partial\Omega} = g\}. \tag{37}$$

And, the collection of v which are called weighting functions, $\mathcal{V}$:

$$\mathcal{V} = \{v | v \in H^1(\Omega), v|_{\partial\Omega} = 0\}. \tag{38}$$

Clearly there is an infinite number of functions satisfying the conditions of $\mathcal{S}$, and $\mathcal{V}$. The catch of the Galerkin's method is now to construct finite-dimensional approximations of $\mathcal{S}$, and $\mathcal{V}$, which implies that one looks for a solution amongst some finte-dimensional set of functions. Frequently, $\mathcal{S}$, and $\mathcal{V}$ can be chosen to be the same space. In the implementation of Isogeometric Analysis in this project, these finite-dimensional set of functions are chosen to be a set of B-spline basis functions spanning the domain Ω.

3.2 B-spline Basis Functions

Let us now establish B-splines as our basis funtions. We are free to choose the finite dimensional approximations to $\mathcal{S}$ and $\mathcal{V}$ from equations (37) and (38) respectively. For this thesis, both $\mathcal{S}$ and $\mathcal{V}$ will be chosen to be the set of B-spline basis functions spanning the basis Ω. These basis functions, in parameter space, are defined as

$$N_{i,j;p1,p2}(\xi,\eta) = N_{i,p_1}(\xi)N_{j,p_2}(\eta), \tag{39}$$

where $N_{i,p_1}(\xi)$ and $N_{j,p_2}(\eta)$ are the B-spline basis as defined in equation (2) in the ξ and η direction and with polynomial order p_1, p_2 respectively and with one knot vector corresponding to each parametric direction. The general shape of nine basis functions for $p_1 = p_2 = 2$, is shown in figure 9.

It will be necessary to adopt a global indexing of the basis functions in equation (39) such that

$$N_{\hat{i}} = N_{i,p_1;j,p_2}(\xi,\eta), \tag{40}$$

where the polynomial degrees are implied. A system to map global $\hat{i}$ to local i,j will be implemented, and is central to the code. The i,j coordinates will correspond to the knot indexes of the the lower left corner of the span of $N_{i,j}$.This mapping is represented as a matrix GToL which in row $\hat{i}$ contains the two values i,j. In addition, a list of global basis functions that are nonzero in a given element e is also stored in a matrix. Any subsequent references to a N with one subscript will be referring to this notation, still representing a basis function spanned by two seperate B-splines. The global numbering $N_{\hat{i}}$ will be analog to the numbering of elements, starting from bottom left in parameter space (see figure 10).

3.3 Elements in Isogeometric Analysis

The concept of elements is more ambiguous in Isogeometric Analysis than it is in FEM. One can recall that a B-spline basis function is defined on $p+1$ knot

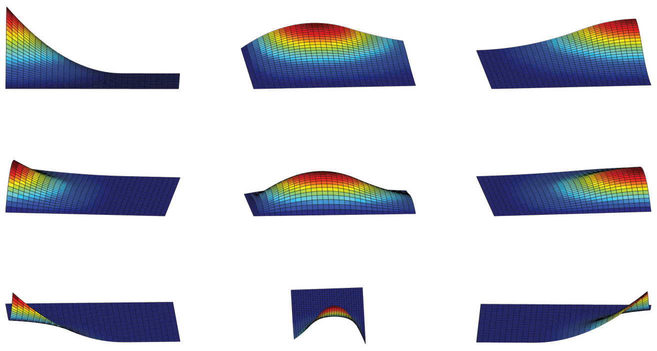


Figure 9: Nine basis functions$N_{\hat{i}}(\xi, \eta)$ when $p_1 = p_2 = 2$. The bottom and left basis functions are on the border at their bottom and left sides respectively.

spans in each parametric direction. The convention of [13] is adopted, defining elements to be the span of knots in the knot vector. One must then note that each basis function is defined to span several elements.

The two knot vectors will always span a rectangle in parameter space. To perform numerical integration, the elements in parameter space need to be mapped to an integration element or a parent element to perform the Gaussian Quadrature (se section 3.8), similar to FEM. However, one also needs a mapping from the parameter space to the physical space. This is done in the regular B-spline fashion of implementing control points, and shall be discussed in more detail in section 3.7. An overview of the spaces, element numbering and mappings necessary, as well as the span of a basis function is presented in figure 10.

3.4 Matrix Formulation

Once the set of B-spline basis functions as defined above are chosen as the finite-dimensional approximation to the trial solution and weighting function spaces, we can proceed in the familiar FEM way of matrix representation. The solution that is wanted, which will be called d will be built by summing basis functions multiplied by a factor (a weighting of that basis function) d_i,

$$d = \sum_{i=1}^{n_{eq}} d_i N_i, \tag{41}$$

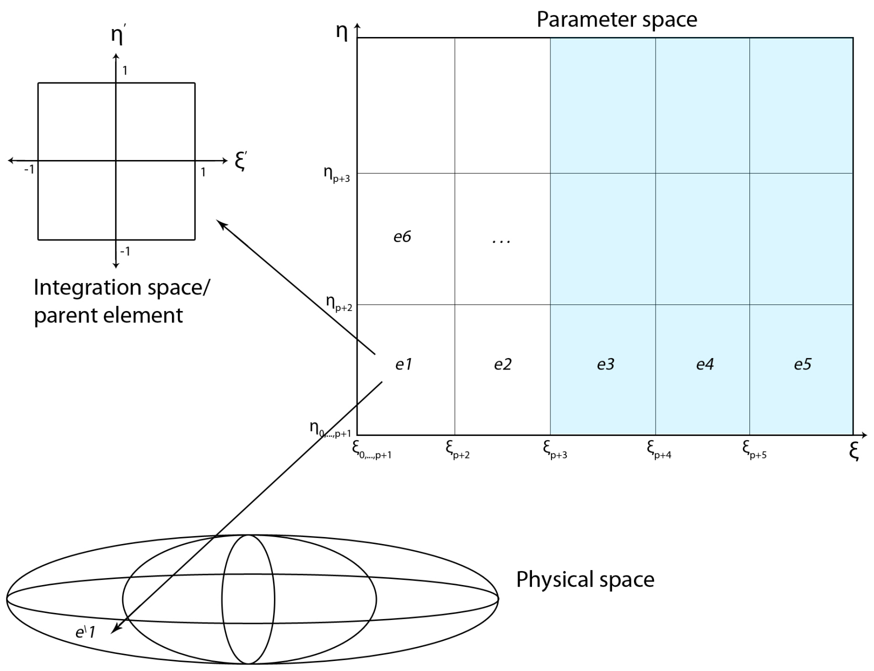


Figure 10: An overview of the three spaces in use in Isogeometric Analysis, as well as element numbering. The shaded blue area corresponds to the span of basis function $N_{\xi_{p+3},2;\eta_{p+1},2}$

where n_{eq} is the number of spline functions. Also, the weighting functions v will now also be B-spline basis functions N_j. Thus, we can rewrite the weak formulation of equations (33), (34) and (35) as,

$$a(d, N_j) = l(N_j). \tag{42}$$

Inserting (41) for d gives

$$a(\sum_{i=1}^{n_{eq}} d_i N_i, N_j) = l(N_j). \tag{43}$$

It is clear from (34) that $a(\cdot, \cdot)$ is bilinear, meaning that one can write

$$\sum_{i=1}^{n_{eq}} d_i a(N_i, N_j) = l(N_j). \tag{44}$$

Thus, for $i, j = 1, \ldots, n_{eq}$, this can be represented in matrix form:

$$K_{i,j} = a(N_i, N_j) \tag{45}$$

$$F_i = L(N_i) \tag{46}$$

and;

$$\mathbf{K} = [K_{i,j}], \tag{47}$$

$$\mathbf{F} = \{F_i\}, \tag{48}$$

$$\mathbf{d} = \{d_i\}, \tag{49}$$

where **K** is the familiar stiffness matrix, **F** the force vector, and **d** the displacement vector famililar from FEM and the matrix equation to solve is:

$$\mathbf{Kd} = \mathbf{F}. \tag{50}$$

3.5 Imposing Boundary Conditions

3.5.1 Homogeneous Dirichlet Boundary Conditions

The homogeneous Dirichlet boundary contidions are imposed after assembling the stiffness matrix and the force vector. A simple search through all of the basis functions identifies the basis functions that are located on the boundary. Then the equations and variables (rows and columns) corresponding to these basis functions in the stiffness matrix are altered so that the value of these basis functions is equal 0. Specifically, if for example basis function N_b is on the boundary, then row number b and column number b in the stiffness matrix is set equal to 0, whilst the diagonal entry is set to $\mathbf{K}_{b,b} = 1$, and $\mathbf{F}_b$ is set equal to zero.

3.5.2 Dirichlet Boundary Conditions

For non-homogeneous Dirichlet boundary conditions, say $g = g(x)$ on the boundary, one needs to calculate what the spline values at the boundary should be to enforce the given boundary condition. This can be done using a **Least Squares Approximation**. The method of Least Squares Approximation will not interpolate all points of the given boundary, but will minimize the error at the data points - at least in a least squares sense. The problem can be formulated as follows [34, Problem 5.34]: Given data $(x_i, y_i)_{i=1}^{m}$ with $x_1 < \cdots x_m$, and a n-dimensional spline space $\mathbb{S}_{p,t}$, find a spline g that minimizes

$$\sum_{i=1}^{m} (y_i - h(x_i))^2. \tag{51}$$

where h is a spline in $\mathbb{S}_{p,t}$.

This least square minimization problem can be written in terms of matrices as

$$\min \|\mathbf{Ac} - \mathbf{b}\|^2, \tag{52}$$

where $a_{i,j} = N_j(x_i)$, and $b_i = y_i$. This problem always has a solution $\mathbf{c}^*$ which is found by solving

$$\mathbf{A}^T\mathbf{Ac}^* = \mathbf{A}^T\mathbf{b}. \tag{53}$$

This method can then be used along each boundary to closely approximate the Dirichlet boundary conditions. This only holds for sufficiently smooth boundary conditions. The approximation will only be interpolating at the endpoints.

For the Poisson problem, regular Dirichlet boundary conditions are sufficient. For higher order differential equations, however, both the values u and the derivatives $\nabla u \cdot \mathbf{n}$ at the boundary must be provided for the solution to exist uniquely. This is slightly more intricate. However, in the case that $\nabla u \cdot \mathbf{n} = 0$, we can assure that both the values of u and $\nabla u \cdot \mathbf{n} = 0$ is satisfied at the boundary by letting the control points on the boundary be equal to their known or calculated value of u at the border whilst at the same time setting the next control point inward from the boundary equal to the same value as u. This is shown in figure 11.

3.5.3 Multi Point Restraints

In the last part of this thesis, periodic boundary conditions are applied, which means that the right boundary is linked to the left, and the top boundary is linked to the bottom. To achieve this, a multi point restraint method called the Master - Slave method is applied. The method is explained as follows [18]: Given a list of constraints on the form

$$u_1 = u_7,\ u_3 = u_9,\ u_5 = u_9 + u_6, \tag{54}$$

one needs to assign slave Degrees Of Freedoms (DOFs) and master DOFs. For example, let u_7, u_9, u_5 be the slave DOFs in the respective equations. Then one solves each constraint as

$$u_{slave} = u_{master}. \tag{55}$$

This gives

$$u_7 = u_1,\ u_9 = u_3,\ u_5 = u_9 + u_6, \tag{56}$$

respectively. Problems arise when the master DOFs on the right hand side are slaves in other equations - such as in the equation $u_5 = u_9 + u_6$, where u_9 is the

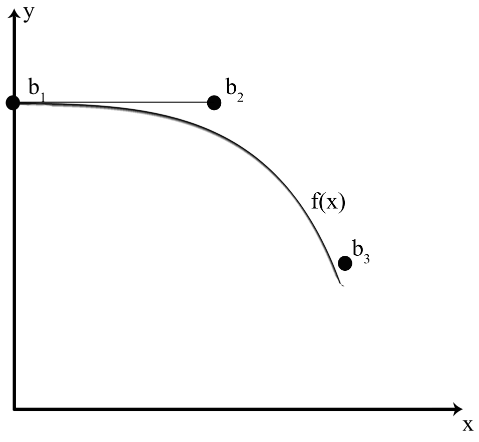


Figure 11: Illustration of setting $\nabla u \cdot \mathbf{n} = 0$ on the boundary by letting the two DOFs nearest to the boundary attain the same value.

slave in the second equation. In this case one needs to back-trace so that all right hand sides of all the constraints only contain DOFs that are masters everywhere. In this set of equations one thus replaces u_9 with u_3 as we know from the second constraint, giving

$$u_7 = u_1,\ u_9 = u_3,\ u_5 = u_3 + u_6. \tag{57}$$

Then, we are able to assemble a transformation matrix as follows:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \tilde{u_5} \\ u_6 \\ \tilde{u_7} \\ u_8 \\ \tilde{u_9} \end{bmatrix} \tag{58}$$

If our solution vector is $\mathbf{u}$, we can write this in matrix notation as

$$\mathbf{u} = \mathbf{T}\hat{\mathbf{u}} \tag{59}$$

The slave variables are "dropped" from the system, but their place is retained in the solution array, simplifying the work of setting their values at the end of the computations. This is done by using the matrix transformation $\mathbf{T}$. Now, if our assembled Isogeometric Analysis system is $\mathbf{Ku} = \mathbf{f}$, we can apply the multipoint constraints by restating the system as

$$\hat{\mathbf{K}}\hat{\mathbf{u}} = \hat{\mathbf{f}}, \tag{60}$$

where

$$\hat{\mathbf{K}} = \mathbf{T}^{\mathbf{T}}\mathbf{K}\mathbf{T}, \tag{61}$$

and

$$\hat{\mathbf{f}} = \mathbf{T}^{\mathbf{T}}\mathbf{f}, \tag{62}$$

This multipoint master-slave method can be used to set periodic boundary conditions. All values on the left boundary must be set equal to their corresponding variables on the right boundary, and similarily for the bottom and top boundraries. In addition, all corners must be set equal. To also apply the derivative boundary conditions in the periodic case, one needs to apply the following constraints for each corresponding pair of a boundary DOFs and the next DOF inside the domain in the normal direction.

$$\frac{u_{lb+1} - ulb}{\Delta x_1} = \frac{u_{rb} - u_{rb-1}}{\Delta x_2} \tag{63}$$

Where in this example u_{lb} is on the left boundary, u_{lb+1} is the first DOF to the right of the left boundary, u_{rb} is on the corresponding right boundary DOF, and u_{rb-1} is the first DOF to the left of the right boundary. For the application later in this thesis, a uniform grid is assumed, thus cancelling the Δx's. In addition to these derivative restraints, the derivatives near the corners must also carefully be preserved. The DOFs selected as masters and slaves can be seen in figure 12. All of the constraints were assembled into a transformation matrix using the method introduced above, and applied to the system.

3.6 Error Analysis

The aim of error estimation in Isogeometric Analysis is firstly to show the same error convergence results as for classical FEA. This is not at all trivial in a mathematical sense, and shall not be deducted here. The interested reader can consult Bazilevs' paper [4] for the full mathematical proceedings. However, Bazilevs' results show that the error in Isogeometric Analysis converges in the same way as in classical FEA. The result is that if the error e is defined as

$$e = u_{approx} - u_{exact}, \tag{64}$$

where u_{approx} is the approximate solution and u_{exact} is the exact solution, the error should behave as

$$\|e\| \leq Ch^p. \tag{65}$$

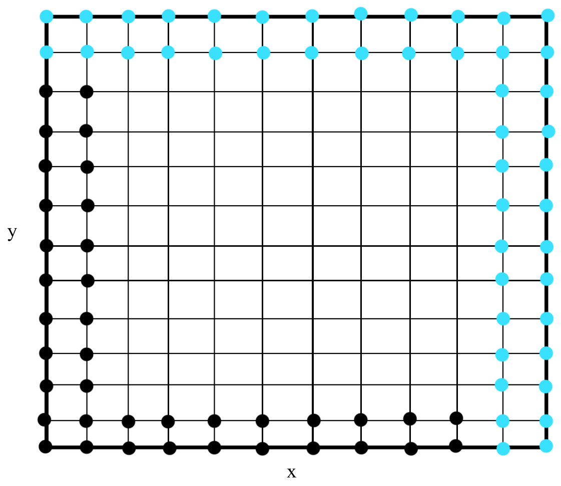


Figure 12: Illustration of master and slave assignments. Let x and y be the parametric directions, and each intersection represent a DOF. Light blue dots define the slave DOFs, while black dots define the elected master DOFs

C is some constant, h is a measure of the size of elements, p is the polynomial order, and the norm being used is the one corresponding to the Sobolev space, the **energy norm**, defined as:

$$\|e\|^2 = a(e, e), \tag{66}$$

$a(\cdot, \cdot)$ being defined exactly as in equation (34), giving;

$$\|e\| = \sqrt{\iint_{\Omega} \nabla(e) \cdot \nabla(e) \mathrm{d}A}. \tag{67}$$

The significant term in equation (65) is h^p, and thus, in terms of the energy norm $\|e\|$, we expect the error to behave as

$$\|e\| \approx \mathcal{O}(h^p). \tag{68}$$

Now, taking log of equation (65) on both sides,

$$\log(\|e\|) = \log(C) + p \log(h), \tag{69}$$

this shows that plotting log of the energy norm against log of the element size, should give a straight line, with slope equal to the polynomial degree p.

This error analysis results proves to be very significant. This is because, reducing h by a factor of 2, which implies dividing all elements of FEM, and halving the knot spans in each direction in Isogeometric Analysis, the number of degrees of freedom (equations to be solved) increases significantly greater in FEM than the number of degrees of freedom in Isogeometric Analysis. This actually means that Isogeometric Analysis will converge at the same rate as FEM, but at lower computational cost! [13]

3.7 Mapping

As mentioned in previous sections, two mappings are performed, as three spaces are used; the integration reference element, the parametric space, and the cartesian coordinates (physical space), see figure 10. The mapping from parameter space to the integration element is a trivial affine transformation, and involves placing the integrationpoints correspondingly in each parametric element, and mulitplying by a area-factor $A_{ParamElement}/A_{ReferenceElement}$. However, the mapping from cartesian coordinates and into parameter space and vice verca requires the use of the Jacobi matrix and the Jacobian [32]. The Jacobian matrix is defined as

$$\mathbf{J} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{pmatrix}, \tag{70}$$

and represents the transformation from cartesian coordinates into parameter space. The Jacobian is defined as the determinant of $\mathbf{J}$, $|\mathbf{J}|$. The value of the Jacobian turns out to be the area-scaling factor of the transformation. The inverse tranformation - the forward mapping from parameter space into physical space - $\mathbf{G}$ will also be needed. $\mathbf{G}$ is related to $\mathbf{J}$ by,

$$\mathbf{G} = (\mathbf{J}^T)^{-1}, \tag{71}$$

or, the inverse of the transpose of $\mathbf{J}$.

