Block ILU smoothers for p-multigrid methods in Isogeometric Analysis

Mark Looijé

Block ILU smoothers for p-multigrid methods in Isogeometric Analysis

by

Mark Looije

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday March 18, 2022 at 10:00 AM.

Student number:	4381874	
Project duration:	September 1, 2020 – March 18, 2022	
Thesis committee:	Prof. dr. ir. C. Vuik,	TU Delft, supervisor
	Prof. dr. ir. H. X. Lin,	TU Delft
	R. P. W. M. Tielen,	TU Delft, supervisor

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Isogeometric Analysis (IgA) is an extension of the more well known Finite Element Method (FEM). It allows for more accurate descriptions of boundary value problems on irregular domains. However, many of the traditional iterative solution strategies that are known to work well in FEM do not show the same behavior in IgA, especially for increasing order of basis functions p.

A method shown to have fast convergence in this situation is a p-multigrid method with a smoother based on a Block ILUT factorization. Most of the blocks of this factorization are efficiently calculated. The same holds for the smoothing steps.

It is therefore our objective to make changes to the Block ILUT smoother. Inspiration is taken from methods where ILU factorizations are constructed using a fixed-point iteration. We combine these with the existing Block ILUT smoother. This ultimately leads to two new proposed methods we will call Block Fixed-point ILU and Block ParILUT.

The existing methods as well as the newly suggested methods are tested and compared on computational costs of the factorization, the number of nonzero entries in this factorization and the number of multigrid iterations needed to reach convergence, if these factorizations are to be used as smoother. The benchmark used for these tests is a convection diffusion reaction (CDR) equation on a multipatch geometry with 4, 16 or 64 patches.

Preface and acknowledgments

During the past one and a half years I have been working on this master's thesis "Block ILU smoothers for p-multigrid methods in Isogeometric Analysis", executed at the Numerical Analysis department and supervised by Roel Tielen and Kees Vuik. Doing this master's thesis has been both the final as well as by far the most challenging part of my studies. Did not always enjoy it either, progress and motivation definitely came in waves. But in the end I am proud of what I produced and thankful for the help I have received.

Therefore I want to thank both Roel and Kees for being my supervisors, for guiding me even though I was not always the easiest to work with. Next to them I want to thank my parents and especially the friends I know from outdoor sports association Slopend or my studies. They were the distraction I needed, they are the ones that made my time in Delft enjoyable. And they are the ones that gave the extra support and encouragement when I needed it.

Mark Looije
Delft, March 2022

Contents

Chapter 1

Introduction

"Block ILU smoothers for p-multigrid methods in Isogeometric Analysis".
The title of this thesis consists of several parts. We will start with Isogeometric Analysis, the other sections will follow from this.

Isogeometric Analysis (IgA) is easiest explained and motivated as an extension to the Finite Element Method (FEM), first described by Hughes in 2004 [1]. While FEM is currently the most well known and widely used method for solving problems over irregular domains, this method does have its inefficiencies. The main one being the translation between the geometry created by CAD (Computer Aided Design) and an analysis-suitable geometry. This process is computationally expensive; it is estimated that for complex engineering designs this translation is responsible for 80% of overall analysis time [2]. All while only being an approximation to the 'exact' geometry.
This is the original motivation for IgA; the use of B-Spline basis functions allows for a highly accurate representation of complex geometries as well as establishing a link between the design and the engineering tools. Another positive for IgA is the higher continuity of basis functions, a useful property that is more and more appreciated.

Challenges in IgA include assembly of the system matrix and right hand side vector, which isn't as straightforward as it may be in for example FEM. The main challenge however, lies in the fact that many common solvers for linear systems don't perform well for our IgA discretization.
The proposed method for solving the linear system is a p-multigrid method. In most multigrid methods the coarsening is in the mesh width h, here coarsening is applied in the spline degree p. The choice for p-multigrid makes it that the problem on the coarse grid looks a lot more familiar, where we know how to efficiently solve this coarser problem. But of course, p-multigrid comes with its own intergrid operators and challenges. And it requires a different choice of smoothers.