Thus, when implementing the solver using mapping from parameter space into physical space, several implementation adjustments must be made. Firstly, the Gauss integration points of each element in parameter space must be mapped to their cartesian coordinate counterparts. This is done by the standard spline implementation

$$\begin{bmatrix} x \\ y \end{bmatrix} = \sum_{i,j} N_i(\xi) N_j(\eta) \mathbf{B}_{i,j}, \tag{72}$$

where $\mathbf{B}$ are the control point coordinates. This mapping is needed both in plotting the results, and in evaluating the right-hand side of the poission equation, f. Secondly, assembling the stiffness matrix $\mathbf{K}$, one needs to integrate (equation (34))

$$a(N_i, N_j) = \iint\limits_{\Omega} \nabla N_i(\xi,\eta) \cdot \nabla N_j(\xi,\eta)\, \mathrm{d}x\, \mathrm{d}y. \tag{73}$$

However, $\nabla N_i(\xi,\eta)$ and $\nabla N_j(\xi,\eta)$ are calculated in parameter space, whereas the integral is performed in cartesian coordinates. Therefore the foward mapping $\mathbf{G}$ must be applied. In addition, the area factor (the Jacobian) must be applied, giving

$$a(N_i, N_j) = \iint\limits_{\Omega} \mathbf{G}\nabla N_i(\xi,\eta) \cdot \mathbf{G}\nabla N_j(\xi,\eta) |\mathbf{J}|\, \mathrm{d}x\, \mathrm{d}y. \tag{74}$$

Thirdly, assembling the force vector $\mathbf{F}$, the integral (equation (35))

$$L(N_i) = \iint\limits_{\Omega} f(x,y) \cdot N_i(\xi,\eta)\, \mathrm{d}x\, \mathrm{d}y, \tag{75}$$

is used. Here, f is evaluated at (x, y), where x and y are the Gauss points mapped into cartesian coordinates corresponding to (ξ, η). Thus, since no derivatives are present, only the Jacobian is needed resulting in:

$$L(N_i) = \iint\limits_{\Omega} f(x,y) \cdot N(\xi,\eta)_i |\mathbf{J}|\, \mathrm{d}x\, \mathrm{d}y. \tag{76}$$

As mentioned, the Jacobian $\mathbf{J}$ represent the area scaling at the point evaluated. A set of knot vector values and corresponding control points are presented in figure 13, giving mapped integration points in cartesian coordinates as in figure 14. As one can see here, the points are further apart near the edges and the corners, and closer together, but evenly distributed throughout the interior. The Jacobian at each point towards the edge is thus expected to be rather large, as the area increases in these regions. The Jacobian should in this example be smaller and nearly constant for the interior points. These expectations are verified to be true, as the value of the function creating the Jacobian was plotted at each of the points of figure 14, shown in figure 15.

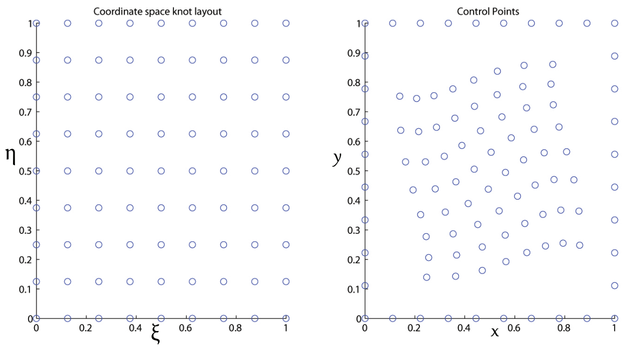


Figure 13: Knot vector intersections, and corresponding control points

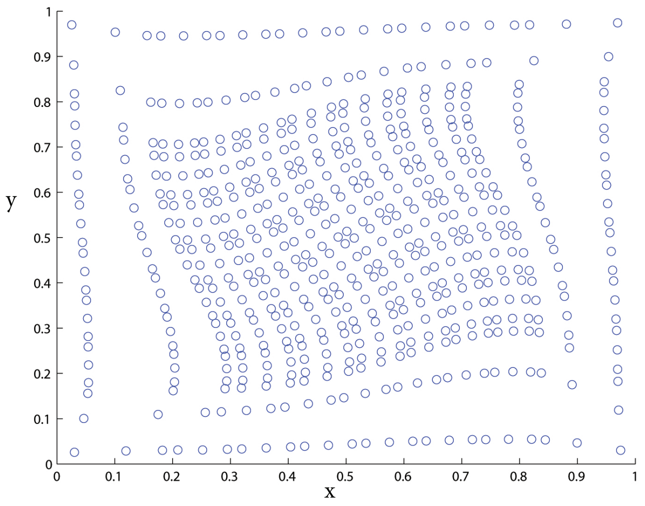


Figure 14: Integration points mapped into cartesian coordinates

3.8 Numerical Integration

Numerical integration is central in building the matrices required for the Isogeometric Analysis. In this thesis, the standard **Gauss Quadrature** [31, Chapter 17.5] is used, as is the common approach in finite element analysis. Gauss quadrature tries to obtain the best numerical estimate by picking optimal points

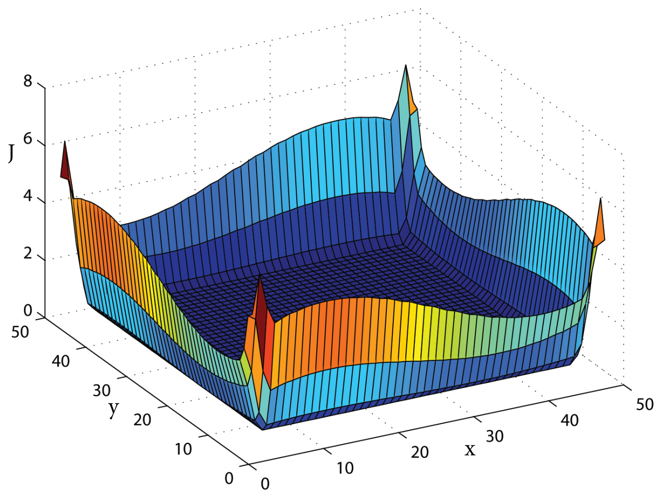


Figure 15: The value of the Jacobian at each of the integration points

to evaluate the function being estimated at. The Fundamental Theorem of Gaussian quadrature [50] says that the optimal points to evaluate the function at, are the zeros of the Legendre polynomial. By weighting these points with appropriate weights (often called the Christoffel Numbers[8]), an optimal approximation is obtained and the Gaussian quadrature fits all polynomials up to degree $(2n_1 - 1)(2n_2 - 1)$ exactly, where n_1 and n_2 are the number of points evaluated in each coordinate direction in 2D. The general formula on a two dimensional domain is

$$\int_{-1}^{1} f(x,y)\,\mathrm{d}x\mathrm{d}y \approx \sum_{i=1}^{n_1}\sum_{j=1}^{n_2} w_{i,j} f(t_i,u_j), \tag{77}$$

n_1 and n_2 are the number of points used in each direction, $w_{i,j}$ are the weights, and t_i, u_j are the ideally placed evaluation points. It can be noted that the integration domain is the square$[-1,1]\times[-1,1]$ integration; so any other integration domain needs to be mapped into this domain. The placement of the Gauss evaluation points on $[0,1]\times[0,1]$ is shown in figure 16. For this thesis, the number of Gauss points was adjusted to equal $(p_1+1)(p_2+1)$ at runtime, where p_1 and p_2 are the B-spline polynomial degree in each of the two parametric directions, which ensures a suitable accuracy.

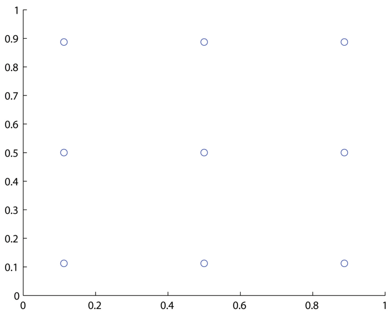


Figure 16: Locations of the 9-point 2D Gauss evaluation points on the square $[0,1] \times [0,1]$

It will be noted, as found by Hughes, Reali and Sangalli [26], that Gauss integration is not ideal in the same way for Isogeometric Analysis as it is for polynomials and the finite element method. The finite element basis is C^0-continuous accross elements, whereas the B-spline basis usually has higher degrees of continuity accross elements, something that would have to be accounted for in an ideal method specifically for Isogeometric Analysis. However, within the scope of this thesis, Gaussian quadrature is used, simultaneously keeping with FEM-tradition.

4 Solver Implementation Overview

This sections aims to give a brief overview of the structure of the Isogeometric Analysis solver implemented in this project. A flowchart of the program is presented in figure 17, which outlines the main structures of the program. The code

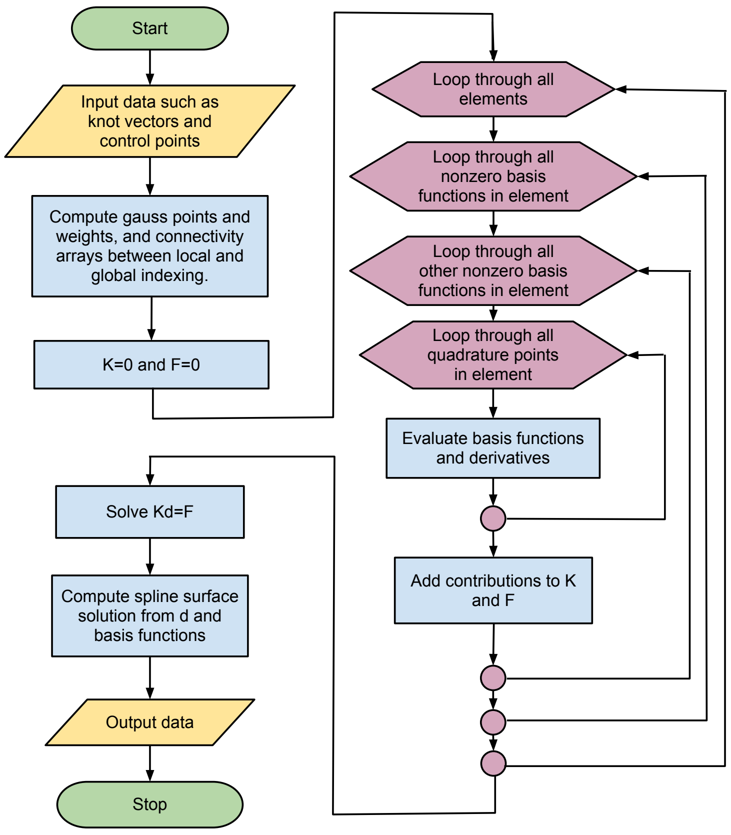

Start
Input data such as knot vectors and control points
Compute gauss points and weights, and connectivity arrays between local and global indexing.
K=0 and F=0
Loop through all elements
Loop through all nonzero basis functions in element
Loop through all other nonzero basis functions in element
Loop through all quadrature points in element
Evaluate basis functions and derivatives
Add contributions to K and F
Solve Kd=F
Compute spline surface solution from d and basis functions
Output data
Stop

Figure 17: Flowchart of program

architecture is very modular, some important functions created are described in table 1. Most of these functions are used at startup, in the order listed except for **getBoundarySplines**, which is used after the assembly of the stiffness matrix and force vector is completed. The **getSplineValsAndDerivs** function is called before the assembly starts, calculating all the B-spline basis function values at all the integration points that will be needed. This reduces the "evaluation of basis functions and derivatives" in the innermost loop of the flowchart to table look-ups in the pre-calculated spline value matrices. It will also be noted here that the program is compatible with different polynomial orders and numbers of basis equations in the two parametric directions independent of each other throughout, even though many examples are run with the sufficient accuracy of $p_1 = p_2 = 2$

Function Name	Inputs	Outputs
getGeometry	p_1, p_2, n_1, n_2	Knot vectors and control points. Only this function needs to be changed to change the entire geometry.
getConnectivity	n_1, n_2, p_1, p_2	GToL - Global To Local mapping of spline basis function mapping. SinE - List of the nonzero spline basis functions in each element.
getGaussPoints	knotvector1, knotvector2, n_1, n_2, p_1, p_2	Gauss point weights and an array of Gauss points in parameter space, one for each parameter direction.
getSplineValsAndDerivs	p, knotvector, Gausspoints	Matrix N: [(basis function index) x (Gauss points)] containing values of each basis function value at each Gauss point, and matrix ND which is the same, but with the derivative values at each point for each basis function. For the higher order solver, NDD - the second derivative values - was also returned.
getBoundarySplines	GToL, n_1, n_2	Array of global index to all boundary basis functions. Depending on which implementation of Dirichlet boundary conditions were used, this function can return either a list of all (global) boundary basis conditions, one array per side of the boundary, or a list containing the mapping from master to slave DOFS

Table 1: Important functions. p, p_1 and p_2 are the polynomial degrees, n_1,n_2 are the number of spline basis functions in each parametric direction, other input values are defined as outputs of other functions in the table

5 Verification of Solver

One aim of this thesis is to use an Isogeometric Analysis solver to experiment with the Cahn-Hilliard equation, which will be introduced in the next sextion. The Cahn-Hilliard equation is rather complex, and has no easily obtainable known exact solution. However, in developing a solver from scratch - it is vital to be able to verify the solver at different stages of development to find errors and check if the solutions are as expected. This section presents the results of two major verification steps, namely the solution of the Poisson problem, and the solution of the biharmonic equations. The solutions presented for the Poisson problem verifies the core of the solver, and its applications to second order partial differential equations. The solutions presented for the biharmonic equations verifies the solver for fourth order partial differential equations - which is the class of equations to which the Cahn-Hilliard equation belongs. It is noted that as the Cahn-Hilliard solver will operate in parameter space, applying no mapping to a physical space, the verifications below are also perfomed directly on the parameter space, setting the paramter space and the physical space equal. The domain used is $\Omega = [0, 1] \times [0, 1]$

5.1 Verification of Results on the Homogeneous Poisson Problem

Implementing the solver for the Poisson problem (see equation (29)), with homogeneous Dirichlet boundary conditions, a suitable solution would have to be zero on the boundary, and have a readily available exact solution. Therefore the exact solution of u was chosen to be

$$u(x, y) = \sin(2\pi x)sin(2\pi y) \tag{78}$$

which is zero at the boundary of Ω. Then,

$$\nabla^2 u(x, y) = -8\pi^2 \sin(2\pi x)sin(2\pi y), \tag{79}$$

which means that the f on the right hand side of the Poisson equation (equation (29)) would have to equal

$$f = -8\pi^2 \sin(2\pi x)sin(2\pi y), \tag{80}$$

for the test problem implementation.

The results of a coarse implementation, using only 6 basis functions with polynomial order 2 in each direction, along with the exact solution, and the pointwise

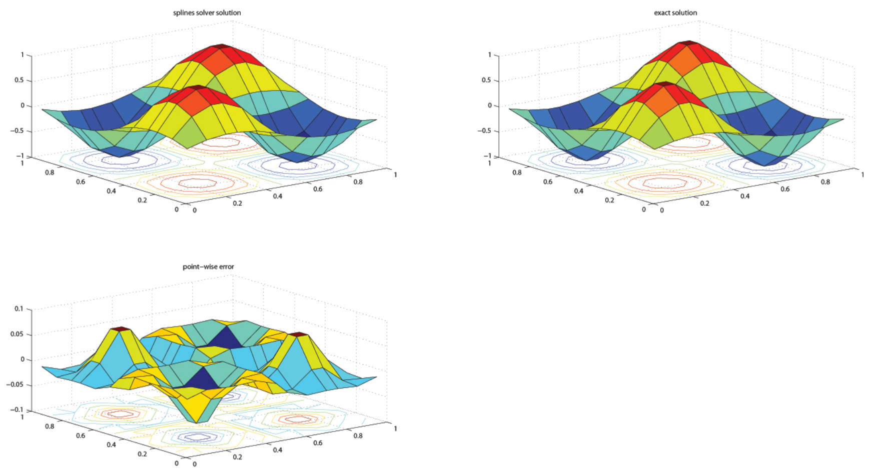


Figure 18: Results with 6 basis functions in each direction and $p_1 = p_2 = 2$. The scale on the vertical axis of the point-wise error plot here is 0.1

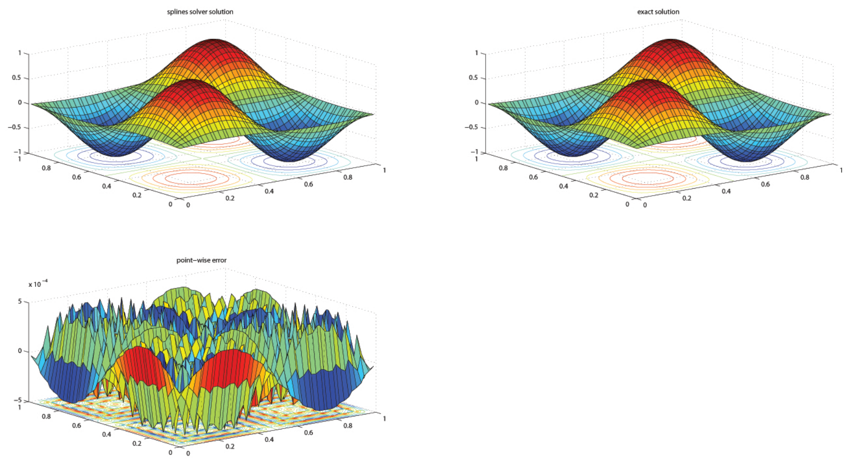


Figure 19: Results with 18 basis functions in each direction and $p_1 = p_2 = 2$. The scale on the vertical axis of the point-wise error plot here is 10^{-4}

error can be seen in figure 18. Results for a finer implementation, using 18 basis functions in each directon, and $p = 2$ are shown in figure 19. An error analysis was computed doing several runs with different element size h and p, to study the behaviour compared to the expected error described in section 3.6.As we can see in figure 20, and table 2, the error behaves exactly as expected (see section 3.6), in effect validating the results of the program for the Poisson problem. The log-log graph is linear, and the slope of each line is closely equal to p. The slight misfit in the largest value of h is attributed to inexact numerical integration, causing (primarily overintegration) as the extreme points of $f(x, y)$ correspond with the centre - and most heavily weighted - Gauss integration points at this low resolution. These errenous data points at $h = 0.5$ are omitted in the generation of the values in table 2.

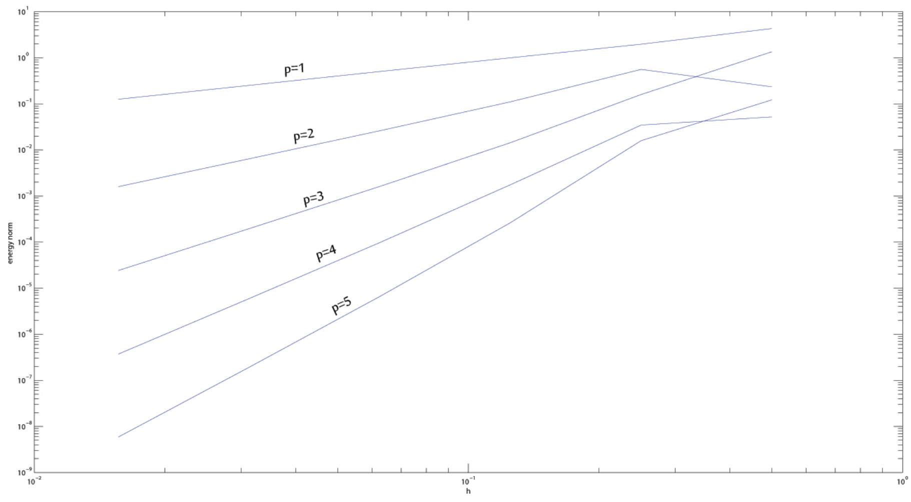


Figure 20: Error plot for homogeneous Dirichlet conditions on the Poisson problem for different polynomial degrees p. $h = 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625$

Polynomial order	Slope of line in error plot
$p = 1$	0.9920
$p = 2$	2.1005
$p = 3$	3.1560
$p = 4$	4.1215
$p = 5$	5.3140
$p = 6$	6.0735

Table 2: The slope of the error plot lines of the homogeneous Dirichlet Poisson problem showing very good agreement with their expected value. The value for $p = 6$ was calculated at a later time than the figure.

Polynomial order	Slope of line in error plot
$p = 1$	1.3900
$p = 2$	2.1170
$p = 3$	3.2520
$p = 4$	4.2975
$p = 5$	5.1965
$p = 6$	6.3954
$p = 7$	7.0884

Table 3: The slope of the error plot lines of the non-homogeneous Dirichlet Poisson problem showing very good agreement with their expected value.

5.2 Verification of Results on the Non-Homogeneous Poisson Problem

The same convergence results were obtained for the homogeneous Poisson problem, thus allowing any values for the dirichlet boundary conditions. The convergence results are shown in figure 21 and table 3

5.3 Verification on Higher Order Equations

The Cahn-Hilliard equation to be studied shortly is a fourth order partial differential equation with no easily verified exact known solution. Hence, the next step is to verify the solver for a fourth order partial differential equation that has known solutions. For this purpose, the Biharmonic Equation was chosen.

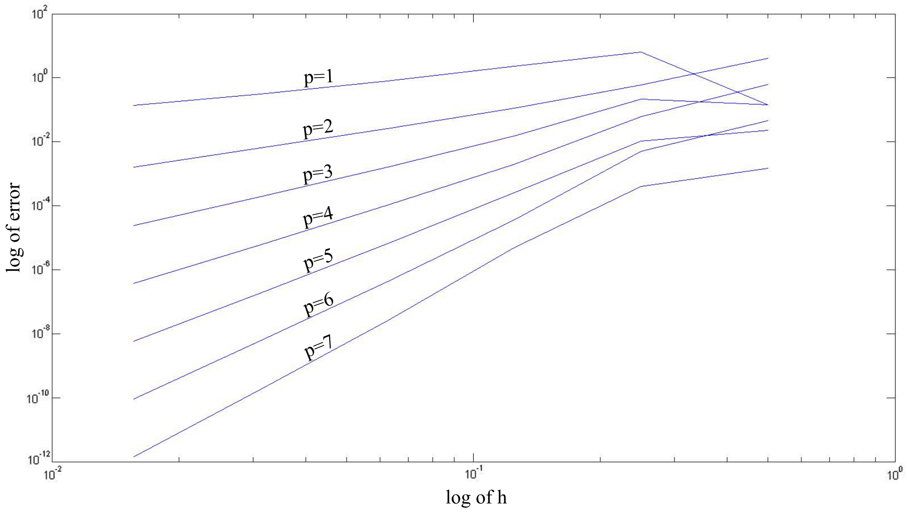


Figure 21: Error plot for non-homogeneous Dirichlet Poisson with different polynomial degrees p. $h = 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625$

5.3.1 The Biharmonic Equation

The Biharmonic Equation arises in the field of continuum mechanics [46], involving applications such as thin plate theory and flexure of elastic plates. In two dimensions it is also used to model a simplification of the Navier-Stokes equations for slow, viscous flow problems with Reynolds Number $R_e = \rho UL/\mu$, $R_e \ll 1$. [46, Chapter 8.], [41] The Biharmonic equation is given in strong form as follows;

$$\Delta^2 u(x,y) = f(x,y) \text{ on } \Omega, \tag{81}$$

where $\Omega = [0,1]\times[0,1]$ as stated earlier. As mentioned in the earlier discussion of boundary conditions, fourth order partial differential equations need two sets of boundary conditions for the solution to exist uniquely. The Dirichlet boundary conditions applied in this verification are

$$\begin{aligned} u(x,y) &= g \text{ on } \Gamma_D \\ \nabla u(x,y)\cdot \mathbf{n} &= 0 \text{ on } \Gamma_D, \end{aligned} \tag{82}$$

where Γ_D consists of the entire boundary, and function g is assumed to be given. Now we perform the same method as was done for the Poisson problem. We multiply (81) by a test function v integrate twice by parts, and obtain the following weak form: Find $u \in \mathcal{S}$ so that

$$\iint\limits_{\Omega} \Delta u \cdot \Delta v \,\mathrm{d}x\,\mathrm{d}y \quad = \iint\limits_{\Omega} f \cdot v \,\mathrm{d}x\,\mathrm{d}y. \forall v \in \mathcal{V}. \tag{83}$$

Here, the spaces $\mathcal{S}$ and $\mathcal{V}$ are as follows,

$$\begin{aligned} \mathcal{S} &= \{u | u \in H^2(\Omega), u|_{\Gamma D} = g\}, \\ \mathcal{V} &= \{v | v \in H^2(\Omega), v|_{\Gamma D} = 0\}. \end{aligned} \tag{84}$$

which is analouge to the spaces for the Poisson problem, except that here the functions u and v are required to be twice differentiable on Ω. This is easily ensured since splines (as mentioned earlier) have continous derivatives up to order $p - m$ at the ith knot, where m is the number of times the ith knot appears in the knot vector, and p is the polynomial order. Thus, since $m = 1$ for the spline implementation of the latter part of this thesis, we only require a polynomial order of at least $p = 3$ The rest of the development of obtaining a matrix formulation of this weak form follows exactly as for the Poisson problem.

5.3.2 Results

The exact solution opted for in the verification, which also satisfies the boundary conditions, is

$$u(x, y) = \cos(4\pi x) \cos(4\pi y) \tag{85}$$

Thus, the right hand side of (81) is obtained by repeatedly differentiating u, giving

$$f(x, y) = 512\pi^4 \cos(4\pi x) \cos(4\pi y). \tag{86}$$

The only neccessary modifications to the code is the production of second derivatives, the integration on the left hand side of (83), and the implementation of the boundary conditions. The Dirichlet boundary conditions were calculated and applied using the Least Squares approximation introduced earlier, using the exact solution along the boundaries as g. Then the method of setting $\nabla u \cdot \mathbf{n} = 0$ by making the values of each first interior point equal to the corresponding boundary point was applied. The solution of the solver can be seen compared to the exact solution for a coarse simulation of only 11 B-splines in each parametric direction in figure 22, and for a higher accuracy of 35 B-splines in each parametric direction in figure 23. The convergence of the error analyis is again as expected, shown in figure 24 and table 4

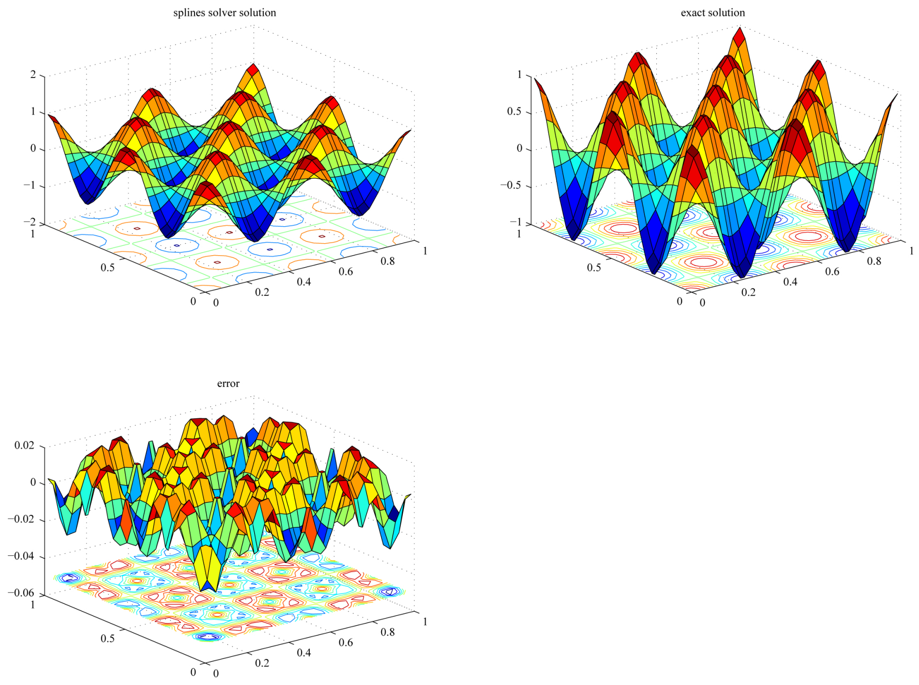


Figure 22: Results with 11 basis functions in each direction and $p_1 = p_2 = 3$. The scale on the vertical axis of the point-wise error plot here is 0.2

Polynomial order	Slope of line in error plot
$p = 3$	3.5214
$p = 4$	4.6172
$p = 5$	5.3374
$p = 6$	6.8094
$p = 7$	6.9864

Table 4: The slope of the error plot lines of the Biharmonic equation showing very good agreement with their expected value.

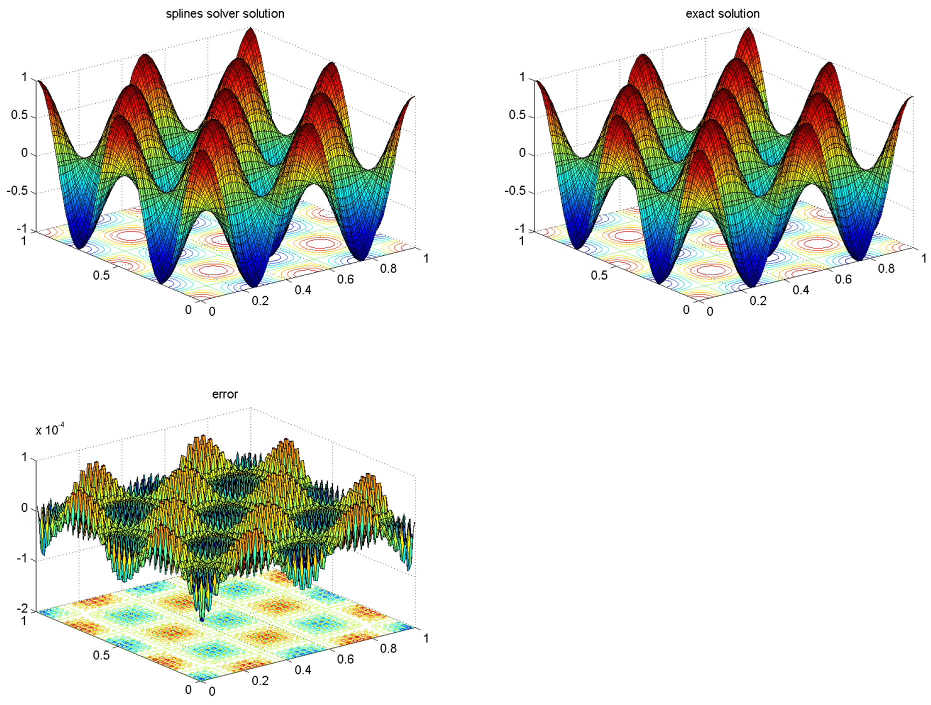


Figure 23: Results with 35 basis functions in each direction and $p_1 = p_2 = 3$. The scale on the vertical axis of the point-wise error plot here is 10^{-4}

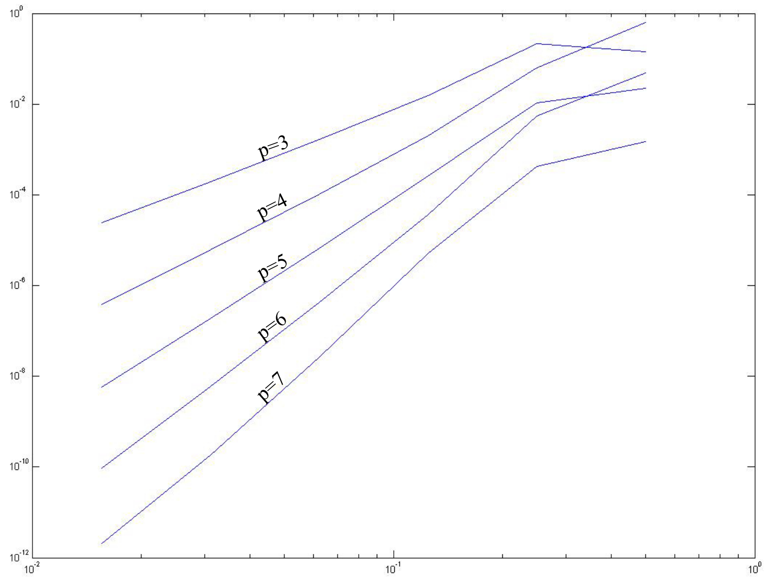


Figure 24: Error plot for different polynomial degrees $p > 3$. $h = 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625, 0.0078125$

6 Numerical Experiments

Higher order partial differential equations is an important class of equations, with applications in biomedical and engineering fields such as liquid-liquid problems, liquid-vapor problems, emulsification, canser growth and rotation-free thin shell theory [22]. However, such higher order partial differential equations have in general not been easily implemented in classical FEA. This is because their variational form contain higher order derivatives of the test functions which again implies the need of higher continuity in the basis functions. Traditionally, FEM has C^0 continuity across elements, which is not enough for solving higher order partial differential equations. Hence, work on higher order differential equations has mostly happened through finite difference and spectral techniques. [13] However, these techniques are quite limited, especially on more complicated domains. Thus, other solutions for (nearly) arbitrary domains are needed. FEA has seen some workarounds in the later years to solve higher order differential equations.The Mixed method [28], Discontinuous Galerkin method [52] and the Countinuous Discontinous Galerkin [15] methods are all workarounds for the fundamental limitation of C^0 continuity in FEA. However, there is no general method for solving higher order differential equations in FEM. The methods mentioned above all involve some kind of added complexity, either computational or mathematical, and require more degrees of freedom to be added - to the often already computationally large systems.

The remainder of this thesis will demonstrate Isogeometric Analysis' potentially vital role in solving higher order partial differential equations. With Isogeometric Analysis, there is nearly no added complexity in dealing with such equations. The number of degrees of freedoms remains fundamentally unchanged, and any desireable continuity C^n can be achieved. This very noteworthy advantage of Isogeometric Analysis will be demonstrated on the Cahn-Hilliard equation, which now will be introduced. It is noted that the Cahn-Hilliard equation has been succsessfully solved using FEM-workarounds such as those mentioned above or by reformulating the problem, examples can be found in [2], [51], and [14].

6.1 The Cahn-Hilliard Equation

The Cahn-Hilliard equation is most often used to model the spinoidal decomposition of binary liquids. A spinoidal decomposition is a process in which a homogeneous mixture spontaneously seperates into two phases. However, the equation has also been used in diverse fields such as image processing, planet formation and cancer growth [13]. There are two primary ways to model phase transition phenomena such as spinoidal decomposition; sharp interface models and phase-field models. Traditionally, sharp interface models have been used to model the development of an interface, such as a liquid-solid interaction. However, such a method creates a number of implementation difficulties, such as different differential equations on each side of the interface and discontinuities accross interfaces. Phase-fields models, however models the interfaces as smooth (relatively sharp) transition regions where surface tension is distributed[37]. The Cahn-Hillard equation is an example of a Phase-field model. The main motivation behind the Cahn-Hilliard equation is considering two seperate and competing mechanisms' effect on a binary liquid. Firstly, the system will try to minimize the chemical free energy. Secondly, the system will try to minimize the interface free energy. Thus, by considering these two contributions, Cahn and Hillard (1958, 1961) [7] were able to derive Cahn Hillard equation presented in the next section. A full derivation of the equation can be found in [6].

6.1.1 The Strong Form

The strong form of the Cahn-Hilliard equations can be formulated as follows [22], [13]: Let $\Omega \subset \mathbb{R}^d$, and $d = 2$ or 3. The boundary will be denoted as Γ as before. A binary mixture, as described above, is contained in Ω, and let c be the concentration of one of the components in Ω. Then, the concentration $c : \Omega \times (0,T) \to \mathbb{R}$ is given as

$$\begin{aligned} \frac{\partial c}{\partial t} = \nabla \cdot (M_c \nabla(\mu_c - \lambda \Delta c)) + &\quad \text{on } \Omega \times (0,T) \\ c = g &\quad \text{on } \Gamma \times (0,T) \\ M_c \lambda \nabla c \cdot \mathbf{n} = 0 &\quad \text{on } \Gamma \times (0,T) \\ c(\mathbf{x}, t) = c_0(x) &\quad \text{in } \Omega. \end{aligned} \tag{87}$$

Here,λ is a constant so that $\sqrt{\lambda}$ is a length scale of the problem. M_c is the mobility of the solution. Sometimes it is assumed constant, but herein the standard function as found in the original findings of Cahn and Hilliard [7] is used:

$$M_c = Dc(1-c) \tag{88}$$

D is a chosen constant with dimensions of diffusivity. The function μ_c represents the chemical potential. It is highly nonlinear, and is thus often approximated by a third or fourth order polynomial.[22] [13]. Here, however, the full thermodynamic version is preffered,

$$\mu_c = \frac{1}{2\theta} \log \frac{c}{1-c} + 1 - 2c. \tag{89}$$

θ is a dimensionless number representing T_c/T - the ratio between the critical temperature T_c of the system where the two phases have the same composition and the absolute temperature T. It turns out that for $\theta > 1$, the chemical potential will drive the phase separation to their pure forms, whilst for $\theta \leq 1$ there is only a single solution, namely constant concentration.