Since traditional smoothers such as Gauss Seidel do not perform for higher spline degree p we need to consider alternative smoothers. For this smoothers are suggested based on incomplete LU factorizations. After first discussing a few more elementary factorizations we build up towards Block ILUT, a method taking advantage of the block structure of our system matrix. This method is often used in the work of Tielen [3],[4],[5],[6],[7] and shown to work very well in the context of Isogeometric Analysis.
In the final parts of this thesis we will try to make alterations to the Block ILUT smoother, searching for further improvements. Here new methods are proposed with incomplete LU factorizations based on fixed-point iterations, while also using the existing block structure.

The layout of the core of this thesis follows the same structure. We will start with a description of IgA and how to go from a boundary value problem to a linear system of equations (Chapter 2). Strategies are proposed on how to efficiently solve these systems using multigrid methods (Chapter 3). This is followed by two chapters regarding ILU type smoothers, or ILU type smoothers applied in this setting more specifically. Here chapter 4 describes methods used before by Tielen or others (though explained more elaborately and with reproduced test results). Chapter 5 is my actual contribution to this subject, describes my own research.

Chapter 2

Isogeometric Analysis

2.1. Variational Formulation and Geometry Function

As mentioned before, IgA is used to find an approximation to boundary value problems. For instance, the homogeneous Poisson equation could be considered.

$$\begin{cases} -\Delta u &= f \text{ in } \Omega, \\ u &= 0 \text{ on } \partial\Omega. \end{cases}$$

Here Ω is an open domain in $\mathbb{R}^d$, $\partial\Omega$ the boundary of Ω, $f \in L^2(\Omega)$.

2.1.1. Variational Form

As is the case with FEM, the solution stategy is based on a variational formulation (also known as weak formulation). To find the variational formulation to this boundary value problem, we consider the function space V. Here V is defined as

$$V := \{v \in H^1(\Omega),\ v = 0 \text{ on } \partial\Omega\},$$

where $H^1(\Omega)$ the first order Sobolev space. This space consists of all functions $v \in L_2(\Omega)$ that have weak and square-integrable first derivatives as well as being 0 on the boundary. Please note that the solution u to the boundary value problem must lie in this set V.

The variational form is obtained by multiplying the PDE with a test function $v \in V$ and integrating over Ω. Then integration by parts and Gauss' divergence theorem are applied. The boundary integral vanishes as v is zero on the boundary.

$$-\int_\Omega \Delta u v \, \mathrm{d}\Omega = \int_\Omega f v \, \mathrm{d}\Omega,$$

$$-\int_\Omega \operatorname{div}(\nabla u v) \, \mathrm{d}\Omega + \int_\Omega \nabla u \nabla v \, \mathrm{d}\Omega = \int_\Omega f v \, \mathrm{d}\Omega,$$

$$-\int_{\partial\Omega} \nabla u v \cdot n \, \mathrm{d}\Gamma + \int_\Omega \nabla u \nabla v \, \mathrm{d}\Omega = \int_\Omega f v \, \mathrm{d}\Omega,$$

$$\int_\Omega \nabla u \nabla v \, \mathrm{d}\Omega = \int_\Omega f v \, \mathrm{d}\Omega,$$

which will be abbreviated as

$$a(u, v) = \langle f, v \rangle .$$

For the boundary value problem to hold, $a(u, v) = \langle f, v \rangle$ must hold for every $v \in V$. Hence solving the variational form for every such v will give the desired solution to the boundary value problem.

To discretize the variational form, V is replaced by a finite dimensional subspace $V_h \subseteq V$. Let $\phi_1, ..., \phi_n$ be a basis of V_h, then the numerical approximation $u_h \in V_h$ is constructed as the linear combination of these basis functions:

$$u_h = \sum_{i=1}^{n} u_i \phi_i.$$

Inserting u_h into the variational form and testing with $v = \phi_i$ for $i = 1, ..., n$ we find

$$a(u_h, \phi_i) = \sum_{i=1}^{n} u_i \ a(\phi_i, \phi_j) \text{ for } j = 1, ..., n$$

And therefore we obtain the linear system

$$A\mathbf{u} = \mathbf{f}, \text{ where}$$

$$A_{i,j} = a(\phi_i, \phi_j),\ f_i = \langle f, \phi_i \rangle \quad i, j = 1, ..., n.$$

Determining the coefficients u_i through this linear system will give the approximation to the variational form and therefore the boundary value problem [8],[9].