6.1.2 A Dimensionless Form

Before obtaining our weak form of the problem, it is useful to scale the problem to obtain a dimensionless form of the equation. Scaling problems can help identify dimensionless constants - relations that reduce the problems complexity in the sense of having to adjust fewer constants governing the system. We will need two scales, one for time and one for length. The dimensionless variables can be written as,

$$x^* = x/L_0, \; t^* = t/T_0. \tag{90}$$

L_0 is chosen to be a suitable length scale, and $T_0 = L_0^4/(D\lambda)$, as in [13, ch. 11]. Applying these new variables to (87), we obtain the dimensionless form of the Cahn-Hilliard equation:

$$\frac{\partial c}{\partial t^*} = \nabla \cdot (M_c^* \nabla(\mu_c^* - \Delta c)). \tag{91}$$

From now on, we shall use the dimensionless form, and hence drop the superscript $*$. One can note one further dimensionless number, α [22]. Setting

$$\alpha = \frac{L_0^2}{3\lambda}, \tag{92}$$

we see that the thickness of the interface boundraries are proportional to $\alpha^{-0.5}$. Now, if θ is fixed, the whole problem is charecterized by the value of α. The value of θ is chosen to be equal to 3/2 which is true for a physically relevant case [19]. Thus, α is somewhat similar to the Reynolds number in fluid dynamics - fully characterizing the solutions.

6.1.3 The Weak Form and Spatial Discretization

The weak form is obtained in the standard way, multiplying (91) by a test function v and integrating by parts. Assuming periodic or Dirichlet boundary conditions, the following weak from is obtained;

$$\iint_\Omega v\frac{\partial c}{\partial t}\,\mathrm{d}\Omega = -\iint_\Omega \nabla v(M_c\nabla\mu_c + \nabla M_c\Delta c)\mathrm{d}\Omega - \iint_\Omega \Delta v M_c \Delta c \mathrm{d}\Omega. \tag{93}$$

Now, from our B-spline basis functions, we have that

$$c = \sum_{i=1}^{n_{eq}} c_i N_i, \tag{94}$$

and

$$\frac{\partial c}{\partial t} = \sum_{i=1}^{n_{eq}} \frac{\partial c_i}{\partial t} N_i, \tag{95}$$

and that v also are spline functions N_j. Hence, the left hand side can be spatially discretisized in the familiar way:

$$\iint\limits_{\Omega} v \frac{\partial c}{\partial t} \mathrm{d}\Omega = \iint\limits_{\Omega} \sum_{i=1}^{n_{eq}} \frac{\partial c_i}{\partial t} N_i N_j \mathrm{d}\Omega, \tag{96}$$

and hence

$$\iint\limits_{\Omega} \sum_{i=1}^{n_{eq}} \frac{\partial c_i}{\partial t} i N_i N_j \mathrm{d}\Omega = \sum_{i=1}^{n_{eq}} \frac{\partial c_i}{\partial t} \iint\limits_{\Omega} N_i N_j \mathrm{d}\Omega, \tag{97}$$

with

$$\mathbf{M} = [M_{i,j}] = \iint\limits_{\Omega} N_i N_j \mathrm{d}\Omega \tag{98}$$

being the Mass matrix of the left hand side of the system. The right hand side of (93) cannot be discretisiced in this way because of the nonlinearities present, as will be discussed shortly, but we shall name the operator on the right hand side

$$\mathbf{A}(c) = -\iint\limits_{\Omega} \nabla v (M_c \nabla \mu_c + \nabla M_c \Delta c \mathrm{d}\Omega - \iint\limits_{\Omega} \Delta v M_c \Delta c \mathrm{d}\Omega. \tag{99}$$

Thus we can write the spatially discretized system as

$$\mathbf{M} \frac{\partial \mathbf{c}}{\partial t} = \mathbf{A}(\mathbf{c}). \tag{100}$$

6.2 Time Integration and Numerical Challenges

The Cahn-Hilliard equation is known to be very stiff [29], and, as we have already seen, it is very nonlinear. These both have dramatic effects on computational costs when time-integrating (100). Stiffness is difficult to define exactly, but it is characterized by explicit time integration methods performing very badly, and is frequent in systems of partial differential equations where different mechanisms act on very different time scales simultaneously[23] [48]. This is the case for the Cahn-Hilliard equation. In a stiff system, stability, rather than accuracy, dictates the length [1] . In practice, this means that one has to utilise implicit time integration methods. One problem with using an implicit method is the computational cost. Generally these methods need to solve rather large systems of equations at each time-step. In this thesis Matlab's built in stiff solver **ode15s** was used to do the time-integration. This is a quite robust and well built solver that selects (and varies) its degree between 1 and 5, and also adjusts its time-step lenght to stay within given accuracy bounds, whilst keeping computational times relatively low. However, a drawback is that it frequently computationally approximates the Jacobian matrix of the right hand side. This means that the operator $\mathbf{A}$ in (100) gets called a large number of times.

The right hand side of the Cahn-Hilliard equation is highly nonlinear. An equation is called nonlinear if the unknown and its partial derivatives are related in a nonlinear manner [49]. Both M_c and μ_c contribute to the nonlinearity of the function. Hence, the right hand side, as we have seen cannot be represented in a static matrix form. In short, the right hand side depends heavily on the concentration c and its derivatives, and therefore it has to be re-calculated every time c is updated (i.e. at every time-step). One ends up with a very large number of calls to the operator $\mathbf{A}$. This has a substantial effect on run-times, even as $\mathbf{A}$ was coded to reduce its computational cost.

In addition, the Cahn-Hilliard equation obviously demands some level of resolution in order to track the boundaries of the phase-interfaces correctly. If the interface boundary is much smaller than the computational resolution it is obvious that the boundaries will not be represented accurately.

6.3 Implementation Adjustments to Solver

The same solver thas was verified earlier was used to implement the Cahn-Hilliard equation. However, some changes had to be made to the solver for it to solve (100). Firstly the mass matrix M had to be calculated. This was performed by a simple integration - very similar to the Poisson problem. The mass matrix is not dependent on c or t and it is therefore sufficient to generate it once, at startup. The same does not hold for the right hand side operator $\mathbf{A}$. It is highly dependent on c and its derivatives, and thus needs to be updated at every timestep. A function called **getRHS(t,c)** was devised taking a given concentration as a column vector input, then giving the right hand side output of a new column vector. As we know,

$$c = \sum_{i=1}^{n_{eq}} c_i N_i, \tag{101}$$

is evaluable at the Gauss points, hence, a reordering of the nested for-loops shown back in figure 17 was deviced. This would ensure that the concentration

c was computed much fewer times per Gauss point. The reordering is shown in algorithm 2. Even though all spline values and mapping values are calculated at startup and not in every loop, the frequent calls to **getRHS(t,c)** to evaluate **A** and its Jacobian have a great impact on run-times. Matlab is also known for not handling nested for-loops ideally, which adds to the inefficiency. Compiling the code in another programming language with better compilation for nested for-loops might well give significantly decreased run-times. Some relevant values of runtimes per evaluation for **getRHS(t,c)** is given in table 5. As one can see, large levels of accuracy are quite time-expensive, if one assume large numbers of calls to the **getRHS(t,c)** function.

```
Data: $t$, $\mathbf{c}$, all neccecccary precalculated spline and mapping values.
Result: The column vector $\mathbf{A}$
Set $\mathbf{N}_0 = 1$
for every element do
    for every nonzero basis function $N_j$ do
        for every Gauss point do
            $c$, $\nabla c$, $\Delta c = 0$;
            for every nonzero basis function $N_i$ do
                $c = c + c_i N_i$;
                $\nabla c = \nabla c + c_i \nabla N_i$;
                $\Delta c = \Delta c + c_i \Delta N_i$;
            end
            Calculate $M_c$ based on the computed $c$;
            Calculate $\nabla \mu_c$ based on the computed $c$ and $\nabla c$;
            Compute the right hand side with the now available variables;
        end
        Perform numerical integration and direct result to $A_j$;
    end
end
```

Algorithm 2: Cahn-Hilliard right hand side algorithm

.

The ode15s solver in Matlab requires the right hand side to be a function of only t and c, but it is neccecary to access all of the spline related variables to compute the right hand side. This was solved by writing combined setter and getter functions that took advantage of Matlab's **persistent** variable types. Persistent variables retain their value in the function, so that the value is the same for all function calls to that function. The combined setter and getter functionality was implemented by checking if the function had input arguments

	$h = 0.5$	$h = 0.25$	$h = 0.125$	$h = 0.0625$	$h = 0.03125$
$p = 3$	0.0132	0.0399	0.1578	0.6335	2.5626
$p = 4$	0.0282	0.1112	0.4465	1.7888	7.1309
$p = 5$	0.0681	0.2872	1.071	4.3075	17.1904
$p = 6$	0.1451	0.5886	2.3018	9.3183	37.1483

Table 5: The runtime in seconds a single call to **getRHS(t,c)**, for different polynomial degrees p and element-sizes h

passed to it. If there were no arguments passed, the persistent value was returned. If arguments were passed to the function, the function would set the persistent value to the input argument. A simple example for getting the number of elements is shown below, however, simliar functions were constructed to be able to access all neccecary spline (and other) variables inside other functions. A complete listing of the main components of the Cahn-Hilliard code, excluding the trivial getter/setter functions, is found in appendix A.

```
function [NoE] = getNoE(valNoE)
persistent nnoe;
%if there is a input argument, update persisent
if nargin==1
    nnoe=valNoE;
end
%return the persistent variable, wether updated or not.
NoE=nnoe;
end
```

6.4 Results

As mentioned earlier, $\theta = 3/2$ corresponds to a physical case. α is generally set equal to 3000 as found in litterature [22]. This is to more easily be able to qualitatively compare results with those of the litterature [13] [22]. As discussed, the Cahn-Hilliard equation does not have any exact solutions, however, the stationary solutions in the periodic case are qualitatively known from the work of [22] and the likes, and in general, the expected behaviour of the equation is qualitatively known. As stated earlier, there are two mechanisms that are modeled in the Cahn-Hilliard equation. They operate on different time scales, and the quickest process is the minimization of the chemical free energy which seperates the solution into its two seperate phases. Thus the first behaviour that is expected to dominate is for the solution to spontaneously seperate into their pure phases very quickly. [22] reports a time figure of around 1×10^{-4} brefore this process is largely completed. Secondly, the minimization of interface free energy is expected to join the pure phases into larger regions as one approaches the steady state solutions. It is noted that the time-scale can vary over many orders of magnitude, as shall be seen.

6.4.1 Dirichlet Boundary Conditions

Before attempting periodic boundary conditions it was decided upon studying the behavior of the solution in the Dirichlet boundary case. These solutions may not have a very direct phsyical application, but it allowed qualitative analysis of the obtained results in a controlled environment. The Dirichlet boundary conditions were opted to be $c = 1$ on the bottom and left boundary, and $c = 0$ on the top and right boundary and $\nabla c \cdot \mathbf{n} = 0$ on all of the boundary. In the interior of the domain, the concentration was set to be random values of c around the constant volume fraction $\bar{c} = 0.5$ such that $c \in [0.45, 0.55]$. This initial condition can be seen in figure 25.

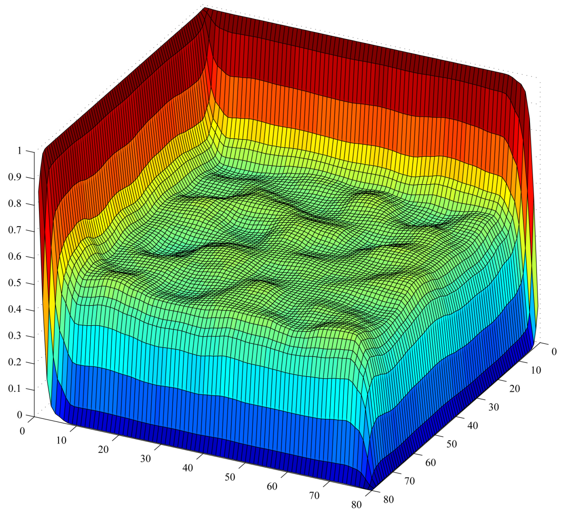


Figure 25: Initial concentration of the Cahn-Hilliard Dirichlet boundary problem

Using these boundary conditions, the qualitative behavior is intuitively expected to be as follows: First, the concentrations in the center should seperate into pure phases ($c = 0$ or $c = 1$), then the two phases should gather along the corresponding boundary. The steady state solution is intuitively expected to be a diagonal line seperating the two faces, stretching between the two corners on which the boundary condition changes. Snapshots of the solver using $p = 3$ and $h = 0.125$ is shown in figure 26. Using resolutions lower than this makes no sence, as the accuracy is too coarse to depict any interfaces in any distinct manner. Obviously $p > 2$ is a must for the continuous basis to be well defined. As

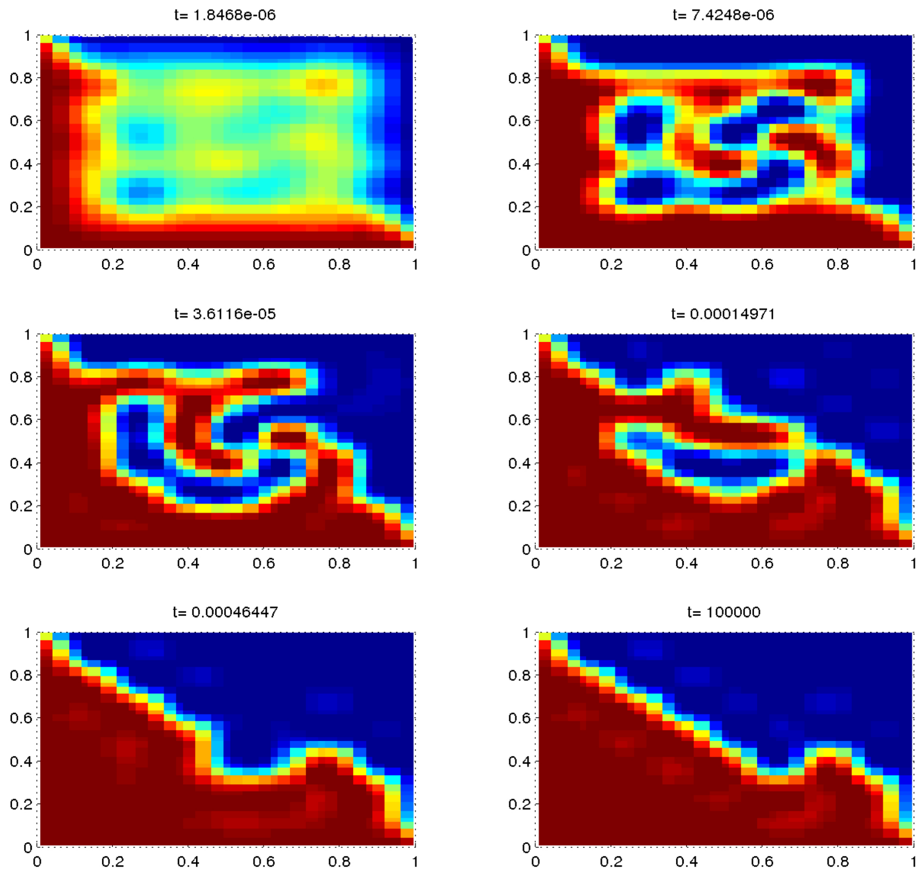


Figure 26: Solving the Cahn-Hilliard equation with $p = 3$ and $h = 0.125$. Time to solve = 12 minutes

we can see, the solution behaves qualitatively as expected. The phase separation dominates the first parts of the solution, and is largely completed at 10^{-5} which is very similar to the values found in litterature, as mentioned above. Then the minimization of interface free energy kicks in and drives the solution towards the steady state solution as expected. The run took about 12 minutes to complete.

The resolution obtained by $p = 3$ and $h = 0.125$ is not overly accurate, so the solution was studied in both the case of increasing p, and halving h, to be able to compare the differences in percieved accuracy and run-times. The solution of $p = 4$ and $h = 0.125$ took 23 minutes to compute, and is shown in figure 27. The results behave as expected, except that the steady state solution was not completely achieved by time $t = 10^5$. Then, instead of increasing p, h - the element size - was halved. Obviously, this increases the computational costs significantly, and the solution took 4.7 hours to complete. The solution using $p = 3$ and $h = 0.0625$ is shown in figure 28.

Finally, the resolution was increased to $p = 4$ and $h = 0.0625$. This run took a whole 11.5 hours to complete, integrating from $t = 0$ to $t = 1 \times 10^5$. The results, shown in figure 29 show good tracking of the boundaries, and the expected behaviour.

The time interval needed to approach stationary solutions is very large, and the step size of the solver must vary over many orders of magnitude. Matlab's ode15s-solver does this quite well. A graph of the log of the time-step size is presented in figure 30. The time-steps varies from 10^{-8} to 10^4 to obtain the stationary solution. This graph of the the time-step size is characteristic and very similar to every other run of the solver. It is noted that the time at which the step-size starts to increase dramatically is when the phase seperation is completed- around $t = 10^{-4}$. Much more integration steps are needed up until this point in time than for the next 10^5 seconds.