2.1.2. Geometry function

Now, suppose the physical domain is parametrized. Suppose that there is some geometry function **F** that is an invertible mapping from parameter domain Ω_0 to physical domain Ω (see figure 2.1).

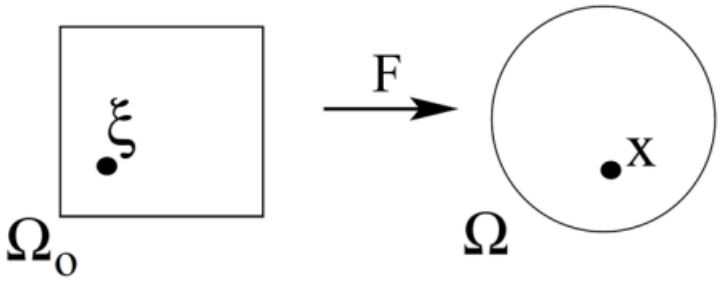


Figure 2.1: Parametrization of Ω. The parametric domain is denoted by Ω_0. From: [8]

The following is a well known integration rule

$$\int_\Omega w(\boldsymbol{x}) d\boldsymbol{x} = \int_{\Omega_0} w(\boldsymbol{F}(\boldsymbol{\xi})) |\det \boldsymbol{DF}(\boldsymbol{\xi})| d\boldsymbol{\xi},$$

with $\boldsymbol{DF}(\boldsymbol{\xi}) = (\frac{\partial F_i}{\partial \xi_j})_{i,j=1,...,d}$ the Jacobian matrix.

Applying this to the integrals in the variational form we find the following [8]:

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v d\boldsymbol{x} = \int_{\Omega_0} (\nabla u \boldsymbol{DF}(\boldsymbol{\xi})^{-1}) \cdot (\nabla v \boldsymbol{DF}(\boldsymbol{\xi})^{-1}) |\det \boldsymbol{DF}(\boldsymbol{\xi})| d\boldsymbol{\xi},$$

$$\langle f, v \rangle = \int_\Omega f v d\boldsymbol{x} = \int_{\Omega_0} (f v)(\boldsymbol{F}(\boldsymbol{\xi})) |\det \boldsymbol{DF}(\boldsymbol{\xi})| d\boldsymbol{\xi}.$$

This is slightly different to what is done in FEM. There the geometry transformation is applied later, on integrals over elements, not the entire domain. As the grid in FEM is (often) build from triangles, this transformation is a linear transformation. The basis functions are often tent-functions, but also other choices for basis functions can be made [9].

In IgA B-splines (or more generally NURBS) are used to define **F**. For the basis functions we will use B-spline basis functions, which will be defined in section 2.2.

2.2. B-splines

An example of a B-spline curve can be seen in figure 2.2. B-spline curves, surfaces and solids are excellent for describing complicated shapes. Next to describing the geometry, the B-spline basis functions are also used as the basis functions in our IgA discretization.

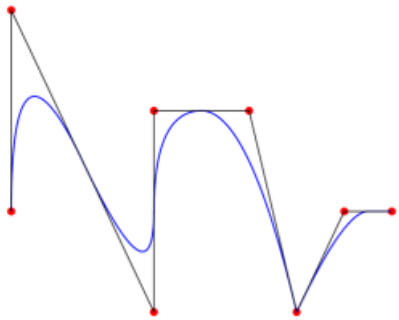


Figure 2.2: An example of a B-spline, in this case a piecewise quadratic curve in $\mathbb{R}^2$. From: [1]

2.2.1. B-spline basis functions

B-spline basis functions are piecewise polynomials and defined using a knot vector. A knot vector in one dimension is a set of coordinates in the parametric space, written $\Xi = \{\xi_1, \xi_2, ..., \xi_{n+p+1}\}$. Here $\xi_i \in \mathbb{R}$ is the i-th knot, i the knot index, p the polynomial order, and n the number of basis functions which compromise the B-spline.
For instance, a typical knot vector for piecewise quadratic ($p = 2$) polynomials would be $\{0,0,0,\frac{1}{8},\frac{1}{4},\frac{3}{8},\frac{1}{2},\frac{5}{8},\frac{3}{4},\frac{7}{8},1,1,1\}$. This vector is open (as it repeats the endpoints p times) and uniform (the distances between the knots is uniform).

B-spline basis functions are defined recursively

$$\phi_{i,0}(\xi) = \begin{cases} 1 & \text{if } \xi_i \leq \xi < \xi_{i+1}, \\ 0 & \text{else,} \end{cases}$$

$$\phi_{i,p}(\xi) = \begin{cases} \frac{\xi - \xi_i}{\xi_{i+p} - \xi_i}\phi_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{\xi_{i+p+1} - \xi_{i+1}}\phi_{i+1,p-1}(\xi) & \text{if well defined,} \\ 0 & \text{else.} \end{cases}$$

This recursion is known as the Cox-de Boor formula. Due to this recursive definition the derivatives of B-spline basis functions are efficiently represented in terms of the lower order bases

$$\frac{d}{d\xi}\phi_{i,p}(\xi) = \begin{cases} \frac{p}{\xi_{i+p} - \xi_i}\phi_{i,p-1}(\xi) - \frac{p}{\xi_{i+p+1} - \xi_{i+1}}\phi_{i+1,p-1}(\xi) & \text{if well defined,} \\ 0 & \text{else.} \end{cases}$$

Similar expressions exist for the higher order derivatives, but we don't deem it necessary discussing them here.

Examples of some lower order basis functions can be seen in figure 2.3. The knot vector used is $\{0,\frac{1}{8},\frac{1}{4},\frac{3}{8},\frac{1}{2},\frac{5}{8},\frac{3}{4},\frac{7}{8},1\}$ for $p = 0$ and expended with additional zeros and ones for increasing p.

Some important properties of the B-spline basis functions include [1],[2]:

- They add to one, that is, $\forall\xi \sum_{i=1}^{n} \phi_{i,p}(\xi) = 1$. This is known as the partition of unity property.
- They are non-negative, that is $\phi_{i,p}(\xi) \geq 0 \ \forall\xi$.
- The support of each $\phi_{i,p}$ is compact and contained in the interval $[\xi_i, \xi_{i+p+1}]$.
- They are C^{p-1}(that is, they have continuous derivatives up to order $p-1$), if there are no repeated knots. They are C^{p-k-1} if a knot is repeated k times [1],[2].

These properties give for some usefull consequences. For example, the support structure gives for sparse matrices. It can also be seen that the amount of non-zero entries increases for higher polynomial order p. The mass matrix M (more on this in later chapters) will be easy to lump due to the partition of unity property, etc.

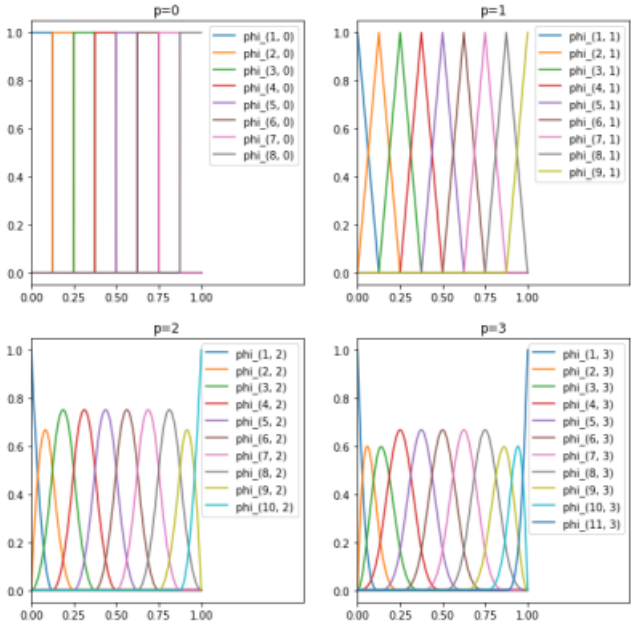


Figure 2.3: Basis functions of order 0,1,2,3 for an open uniform knot vector

2.2.2. B-spline curves, surfaces and solids

B-spline curves in $\mathbb{R}^d$ are constructed by taking linear combinations of B-spline basis functions. The coefficients are the control points. Given n basis functions, $\phi_{i,p}$, $i = 1,2,...,n$ and corresponding control points $B_i \in \mathbb{R}^d$, $i = 1,2,...,n$ a piecewise-polynomial B-spline curve is given by

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

In general the control points are not interpolated by B-spline curves. Their continuity is inherited by the continuity of the B-spline basis functions, though now the continuity is also decreased for repeated control points. Both this interpolation of control points (marked in red) and the decrease in continuity can be seen in figure 2.2.