Although $\alpha = 3000$ - which is the value used so far - was the most obvious value, we know that the width of the interface boundary is inversely proportional to α. Thus, increasing α should result in thinner, sharper interface boundaries, and decreasing it should result in wider interface boundaries. Firstly a simulation using $p = 3$, $h = 0.125$ and $\alpha = 1500$ was completed. This should create wider and more diffuse boundaries. This vas verified, as can clearly be seeen in figure

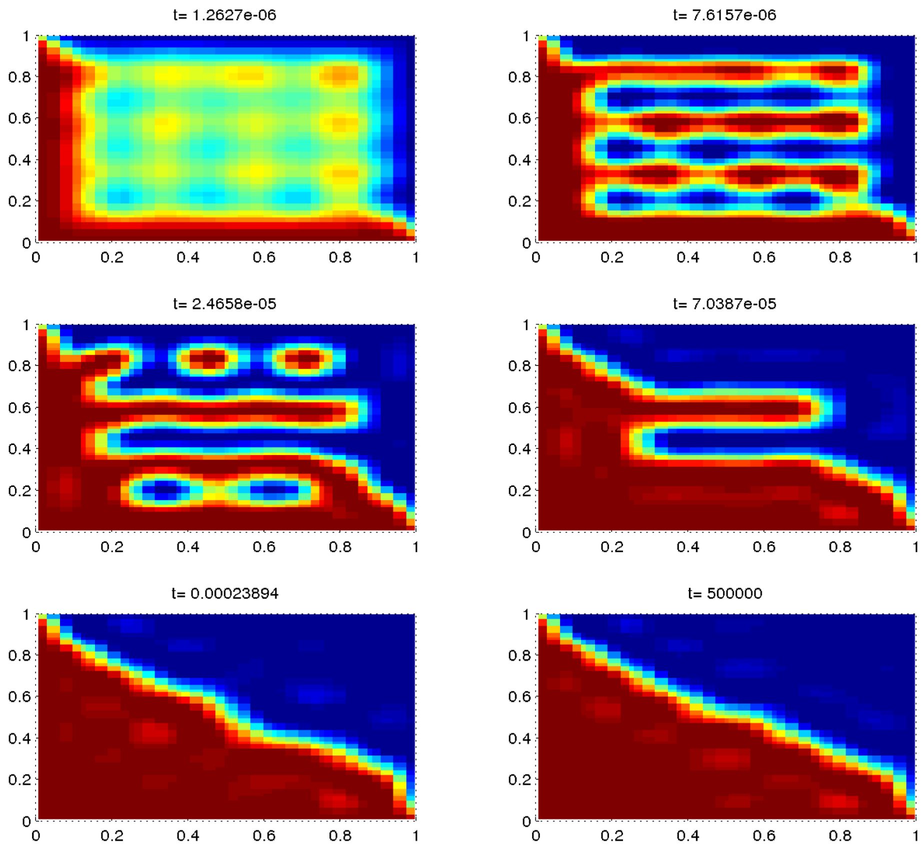


Figure 27: Solving the Cahn-Hilliard equation with $p = 4$ and $h = 0.125$. Time to solve = 23 minutes

31. It is especially visible in the two first time-steps, and when compared to the solutions above.

Later, a new simulation was done at the same resolution, but with $\alpha = 5000$, which should decrease the boundary thickness. A noteworthy increase in the time used by the solver was foun - nearly double of the $\alpha = 1500$. This is because the solver needs to take smaller time-steps to accurately track and maintain stability with such sharp interface boundaries. The results are shown in figure 32. However, the sharper boundaries are not clearly visible. This can be explained by

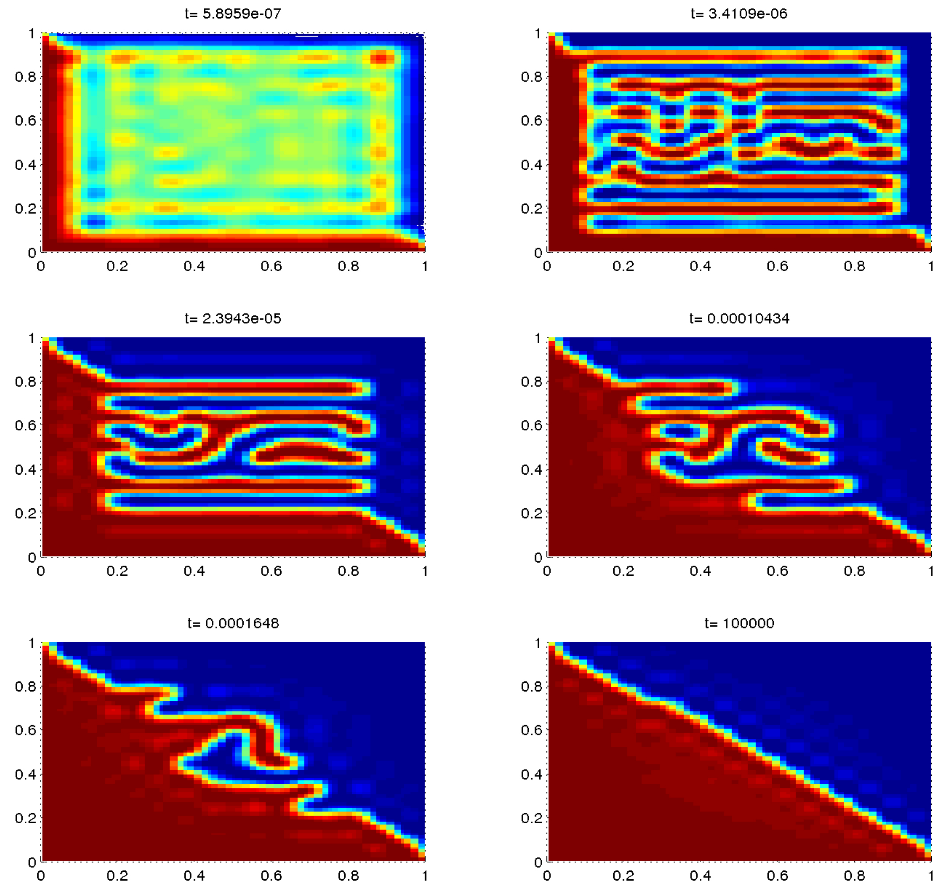


Figure 28: Solving the Cahn-Hilliard equation with $p = 3$ and $h = 0.0625$. Time to solve = 4.7 hours

the rather low resolution that was used for the run. The resolution is too low to trace the boundary exactly. Increasing the resolution is expected to better reveal the difference in interface width for different values of α. However, due to long run times, this was not prioritized.

A final example that was tested was the value of θ. Any $\theta < 1$ should result in a uniform distribution of $c = \bar{c} = 0.5$, very different to the solutions already seen. Thus, θ was set equal to $1/2$ to test the behaviour of the solver. The Dirchlet

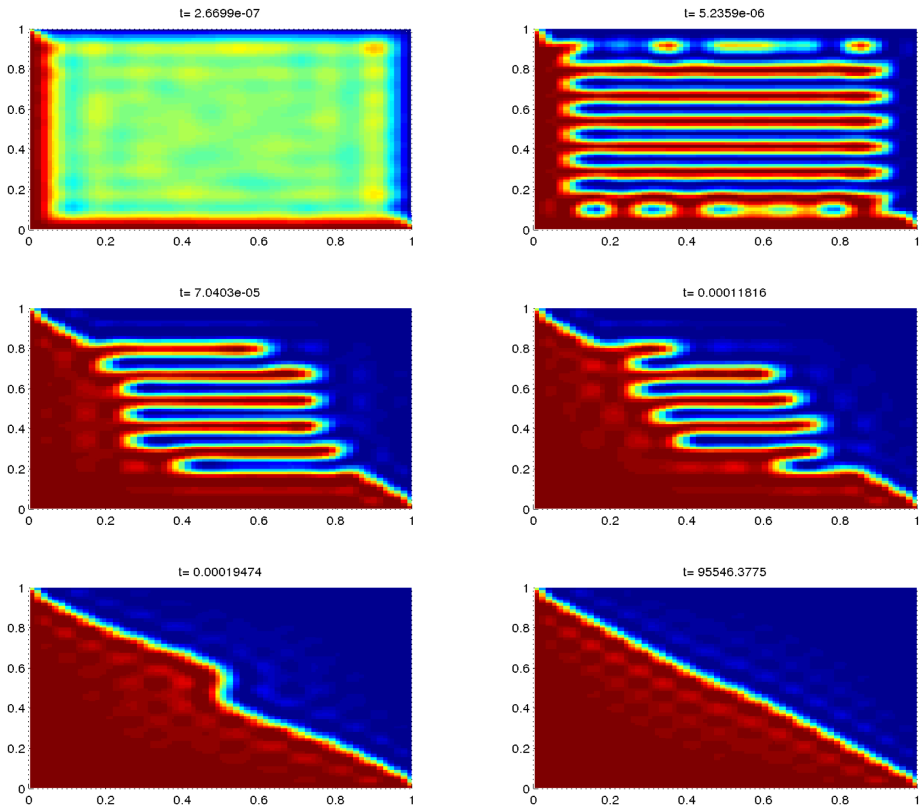


Figure 29: Solving the Cahn-Hilliard equation with $p = 4$ and $h = 0.0625$. Time to solve =11.5 hours

boundary conditions were set to the known value of $c = 0.5$, whereas the initial concentration was changed to have larger extremeties, $c \in [0.3, 0.7]$, to observe the solution more easily. The results were exactly as expected, as shown in figure 33. These tests performed with Dirichlet boundary conditions qualitatively verifies the behavior of the implemented Matlab solver nicely.

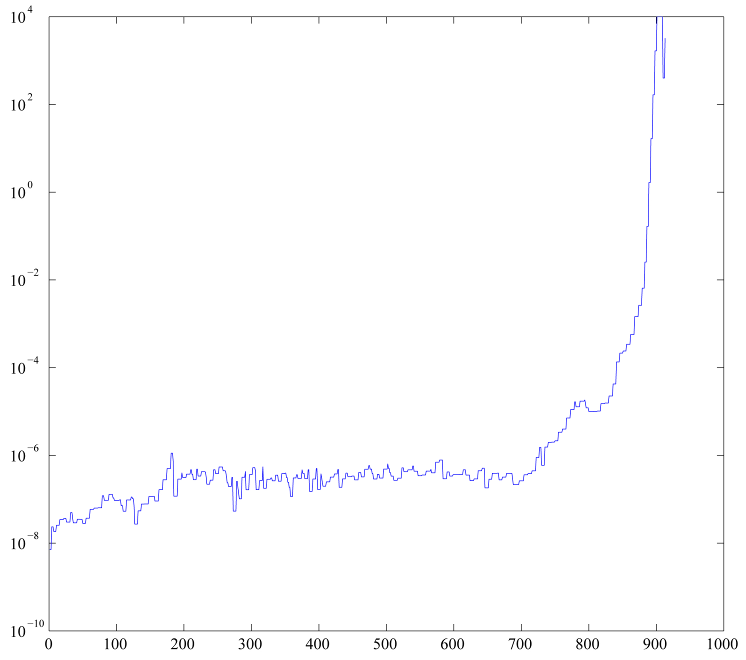
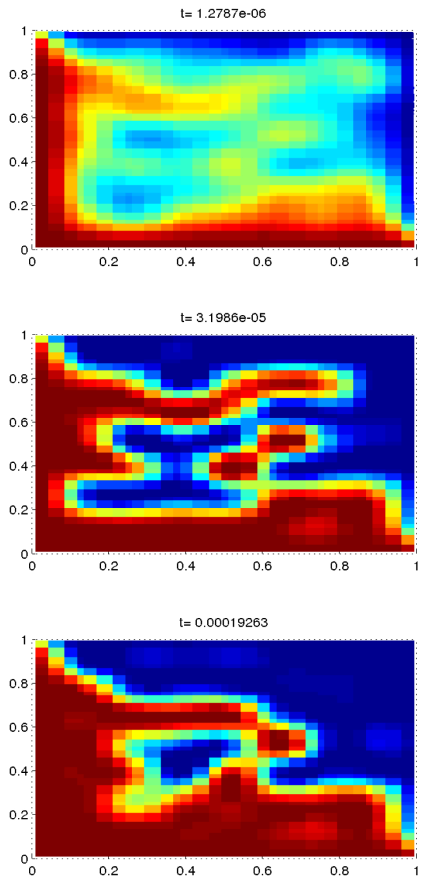


Figure 30: The logarithm of the time per time-step showing the adaptive stepsize of ode15s



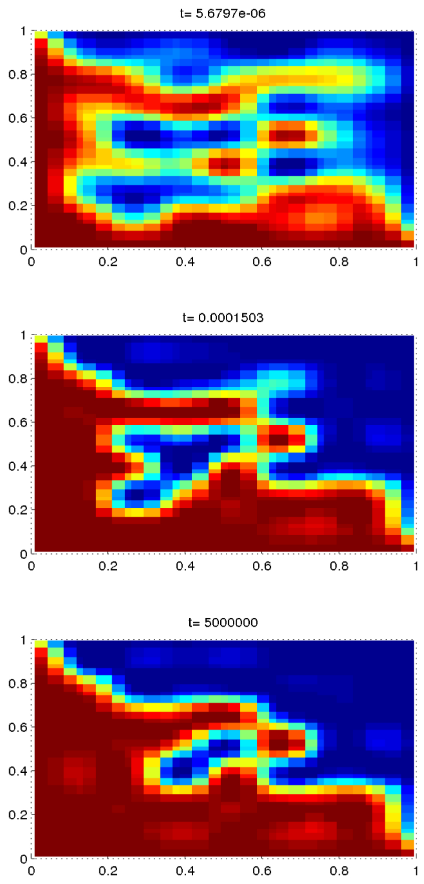


Figure 31: Solving the Cahn-Hilliard equation with $\alpha = 1500$, $p = 3$ and $h = 0.125$. More diffuse interfaces visible.

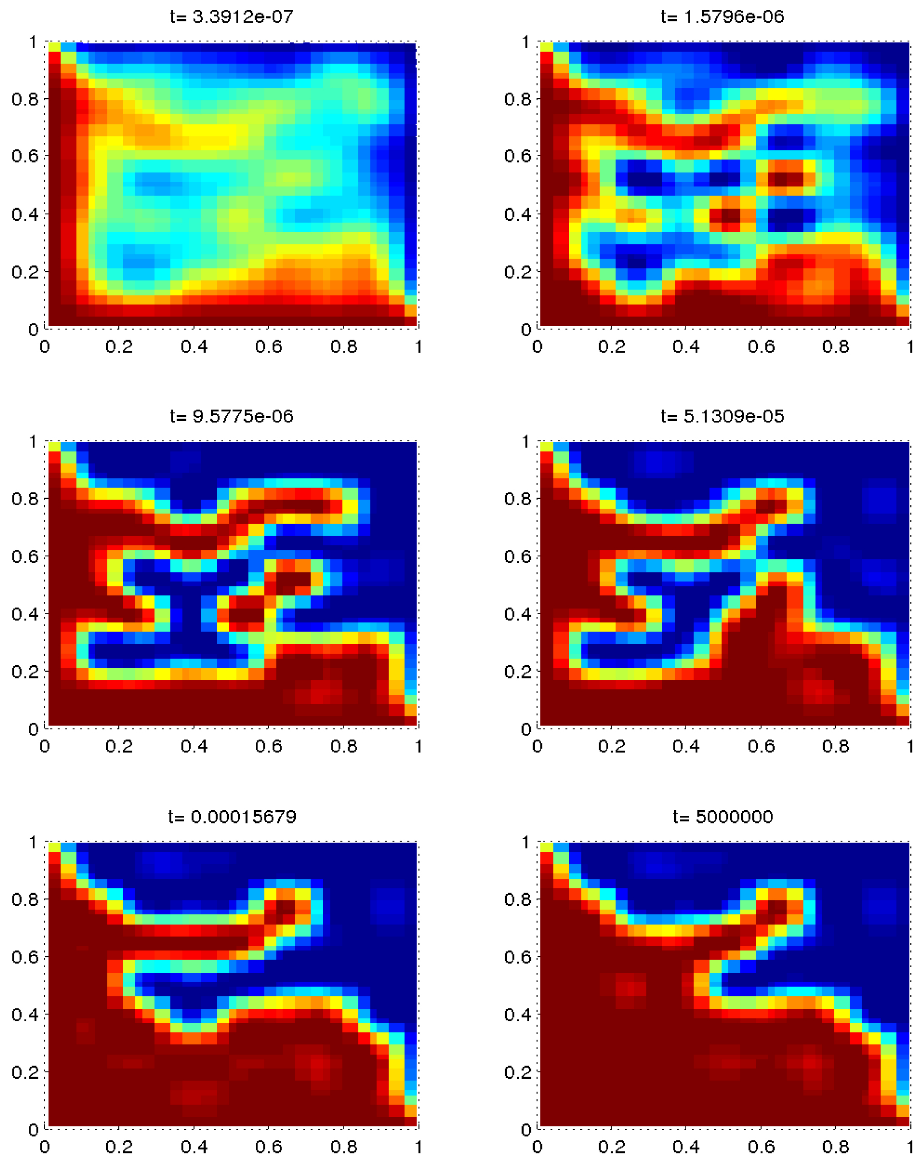
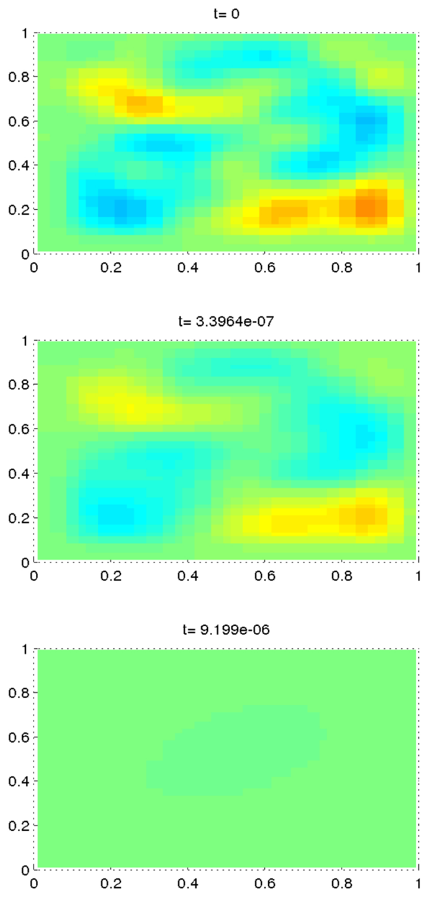


Figure 32: Solving the Cahn-Hilliard equation with $\alpha = 5000$, $p = 3$ and $h = 0.125$.



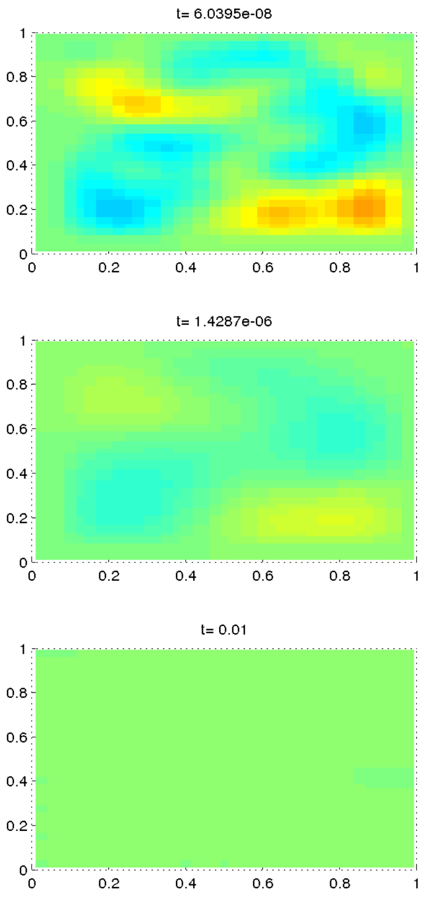


Figure 33: Solving the Cahn-Hilliard equation with $\theta = 0.5$, $p = 3$ and $h = 0.125$. Time to solve =1 minute

6.4.2 Periodic Boundary Conditions

The solver was also implemented using Periodic Boundary Conditions, as introduced in section 3.5.3. This involved choosing master and slave degrees of freedom as shown earlier in figure 12, and setting up the transformation matrix **T**. Most of this is rather straight foward. However, the corners, especially the top right corner, require quite a bit of back-tracking until the slave DOFs are expressed as functions of only master DOFs. The corners needed seperate treatment in the creation of the transformation matrix. However, once completed, the function creating **T** worked for all sizes of rectangular domains - as is always the case in the parameter space in Isogeometric Analysis. As (100) only governs the change over time, it is important that the initial concentration c_0 was periodic. This was accomplished by firstly by setting all of the values around the slave boundary equal to the corresponding master boundary. Then the first internal points were set to match the derivatives of the slave boundary to each corresponding derivative of the master boundary. The initial non-periodic conditions can be seen in figure 34, whereas the initial conditions after it was made periodic is visible in

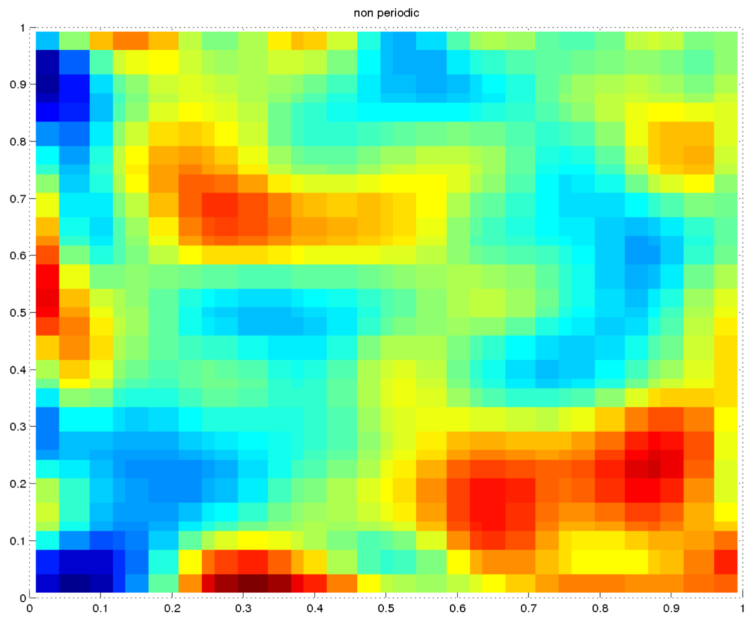


Figure 34: Non-periodic initial conditions for periodic boundary runs

figure 35.

Several runs were performed using this implementation of periodic boundary

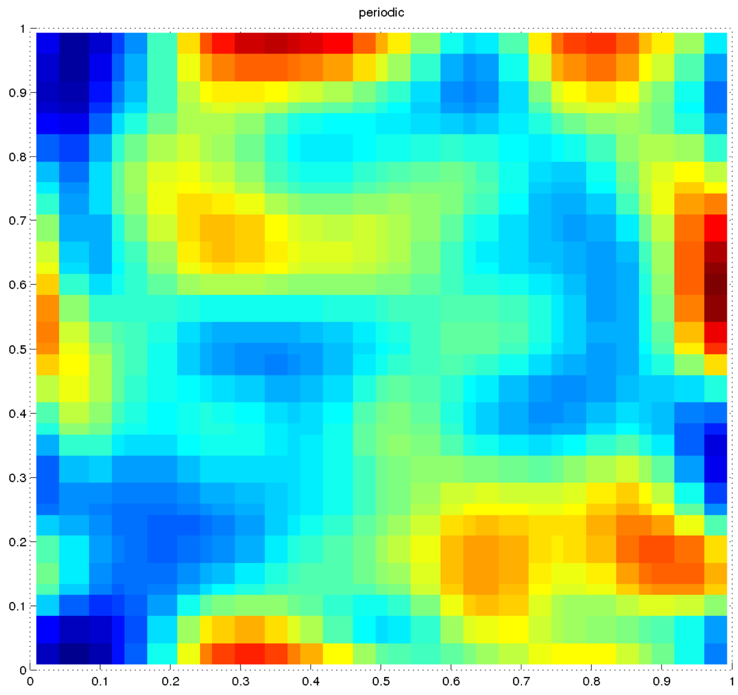


Figure 35: Periodic initial conditions for periodic boundary runs

conditions. One run is shown in figure 36 with $p = 3$ and $h = 0.0625$. It is clear form these results that something is not quite right. The solution does not go to its pure phases completely. In addition an erronous area in the middle of the solution is seen. However, it is easily seen that the solution seems periodical around the edges. To further investigate this behaviour, the same solution was studied without performing the reinsertion of slave variables - thus looking only at the reduced system. These results are shown in figure 37. Without the reinsertion of the slave DOFs, these plots will make little sence around the slave boundary. However, in the reduced system, the qualitatively expected behaviour of the Cahn-Hilliard equation is still apparant, at least for these early time steps. Another set of reduced-system results are shown in figure 38, still showing qualitatively promising results on the interior of the boundary, and periodic behaviour. The error then, seems to be located in the application of periodic boundaries - not in the inner workings of the solver.

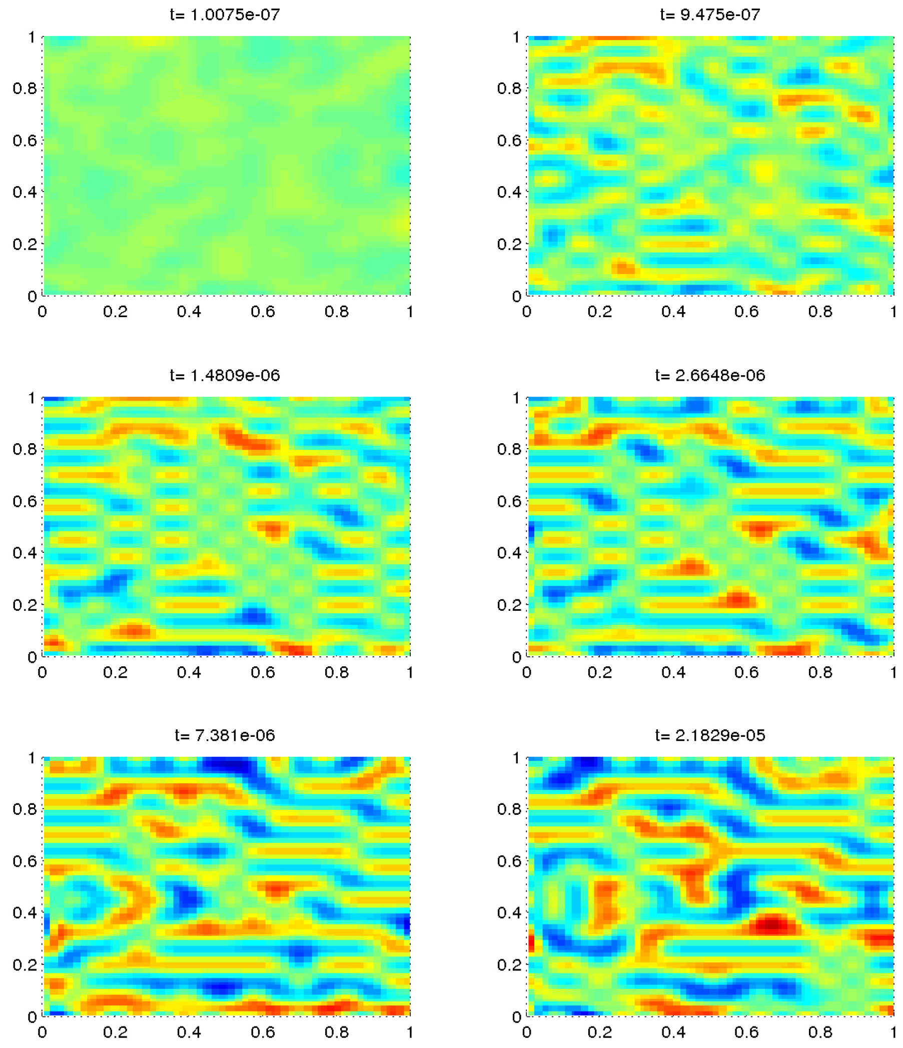


Figure 36: Run with periodic boundary conditions implemented

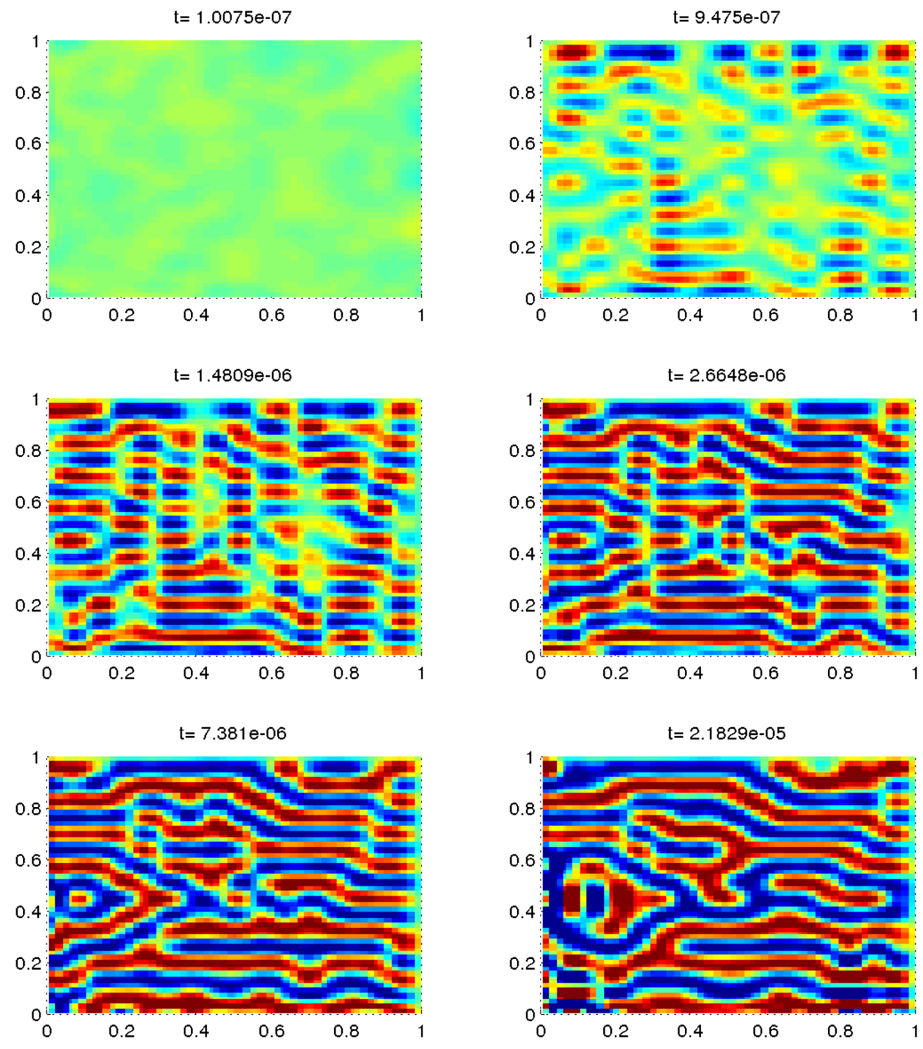


Figure 37: Run with periodic boundary conditions implemented, without slave DOF reinsertion. $p = 3$, $h = 0.0625$

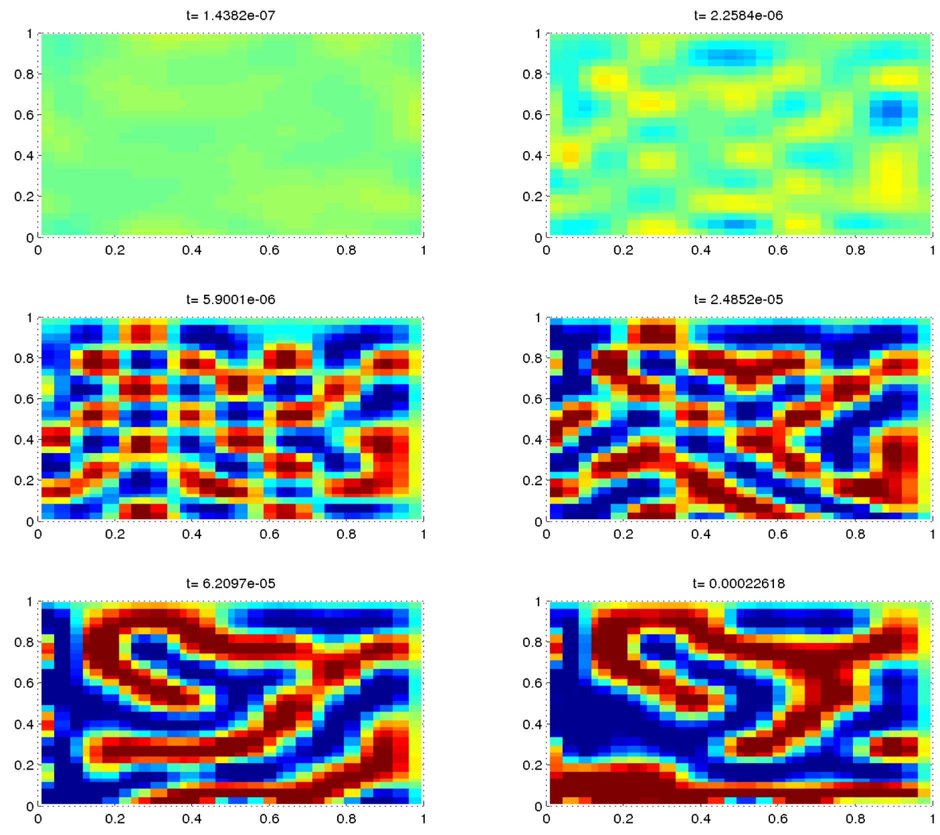


Figure 38: Run with periodic boundary conditions implemented, without slave DOF reinsertion. $p = 3$, $h = 0.125$

6.5 Discussion

The implementation of Dirichlet boundary conditions revealed very good results. The solver modeled several known features of the Cahn-Hilliard equation in a pleasing manner, verifying the solver nicely. However, during the implementation of periodic boundary conditions, some difficulties were encountered. There are two areas that are deemed most likely to be the origin of the error. Firstly, setting up the transformation matrix was quite tidious work, especially finding the relations around the corner areas. A mistake could have been done in creating the code used to set up this large system of equations. However, the work was double checked, and verified for a simple, small domain. Hence, the problem most likely lies in the implementation of the transformation matrix. The implementation is straight foward in the standard system

$$\mathbf{K}\mathbf{u} = \mathbf{f}. \tag{102}$$

Our system is obviously quite different,

$$\mathbf{M}\frac{\partial \mathbf{c}}{\partial t} = \mathbf{A}(\mathbf{c}). \tag{103}$$

The implementation on the left hand side is similar in both cases. It is, however, slightly more difficult to implement the periodic boundary conditions on the non-linear right hand side. Thus it is hypothesised that the error in implementing periodic boundary conditions is to be found on the right hand side. Had the decision to implement periodic boundary conditions been made earlier than it did, time would have most likely been left to single out and fix the error. Sadly, with computational times getting large as Dirichlet boundary conditions were implemented, not enough time was left to discover and correct this error. However, this leads us to the next point of discussion: Computational Times.

It is obvious that a large limitation to the current implementation is the run-times. The large cost of computing $\mathbf{A}(\mathbf{c}$ meant that only combinations of $p = 2, 3$ and $h = 0.125, 0.0625$ were practicable for the above simulations. Cahn-Hilliard is known to be computationally costly, but there are two areas in which solution times ccould be minimized. The computational cost is completely dominated by the solving of the nonlinear right hand side of the Cahn- Hilliard equation. Hence, one can either attempt to reduce the cost of computing the right hand side, or reduce the amount of times one has to compute the right hand side. Reducing the cost of calculating the right hand side is probably possible. The right hand side was indeed coded to reduce run-times, but it is possible that this can be taken further. Three things that could be attempted are as follows: All of the spline basis functions' values are pre-calculated. It is thus probably possible to pre-calculate a larger number neccceccary variables, reducing some right hand side calculations to table lookups, if memory is not a limitation. The second option would be to attempt to simplify the right hand side. This would require a thorough mathematical approach, but it is known that μ_c, for example, often is approximated by polynomial functions. Such simplifications can reduce computational costs somewhat. A third option is to rewrite and compile the code in a different, and preferabely lower level programming language. Knowing that Matlab is less than ideal for multiple nested for-loops, a compilation with speedier handling of for-loops could increase the speed of calculating the right hand side significantly.

The other major option is to reduce the number of calls to the right hand side. As noted earlier Matlab's ode15s solver was utilized for this thesis. Ode15s is a quite robust and well coded solver, however, it is also very general - created to be able to handle most types of equations Matlab users will throw at it. Thus, other, more custimizable methods, reducing the amount of time-steps needed would reduce the number of calls to the right hand side. [22] and [13] both identify this need of a good solver with time-step adaptivity, and suggest implementing the generalized-α method [9] to overcome time-stepping difficulties. This could further reduce the computational costs.

7 Conclusion

B-splines turn out to be very simple yet powerful functions for geometric representation. The most common geometric representation in CAD is based on B-splines, it was through this project verified that B-splines are a valid and good choice as basis functions in the Galerkin method on which FEA is based. Using B-splines as basis functions allows for, at least potentially, the use of CAD models directly, and exact geometries at all levels of refinement, which is known to be important in many applications. The simple structure, and local support of the basis functions and their derivatives makes this implementation smooth and efficient. Using B-splines as basis functions is both verified and reccomended.

The Isogeometric Analysis solver implementation in Matlab was firstly tested and verified on the Poisson equation with homogeneous dirichlet boundary conditions. The error analysis was completed by performind several runs with different polynomial degree p and element size h. The error analysis conformed to the theoretical error prediction, and showed very strong results in favor of Isogeometric Analysis. The error convergence is the same as in FEA, but increases in mesh detail come at a lower computational cost per degree of freedom than in FEA. The solver was then used to solve the higher order Biharmonic equation on the unit square. The same, strong error results held true also for this higher order partial differential equation. Finally, the solver was implemented on a much more complex equation; the non-linear and stiff Cahn-Hilliard-equation. Modifications to the solver, and numerical and computational challenges in solving this non-linear and stiff system were discussed. In the Dirichlet boundary case, the results were qualitatively verified, not only producing early run-time results, but achieving the steady state solution. Achieving the steady solution can be difficult. Periodic boundary conditions were applied, which was a partial success. The solution behaved largely as expected, as a periodic solution, except at the slave boundary. Amongst the issues was that reinserting slave degrees of freedoms did not produce the expected periodic results. This reveals that implementing periodic boundary conditions is slightly more intricate in the mass-matrix-based system of partial differential equations with a nonlinear operator on the right hand side than in a normal stiffness-matrix implementation.

Isogeometric Analysis is clearly a great numerical tool. Some of its strenghts are demonstrated through this thesis, and the andvantages of this type of analysis are clear. Not only will Isogeometric Analysis allow the usage of exact models at all levels of refinement, giving more exact results, it will even give these results at asymptotically lower computational cost. Furthermore, whereas traditional FEM requires complex workarounds to solve higher order equations - and no general method exists - Isogeometric Analysis is able to solve higher order differential equations with almost no adjustments to the method or solver. This is a very noteworthy advantage, and can become very important in fields where higher order partial differential equations frequently appear. In addition, Isogeometric Analysis seems to be a promising and robust solution to bridge the gap that exist between the FEA and CAD technologies. The potential economic savings in removing the need of seperate mesh generation for FEA is enormous. The industry might still be largely sticking to tratditional FEA, but Isogeometric Analysis definitely has the potential to play a very central part of analysis in the years to come, as it grows into a more mature field, potentially outperforming and replacing FEA in many applications.

References

[1] Hala Ashi. *Numerical Methods for Stiff Systems.* PhD thesis, The university of Nottingham, 2008.

[2] John W Barrett, James F Blowey, and Harald Garcke. Finite element approximation of the cahn–hilliard equation with degenerate mobility. *SIAM Journal on Numerical Analysis*, 37(1):286–318, 1999.

[3] Y. Bazilevs, V. M. Calo, Y. Zhang, T. J. R. Hughes, and G. Sangalli. Isogeometric flyid-structure interaction analysis with applications to arterial blood flow. *Computational Mechanics*, (38), 2006.

[4] Y. Bazilevs, L. Beirao de Veiga, J. A. Cottrell, T. J. R. Hughes, and G. Sangalli. Isogeometric analysis: approximation, stability and error estimates for h-refinement meshes. *Mathematical Models and Methods in Applied Sciences*, (16), 2006.

[5] C. De Boor. On calculation with b-splines. *Journal of Approximation Theory*, (6), 1972.

[6] John W. Cahn. On spinodal decomposition. *Acta Metallurgica*, 9(9), 1961.

[7] John W. Cahn and John E. Hilliard. Free energy of a nonuniform system. i. interfacial free energy. *The Journal of Chemical Physics*, 28(2), 1958.

[8] E.B. Christoffel. Ueber die gaussche quadratur und eine verallgemeinerung derselbe. *J. Reine Angew. Math.*, (55):61–82, 1858.

[9] Jintai Chung and Gregory M Hulbert. A family of single-step houbolt time integration algorithms for structural dynamics. *Computer methods in applied mechanics and engineering*, 118(1):1–11, 1994.

[10] E. Cohen, R. F. Reisenfeld, and F. Elber. *Geometric Modeling with Splines: An Introduction.* A. K. Peters Ltd, 2001.

[11] J. A. Cottrell, T. J. R. Hughes, and A. Reali. Isogeometric analysis of structural vibrations. *Computer Methods in Applied Mechanics and Engineering*, (196), 2007.

[12] J. A. Cottrell, A. Reali, Y. Bazilevs, and T. J. R. Hughes. Isogeometric analysis of structural vibrations. *Computer Methods in Applied Mechanics and Engineering*, (195), 2006.

[13] J. Austin Cottrell, Thomas J. R. Hughes, and Yuri Bazilevs. *Isogeometric Analysis.* Wiley, 2009.

[14] Charles M Elliott and Donald A French. Numerical studies of the cahn-hilliard equation for phase separation. *IMA Journal of Applied Mathematics*, 38(2):97–128, 1987.

[15] G Engel, K Garikipati, TJR Hughes, MG Larson, L Mazzei, and RL Taylor. Continuous/discontinuous finite element approximations of fourth-order elliptic problems in structural and continuum mechanics with applications to thin beams and plates, and strain gradient elasticity. *Computer Methods in Applied Mechanics and Engineering*, 191(34):3669–3750, 2002.

[16] Lawrence C. Evans. *Partial Differential Equations.* American Mathematical Society Providence, Rhode Island, second edition, 2010. Graduate Studies in Mathematics Volume 19.

[17] G. Farin. *Curves and Surfaces for CAGD, A Practical Guide.* Morgan Kaufmann Publishers, fifth edition, 1999.

[18] Carlos A. Felippa. Introduction to finite element methods howpublished = `http://www.colorado.edu/engineering/cas/courses.d/ifem.d/ifem.ch08.d/ifem.ch08.pdf`, note = Lecture notes from Department of Aerospace Engineering Sciences and Center for Aerospace STructures, University of Colorado, year = 2004,.

[19] Ellen Kuhl Garth N. Wells and Krishna Garikipati. A discontinuous galerkin method for the cahn-hilliard equation. *Journal of Computational Physics*, (218), 2006.

[20] H. Gomez, V. M. Calo, Y. Bazilevs, and T. J. R. Hughes. Isogeometric analysis of the cahn-hilliard phase-field model. *Computer Methods in Applied Mechanics and Engineering*, (197), 2008.

[21] R. T. Haftka. Automated procedure for design of wing structures to satisfy strength and flutter requirements. Technical report, NASA - Langley Research Center, 1973. Document ID: 19730018200.

[22] Yuri Bazilevs Hector Gomez, Victor M. Calo and Thomas J.R. Hughes. Isogeometric analysis of the cahn-hilliard phase-field model. Technical report, Institute for Computational Engineering and Sciences, The university of Texas at Austin., December 2007.

[23] Desmond J Higham and Lloyd N Trefethen. Stiffness of odes. *BIT Numerical Mathematics*, 33(2):285–303, 1993.

[24] T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis.* Dover Publications, 2000.

[25] T. J. R. Hughes, J. A. Cottrell, and Y. Bazilevs. Isogeometric analysis: Cad, finite elements, nurbs, exact geometry and mesh refinement. *Computer methods in applied mechanics and engineering*, (194), 2005.

[26] T.J.R. Hughes, A. Reali, and G. Sangalli. Efficient quadrature forNURBS-based isogeometric analysis. *Computer Methods in Applied Mechanics and Engineering*, (199), 2010.

[27] Bernie L. Hulme. Discrete galerkin and related one-step methods for ordinary differential equations. *Mathematics of Computation*, 26(120):881–891, 1972.

[28] J. Blowey J. Barrett and H. Garcke. Finite element approximation of the cahn-hilliard equation with degenerate mobility. *SIAM Journal on Numerical Analysis*, 37(1):286–318, 1999.

[29] J. A. Evans M. J. Borden J. Liu, L. Dede and T. J. R. Hughes. Isogeometric analysis of the advective cahn-hilliard equation: Spinodal decomposition under shear flow. Technical report, The Institute for Computational Engineering and Sciences, The University of Texas at Austin, March 2012. ICES REPORT 12-12.

[30] Claes Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method.* Dover Publication, 2009.

[31] Erwin Kreyszig. *Advanced Engineering Mathematics.* Wiley, eigth edition, 1999.

[32] V. D. Liseikin. *Grid Generation Methods.* Springer, 2010.

[33] Daryl L. Logan. *A First Cource in the Finite Element Method.* Cengage Learning, fourth edition, 2007.

[34] Tom Lyche and Knut MÃ¸rken. *Spline Methods Draft.* Department of Informatics, Centre of Mathematics for Applications, University of Oslo, 2011.

[35] Robert C. McOwen. *Partial Differentioal Equations, Methods and Applications.* Pearson Education, second edition, 2003.

[36] David Moratal, editor. *Finite Element Analysis - New Trends and Developments.* InTech, 2012. under CC BY 3.0 license DOI:10.5772/2552.

[37] Bart Blanpain Nele Moelans and Patrick Wollants. An introduction to phase-field modeling of microstructure evolution. *calphad*, 32(2):268–294, 2008.

[38] Brynjulf Owren. *TMA4212 Numerisk lÃ¸sning av partielle differensial-ligninger med endelig differensmetoder.* IME institute, NTNU, 2007. In norwegian only. Curriculum used in class TMA4212.

[39] L. Piegl and W. Tiller. *The NURBS Book (Monographs in Visual Communication).* Springer Verlag, second edition, 1997.

[40] G. Prathap. *The finite element method in structural engineering: Principles and practice of design of field-consistent elements for structural and solid mechanics*, volume 24. Kluwer Academic, 1993.

[41] E. M. Purcell. Life at low reynolds number. *American Journal of Physics*, 45(1), January 1977.

[42] J.N. Reddy. *An introduction to the finite element method.* McGraw-Hill science, third edition, 2005.

[43] Richard Franklin Riesenfeld. *Applications of b-spline approximation to geometric problems of computer-aided design.* PhD thesis, Syracuse University, Syracuse, NY, USA, 1973.

[44] Yousef Saad. *Iterative Methods for Sparce Linear Systems.* Society for Industrial and Applied Mathematics, second edition, 2003.

[45] E. Schelkle and R. Remensperger. Integrated occupand-car crash simulation with the finite element method: The porsche hybrid iii-dummy and airbag model. Technical report, SAE Techincal Paper 910654, 1991. doi:10.4271-910654.

[46] A. P. S. Selvadurai. *Partial Differential Equations in Mechanics 2: The Biharmonic Equation, Poisson Equation.* Springer, 2000.

[47] P. Seshu. *Textbook of Finite Element Analysis.* PHI Learning Pvt. Ltd., 2004.

[48] John M. Stockie and Brian R. Wetton. Analysis of stiffness in the immersed boundary method and implications for time-stepping schemes. *J. Comput. Phys*, 154:41–64, 1998.

[49] Eitan Tadmor. A review of numerical methods for nonlinear partial differential equations. *Bulletin of the American Mathematical Society*, 49(4):507–554, 2012.

[50] Eiric W. Weisstein. Fundamental theorem of gaussian quadrature. `http://mathworld.wolfram.com/FundamentalTheoremofGaussianQuadrature.html`. From MathWorld–A Wolfram Web Resource.Accessed 28 November 2012.

[51] Garth N Wells, Ellen Kuhl, and Krishna Garikipati. A discontinuous galerkin method for the cahn–hilliard equation. *Journal of Computational Physics*, 218(2):860–877, 2006.

[52] Yinhua Xia, Yan Xu, and Chi-Wang Shu. Local discontinuous galerkin methods for the cahn–hilliard type equations. *Journal of Computational Physics*, 227(1):472–491, 2007.

[53] Y. Zhang, Y. Bazilevs, S. Goswami, C. Bajaj, and T. J. R. Hughes. Patient-specific vascular nurbs modelling for isogeometric analysis of blood flow. *Computer Methods in Applied Mechanics and Engineering*, (196), 2007.

[54] J.Z. Zhu and O. C. Zienkiewicz. Adaptive techniques in the finite element method. 4:197–204, 1988.

A Cahn-Hilliard Code Implementation

The main run sequence:

```
%Cahn—Hilliard solver
%Spline variables and calculation————————————————————
%setting p—values
[p1,p2]=getP(3,3);
%setting element size values (e.size = 1/(h+1))
h1=3;
```

```
h2=3;
%calculate spline variables
getSplineVariables(p1,p2,h1,h2);

%Setting Cahn–Hilliard constants
%alpha=3000;
%theta = 3/2;
getCHconstants(3000,(1/2));

%Initial conditions
c0=getInitialConcentration(getSize());
%make random initial concentration periodic
C0=applyPeriodic(c0);

%Calculate Mass matrix
M = getMassMatrix();

%Apply boundary data to Mass matrix
T=getTransformationBoundary(1);
MB=(T.')*M*T;
for i = 1:length(MB(:,1))
   if MB(i,:)==zeros
      MB(i,i)=1;
   end
end

%solve using Mass matrix and Matlab's ode15s solver
%start time
ts=0;
%end time
te=100
%Solving using Matlab ode15s:
options=odeset('Mass',MB);
[T,Y] = ode15s(@getBandCterms,[ts, te],C0,options);

%Reinsert known values
Yactual=performReinsertion(Y);
```

getSplineVariables:

```
function [] = getSplineVariables(p1,p2,h1,h2)
%GETSPLINEVARIABLES ————————————————————————————
%   Inputs:
%   p1,p2 = polynomial degree in each direction
%   h1,h2 = element division. 2^h1 divisions.
%
%   Sets to setter/getter–functions for later access from anywhere:
%   lengthT,lengthU = number of spline functions
%   NoE = number of elements
%   GToL = global to local
%   SinE = which functions have support in element E
%   weightXi1, weightXi2 = gauss weights.
%   xi1,xi2 = gauss points on elements.
%   PinEXi1,PinEXi2 = index of first point in element E.
%   EndpointsofE = physical coordinates of element E, for
%   area calculation.
%   N1,N1D,N1DD,N2,N2D,N2DD = spline and spline deriv.values
%   in gausspts.

%get knotvectors
tStar=getKnotvector(h1);
uStar=getKnotvector(h2);

%making them both p1 and p2 regular
t=[(tStar(1)*ones(p1,1));tStar;tStar(length(tStar))*ones(p1,1)];
u=[(uStar(1)*ones(p2,1));uStar;uStar(length(uStar))*ones(p2,1)];

%Number of spline functions in each direction:
lengthT=length(t)-p1-1;
lengthU=length(u)-p2-1;
%setting problem size for later access
getSize(lengthT,lengthU);
%setting Number of elements for later access
NoE = (lengthT-p1)*(lengthU-p2);
getNoE(NoE);


%Getting connectivity, GlobalToLocal function mapping = GToL
%and which functions (globally) have Support in Element e = SinE
[GToL, SinE]=getConnectivity(lengthT, lengthU, p1,p2);
%setting variables for later access.
getConnect(SinE,GToL);

%finding boundary splines for periodic boundary implementation
[bSbottomtop,bSleftright]=getBoundarySplines(GToL,lengthT,lengthU);
[bbSbottomtop,bbSleftright]=getInternalBoundarySplines(GToL,lengthT,
lengthU);
%setting boundary splines for later access.
getPeriodicBoundary(bSbottomtop,bSleftright,bbSbottomtop,
bbSleftright,bCorners,bbCorners);


%Getting gausspoints xi1 and xi2, weights, and PinEXi1/2 - which
%has the index of the first point in each direction in each element.
[weightXi1, weightXi2, xi1, xi2, PinEXi1, PinEXi2, EndpointsofE] =
getGaussPoints(unique(t), unique(u), lengthT-p1, lengthU-p2,p1, p2);

%setting variables for later access
getWeights(weightXi1,weightXi2);
getXi(xi1,xi2);
getPinE(PinEXi1,PinEXi2);
getEndpoints(EndpointsofE);

```

```
%Finding values and derivatives of all bsplines at each gausspoint:
[N1,N1D,N1DD]=getSplineValsAnd2Derivs(p1,t,xi1);
[N2,N2D,N2DD]=getSplineValsAnd2Derivs(p2,u,xi2);
%setting variables for later access.
getSplineValues(N1,N1D,N1DD,N2,N2D,N2DD);

end
```

getKnotVector:

```
function [ knotvector ] = getKnotvector(divs)
%GETKNOTVECTOR ----------------------
%   Returns knot vector with divs amounts of divisions.
knotvector = [0 1]';

for i =1:divs
    j=1;
    while j ≤ length(knotvector)-1
        knotvector=[knotvector(1:j) ; ...
             (knotvector(j)+knotvector(j+1))/2; ...
             knotvector(j+1:length(knotvector))];
        j=j+2;
    end
end
end
```

getConnectivity:

```
function [INN,IEN] = getConnectivity(lengthT, lengthU, p1, p2)
%GETCONNECTIVITY -----------------
% Returns mapping INN (Global to local mapping) and
% IEN - which functions (global) have support in element e.
% Based on T. Hughes et. al's approach
% Global variable definitions and initializations:
nel = (lengthT-p1)*(lengthU-p2); % number of elements
nnp = lengthT*lengthU; % number of global basis functions
nen = (p1+1)*(p2+1); % number of local basis functions
INN = zeros(nnp,2);% NURBS coordinates array
IEN = zeros(nen, nel); % connectivity array
% Local variable initializations:
e=0;
A=0;
B=0;
b=0;
iloc=0;
jloc=0;
for j=1:lengthU
    for i=1:lengthT
        A = A + 1; % increment global function number
        INN(A,1) = i;
        INN(A,2) = j; % assign spline coordinates
        if i≥(p1+1) && j≥(p2+1)
            e = e+1; % increment element number
            for jloc=0:p2
                for iloc=0:p1
            % global function number
                    B = A − jloc*lengthT − iloc;
            % local function number
                    b = jloc*(p1+1)+ iloc + 1;
                    IEN(b,e) = B; % assign connectivity
                end
            end
        end
    end
end
end
```

getBoundarySplines:

```
function [bSbottomtop, bSleftright] = getBoundarySplines( ...
    GToL,lengthT, lengthU )
%GETBOUNDARYSPLINES ———————————————————
%
%returns (master,slave) global function numbers
%for periodic boundary conditions in each direction
%i.e. bottom−top and left−right.


bSbottomtop = zeros((2*lengthT+2*lengthU−4)/4,2);
bSleftright = zeros((2*lengthT+2*lengthU−4)/4,2);

indexBb=1;
indexBl=1;
indexBt=1;
indexBr=1;
%loops through each global function and detects which ones
%are on the edge.
for i=1:length(GToL(:,1))
    if GToL(i,2) == 1
        bSbottomtop(indexBb,1)=i;
        indexBb=indexBb+1;
    end
    if GToL(i,1) == 1
            bSleftright(indexBl,1)=i;
            indexBl=indexBl+1;
    end
    if GToL(i,2) == lengthU
            bSbottomtop(indexBt,2)=i;
            indexBt=indexBt+1;

    end

    if GToL(i,1) == lengthT
            bSleftright(indexBr,2)=i;
            indexBr=indexBr+1;
    end
end
end
```

getInternalBoundarySplines:

```
function [bbSbottomtop, bbSleftright] = ...
    getInternalBoundarySplines( GToL,lengthT, lengthU )
%GETBOUNDARYSPLINES ─────────────────────────────
%
%   Returns the master slave adresses of interior boundary points

bbSbottomtop = zeros((2*lengthT+2*lengthU−4)/4,2);
bbSleftright = zeros((2*lengthT+2*lengthU−4)/4,2);


indexBb=1;
indexBl=1;
indexBt=1;
indexBr=1;
%loops through and adds any splines on the interior boundary to
%the respective arrays.
for i=1:length(GToL(:,1))
    if GToL(i,2) == 2
            bbSbottomtop(indexBb,1)=i;
            indexBb=indexBb+1;
    end
    if GToL(i,1) == 2
            bbSleftright(indexBl,1)=i;
            indexBl=indexBl+1;
    end
    if GToL(i,2) == lengthU−1

            bbSbottomtop(indexBt,2)=i;
            indexBt=indexBt+1;

    end

    if GToL(i,1) == lengthT−1

            bbSleftright(indexBr,2)=i;
            indexBr=indexBr+1;

    end
end
end
```

getGaussPoints:

```
function [weightXi1, weightXi2, xi1, xi2, PointsInElementXi1, ...
    PointsInElementXi2, EndpointsofE] = getGaussPoints(uniqueT, ...
    uniqueU, lengthT, lengthU, p1,p2)
%GETGAUSSPOINTS Summary of this function goes here
%   Returns the gauss points in each parametric direction
% and the weights
% points to be evaluated:
% lgwt(n1,x1,x2) is a function that returns [gp, weights]
% n1=number of points
% gp=gauss points on interval [x1,x2] and
% weights = gauss weights for gp points.
n1=p1+1;
n2=p2+1;
EndpointsofE=zeros(lengthT*lengthU,4);
%calculating the xi1 gauss points
xi1 = zeros((n1)*(length(uniqueT)-1),1);
for i=1:length(uniqueT)-1
xi1((i-1)*n1+1:(i-1)*n1+n1)=lgwt(n1,uniqueT(i),uniqueT(i+1));
end
%calculating the xi2 gauss points
xi2 = zeros((n2)*(length(uniqueU)-1),1);
for i=1:length(uniqueU)-1
xi2((i-1)*n2+1:(i-1)*n2+n2)=lgwt(n2,uniqueU(i),uniqueU(i+1));
end
%Calculating xi1 endpoints of elements
PointsInElementXi1=zeros((lengthT)*(lengthU),1);
elemcounter=1;
for i=1:(lengthU)
    for j=1:(lengthT)
        PointsInElementXi1(elemcounter)=(n1*j-(n1-1));
        EndpointsofE(elemcounter,1)=uniqueT(j);
        EndpointsofE(elemcounter,2)=uniqueT(j+1);
        elemcounter=elemcounter+1;
    end
end
%Calculating xi2 endpoints of elements
PointsInElementXi2=zeros((lengthT)*(lengthU),1);
elemcounter=1;
for i=1:(lengthT)
    for j=1:(lengthU)
        PointsInElementXi2(elemcounter)=(n2*i-(n2-1));
        EndpointsofE(elemcounter,3)=uniqueU(i);
        EndpointsofE(elemcounter,4)=uniqueU(i+1);
        elemcounter=elemcounter+1;
    end
end
%getting gauss weights
[¬, weightXi1] = lgwt(n1,-1,1);
[¬, weightXi2] = lgwt(n2,-1,1);
```

```
end
```

getSplineValsAnd2Derivs:

```
function [Vals, Derivs, SecondDerivs] = ...
    getSplineValsAnd2Derivs(p, knot, xi)
%function [ Vals, Derivs ] = getSplineValsAndDerivs(p, knot,xi)
%INPUTS
%            p - order
%            knot - knot vector
%            xi - vector of xi points; points to evaluate
%OUTPUTS
%            Vals - Matrix with the value of all spline functions
%            at all xi points.
%            Derivs - Matrix with the value of all derivatives at
%        all xi points
%        SecondDerivs - Matrix with the value of all second
%        derivatives at all xi.

%Creating variables
Vals = zeros(length(knot)-p-1,length(xi));
Derivs= zeros(length(knot)-p-1,length(xi));
SecondDerivs= zeros(length(knot)-p-1,length(xi));
%Switching the knot vector around for compatibility
knot=knot';

%Looping over all xi points
for xipoint = 1:length(xi)

%finding location of last non-zero basis function of xi,  in knot
    %So far assuming p+1 regular knots...
    mu = 1+p;
    while mu<length(knot)-p-1 && xi(xipoint) ≥ knot(mu+1),
        mu = mu+1;
    end

%Finding values of the splines at xi point
     Bp = zeros(p+2,p+1);
    Bp(end-1, 1) = 1;
    for i=1:p,
        for k=mu-i:mu,
            %Accounting for equal knots
        %(leaves no contribution if equal)
            if knot(k+i) ≠ knot(k),
                Bp(k-mu+p+1, i+1) =Bp(k-mu+p+1, i+1) + ...
                    (xi(xipoint) - ...
                    knot(k))/(knot(k+i)-knot(k))*Bp(k-mu+p+1,i);
            end
            if knot(k+i+1) ≠ knot(k+1),
                Bp(k-mu+p+1, i+1) = Bp(k-mu+p+1, i+1) + ...
                    (knot(k+i+1)-xi(xipoint))/( ...
                        knot(k+i+1)−knot(k+1))*Bp(k−mu+p+2, i);
            end
        end
    end
%Finding values of derivatives at xi point
    D=zeros(p+1,1);
    for i=(mu−p):mu
    %Accounting for equal knots
        if knot(i+p) ≠ knot(i)
            D(i−mu+p+1) = D(i−mu+p+1) + ...
                p/(knot(i+p)−knot(i))*Bp(i−mu+p+1,p);
        end
        if knot(i+p+1) ≠ knot(i+1)
            D(i−mu+p+1) = D(i−mu+p+1) − ...
                p/(knot(i+p+1)−knot(i+1))*Bp(i−mu+p+2,p);
        end
    end

    %Finding values of second derivatives
    D2a=zeros(p+2,1);
    D2b=zeros(p+1,1);
if p>2
    k=p−1;
    for j = mu−k:mu
    %accounting for equal knots
       if knot(j+k)≠knot(j)
          D2a(j−mu+k+2) = D2a(j−mu+k+2) + ...
              1/(knot(j+k)−knot(j))*Bp(j−mu+k+2,k);
       end
        if knot(j+k+1)≠knot(j+1)
          D2a(j−mu+k+2) = D2a(j−mu+k+2) − ...
              1/(knot(j+k+1)−knot(j+1))*Bp(j−mu+k+3,k);
        end
    end
    k=p;
    for j = mu−k:mu
    %accounting for equal knots
       if knot(j+k)≠knot(j)
          D2b(j−mu+k+1) = D2b(j−mu+k+1) + ...
              1/(knot(j+k)−knot(j))*D2a(j−mu+k+1);
       end
        if knot(j+k+1)≠knot(j+1)
          D2b(j−mu+k+1) = D2b(j−mu+k+1) − ...
              1/(knot(j+k+1)−knot(j+1))*D2a(j−mu+k+2);
        end
    end

end


%Adding current spline values to Vals
Vals(mu−p:mu,xipoint)=Bp(1:end−1,p+1);
```

```
%Adding current spline derivative Values to Derivs
Derivs(mu−p:mu, xipoint)=D;
%Adding second derivatives to Secondderivs.
coeff=factorial(p)/(factorial(p−2));
Dprefinal=coeff*D2b;
SecondDerivs(mu−p:mu, xipoint)=Dprefinal;
end
end
```

getInitialConcentration:

```
function [c0] = getInitialConcentration(size)
%GETINITIALCONCENTRATION Summary of this function goes here
%   c0 = returns initial random concentration in a vector of
%   size(size)

% setting the random values equal at each run to better
% compare results
rng('default')
%concentration average
cA=0.5
%creates a random initial concentration c= cA +/−0.05
c0=cA*ones(size,1)+0.125*(cA−rand(size,1));
end
```

applyPeriodic:

```
function [inRHS] = applyPeriodic(inRHS)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
%getting values
[bBT,bLR,bbBT,bbLR]=getPeriodicBoundary();

%setting one down from top left an one left from bottom right corner
     inRHS(bbBT(1,2))=2*inRHS(bBT(1,1))−inRHS(bbBT(1,1));
     inRHS(bbLR(1,2))=2*inRHS(bLR(1,1))−inRHS(bbLR(1,1));

%setting boundary values equal
for i=1:length(bBT)
    inRHS(bBT(i,2))=inRHS(bBT(i,1));
end
for i=1:length(bLR)
    inRHS(bLR(i,2))=inRHS(bLR(i,1));
end

%setting boundary derivatives equal
 for i=2:length(bbBT)
     inRHS(bbBT(i,2))=2*inRHS(bBT(i,1))−inRHS(bbBT(i,1));
 end
 for i=2:length(bbLR)
```

```
      inRHS(bbLR(i,2))=2*inRHS(bLR(i,1))−inRHS(bbLR(i,1));
 end

end
```

getMassMatrix

```
function [M] = getMassMatrix()
%GETMASSMATRIX―――――――――――――――――
%   Returns the mass matrix for the Cahn Hilliard equations

%getting variables
[p1,p2]=getP();
NoE=getNoE();
[SinE,GToL]=getConnect();
[weightXi1,weightXi2]=getWeights();
[PinEXi1,PinEXi2]=getPinE();
[N1,¬,¬,N2,¬,¬]=getSplineValues();
size=getSize();
EndpointsofE = getEndpoints();


M = zeros(size);
%integrating
%over each element
for e = 1:NoE
    %over all nonzero splines in element.
    for i = length(SinE(:,e)):−1:1
        iglob=SinE(i,e);
        ilok=GToL(iglob,1);
        jlok=GToL(iglob,2);
    %all other nonzero splines in element.
        for j = length(SinE(:,e)):−1:1
            jglob=SinE(j,e);
            iilok=GToL(jglob,1);
            jjlok=GToL(jglob,2);

            %Gauss Quadrature
            cumulateM=0;
            for g2=1:p2+1 %for all gauss points
                for g1=1:p1+1
                    cumulateM=cumulateM+weightXi1(g1)*...
weightXi2(g2)*(N1(ilok,PinEXi1(e)+g1−1)*N2(jlok,PinEXi2(e)+g2−1)*...
N1(iilok,PinEXi1(e)+g1−1)*N2(jjlok,PinEXi2(e)+g2−1));
                end
            end
        %Adding in global mass matrix.
            M(iglob,jglob)=M(iglob,jglob)+((EndpointsofE(e,2)−...
EndpointsofE(e,1)))*((EndpointsofE(e,4)−EndpointsofE(e,3)))*0.25*...
cumulateM;
        end
    end
end
end
```

getTransformationBoundary:

```
function [TT] = getTransformationBoundary(iscalculate)
%GETTRANSFORMATIONBOUNDARY −−−−−−−
%   This function sets up the multipoint constraints
%   transformation matrix if a input argument is sent,
%   otherwise, it returns the transformation matrix T.
persistent Tmatrix;

if nargin==1
[bBT,bLR,bbBT,bbLR]=getPeriodicBoundary();

T=eye(getSize());
%corners Values

T(bLR(1,2),:)=zeros;
T(bLR(1,2),bBT(1,1))=1;

T(bBT(1,2),:)=zeros;
T(bBT(1,2),bBT(1,1))=1;


T(bLR(length(bLR),2),:)=zeros;
T(bLR(length(bLR),2),bBT(1,1))=1;


%General boundary values
for i=2:length(bBT)−2
    T(bBT(i,2),:)=zeros;
    T(bBT(i,2),bBT(i,1))=1;
end
for i=2:length(bLR)−2
    T(bLR(i,2),:)=zeros;
    T(bLR(i,2),bLR(i,1))=1;
end


%Derivatives
%Side derivative values
for i=3:length(bbBT)−2
    T(bbBT(i,2),:)=zeros;
    T(bbBT(i,2),bBT(i,1))=2;
    T(bbBT(i,2),bbBT(i,1))=−1;
end
for i=3:length(bbLR)−2
    T(bbLR(i,2),:)=zeros;
    T(bbLR(i,2),bLR(i,1))=2;
    T(bbLR(i,2),bbLR(i,1))=-1;
end


% Special values because of chained dependencies
% in the multipoint constraints
%Top left -1,
T(bLR(length(bLR)-1,1),:)=zeros;
T(bLR(length(bLR)-1,1),bLR(1,1))=2;
T(bLR(length(bLR)-1,1),bLR(2,1))=-1;

%Bottom right -1
T(bBT(length(bBT)-1,1),:)=zeros;
T(bBT(length(bBT)-1,1),bBT(2,1))=-1;
T(bBT(length(bBT)-1,1),bBT(1,1))=2;

%The right of top left
T(bbBT(2,2),:)=zeros;
T(bbBT(2,2),bbBT(2,1))=-1;
T(bbBT(2,2),bBT(2,1))=2;

%The left of bottom right
T(bbLR(2,2),:)=zeros;
T(bbLR(2,2),bbLR(2,1))=-1;
T(bbLR(2,2),bLR(2,1))=2;

%Top right
%1:
T(bbLR(length(bbLR),2),:)=zeros;
T(bbLR(length(bbLR),2),bbLR(1,1))=-1;
T(bbLR(length(bbLR),2),bLR(1,1))=2;

%2: Corner - already implemented

%3:
T(bbLR(length(bbLR)-1,2),:)=zeros;
T(bbLR(length(bbLR)-1,2),bbLR(1,1))=-2;
T(bbLR(length(bbLR)-1,2),bbLR(2,1))=1;
T(bbLR(length(bbLR)-1,2),bLR(2,1))=-2;
T(bbLR(length(bbLR)-1,2),bLR(1,1))=4;

%4:
T(bbBT(length(bbBT),2),:)=zeros;
T(bbBT(length(bbBT),2),bLR(2,1))=-1;
T(bbBT(length(bbBT),2),bLR(1,1))=2;

Tmatrix=T;
end
TT=Tmatrix;
end
```

getBandCterms:

```
function [outp] = getBandCterms(t,u)
%GETBANDCTERMS ——————————————————
%   This function returns the right hand side of the
%   Cahn—Hilliard equation


%getting values
cn=u;
[p1,p2]=getP();
NoE=getNoE();
[SinE,GToL]=getConnect();
[weightXi1,weightXi2]=getWeights();
[PinEXi1,PinEXi2]=getPinE();
[N1,N1D,N1DD,N2,N2D,N2DD]=getSplineValues();
size=getSize();
EndpointsofE = getEndpoints();
[alpha,theta]=getCHconstants();


%allocating variables
term1=zeros(size,1);
term2=zeros(size,1);


%integration loop
%over each element
for e = 1:NoE
%over all nonzero splines in element.
    for i = length(SinE(:,e)):—1:1
        iglob=SinE(i,e);
        ilok=GToL(iglob,1);
        jlok=GToL(iglob,2);
            for g2=1:p2+1 %over all gauss points
                for g1=1:p1+1
                    c=0;
                    cx=0;
                    cy=0;
                    cxx=0;
                    cyy=0;
            %all other nonzero splines in element.
                    for j = length(SinE(:,e)):—1:1
                        jglob=SinE(j,e);
                        iilok=GToL(jglob,1);
                        jjlok=GToL(jglob,2);
            %Cumulating values of c,cx,cy,cxx,cyy
                        c=c+N1(iilok,PinEXi1(e)+g1—1)*...
N2(jjlok,PinEXi2(e)+g2—1)*cn(jglob);
                        cx=cx+N1D(iilok,PinEXi1(e)+g1—1)*...
N2(jjlok,PinEXi2(e)+g2—1)*cn(jglob);
                        cy=cy+N1(iilok,PinEXi1(e)+g1—1)*...
N2D(jjlok,PinEXi2(e)+g2—1)*cn(jglob);
```

```
                        cxx=cxx+N1DD(iilok,PinEXi1(e)+g1−1)*...
N2(jjlok,PinEXi2(e)+g2−1)*cn(jglob);
                        cyy=cyy+N1(iilok,PinEXi1(e)+g1−1)*...
N2DD(jjlok,PinEXi2(e)+g2−1)*cn(jglob);

                    end
            %Calculating Mc, muc and the neccecary
            %derivatives
                    Mc=c*(1−c);
                    delMc=[cx−2*c*cx;cy−2*c*cy];
                    del2c=cxx+cyy;
                    delMuc=(3*alpha)/(2*theta)*[(cx/c)+...
(cx/(1−c))−(4*theta*cx);(cy/c)+(cy/(1−c))−(4*theta*cy)];

                    %scaling and weights for gauss integration:
                    snw=((EndpointsofE(e,2)−EndpointsofE(e,1)))...
*((EndpointsofE(e,4)−EndpointsofE(e,3)))*0.25*weightXi1(g1)*...
weightXi2(g2);

            %Adding/integrating contributions to
            %the two right hand side terms
                    term2(iglob)=term2(iglob)+snw*...
(N1DD(ilok,PinEXi1(e)+g1−1)*N2(jlok,PinEXi2(e)+g2−1)+...
N1(ilok,PinEXi1(e)+g1−1)*N2DD(jlok,PinEXi2(e)+g2−1)*...
(Mc*(del2c)));
                    term1(iglob)=term1(iglob)+snw*...
([N1D(ilok,PinEXi1(e)+g1−1)*N2(jlok,PinEXi2(e)+g2−1); ...
N1(ilok,PinEXi1(e)+g1−1)*N2D(jlok,PinEXi2(e)+g2−1)]'*...
(Mc*delMuc+delMc*(del2c)));
                end
            end
    end

end
%output the combined right hand side with implemented
%periodic boundary conditions
outp=applyBoundaryRHS((−1*term1−term2));

end
```

applyBoundaryRHS:

```
function [inRHS] = applyBoundaryRHS(inRHS)
%APPLYBOUNDARYRHS————————————
%Transforms right hand side according to periodic
%boundary transformation

%getting variables
T=getTransformationBoundary();

%calculating
```

```
inRHS=(T.')*inRHS;

end
```

performReinsertion:

```
function [Y] = performReinsertion(Y)
%UNTITLED ------------
%   Reinserts the missing (because of boundary conditions) DOFs
%   using the transformation matrix.

%getting variables
[bBT,bLR,bbBT,bbLR]=getPeriodicBoundary();

%for every returned time-step from ode15s solver
for iii = 1:length(Y(:,1))
T=getTransformationBoundary();
Y(iii,:)=(T*(Y(iii,:).')).';
%only because some values were slightly over 1,
%ruining the color scale slightly
%the values were capped at 1:
    for ii = 1:length(Y(1,:))
        if Y(iii,ii)>1
            Y(iii,ii)=1;
        end
        if Y(iii,ii)<0
            Y(iii,ii)=0;
        end
    end

end
```