B-spline surfaces and B-spline solids are defined in a similar manner. For the B-spline surfaces consider a control net $B_{i,j}$, $i = 1,...,n$, $j = 1,...,m$ and knot vectors $\Xi = \{\xi_1,...,\xi_{n+p+1}\}$ and $H = \{\eta_1,...,\eta_{m+q+1}\}$. Then the tensor product B-spline surface is defined by

$$S(\xi,\eta) = \sum_{i=1}^{n} \sum_{j=1}^{m} \phi_{i,p}(\xi) \psi_{j,q}(\eta) B_{i,j}$$

The formulation for a B-spline solid contains a control net $B_{i,j,k}$ with 3 indices, 3 knot vectors and 3 sets of basis functions.

2.2.3. Refinement strategies

In the multigrid solver it will be necessary to describe B-splines using different sets of basis functions and control points. Several strategies are known for refining the B-spline basis through expanding the knot vector Ξ and adjusting the control points. These strategies are called h-refinement, p-refinement and k-refinement [1],[2].
h- and p-refinement also exist in FEM and are known as knot insertion and order elevation respectively. k-refinement is exclusive to IgA. They are best explained through a simple example:

h-refinement: $\{0,0,\frac{1}{2},1,1\}$ to $\{0,0,\frac{1}{4},\frac{1}{2},\frac{3}{4},1,1\}$; additional distinct knots are inserted.
p-refinement: $\{0,0,\frac{1}{2},1,1\}$ to $\{0,0,0,\frac{1}{2},\frac{1}{2},1,1,1\}$; the multiplicity of every knot is increased.
k-refinement: $\{0,0,\frac{1}{2},1,1\}$ to $\{0,0,0,\frac{1}{2},1,1,1\}$; the multiplicity of the endpoints is increased.
The impact of these refinements on the basis functions can be seen in figure 2.4.

Important observations to make: The number of basis functions almost doubles for h- and p-refinement (though this is better seen in bigger examples), while for k-refinement the number of basis functions only increases by one. Both p- and k-refinement allow for order elevation of the basis functions, though another major benefit of k-refinement opposed to p-refinement being a better continuity for the internal knots.

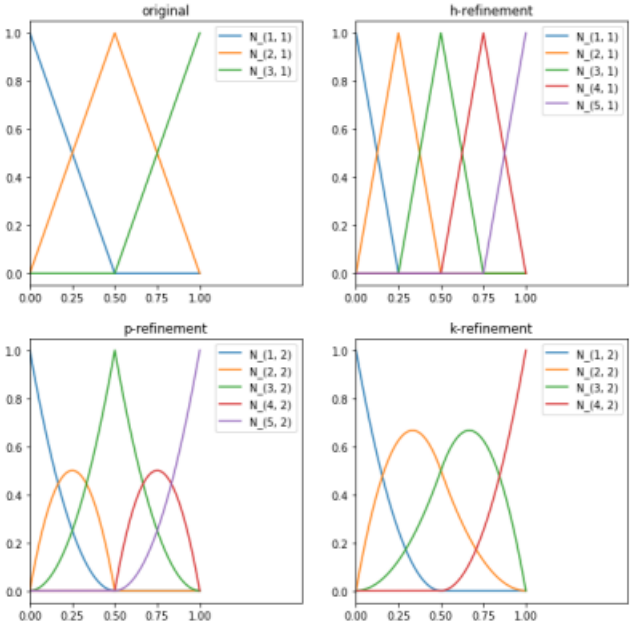


Figure 2.4: Different refinement techniques

Throughout this literature study, we adopt k-refinement unless stated otherwise. In chapter 3 it is further elaborated how changing the knot vector and basis functions impacts the coefficients.

2.3. Matrix Assembly

For the matrix assembly a method is shown that is very similar to the method used in FEM. The idea is to decompose the integral over the entire parameter domain Ω_0 into rectangular or cuboid atomic contributions and apply standard cubature rules such as Gauss formulae on each rectangle resp. cuboid afterwards.

Using system matrix A and right hand side vector $\mathbf{f}$ from section 2.1, assuming $\mathbf{F} = \mathbf{id}$ so that the expressions remain readable, we receive

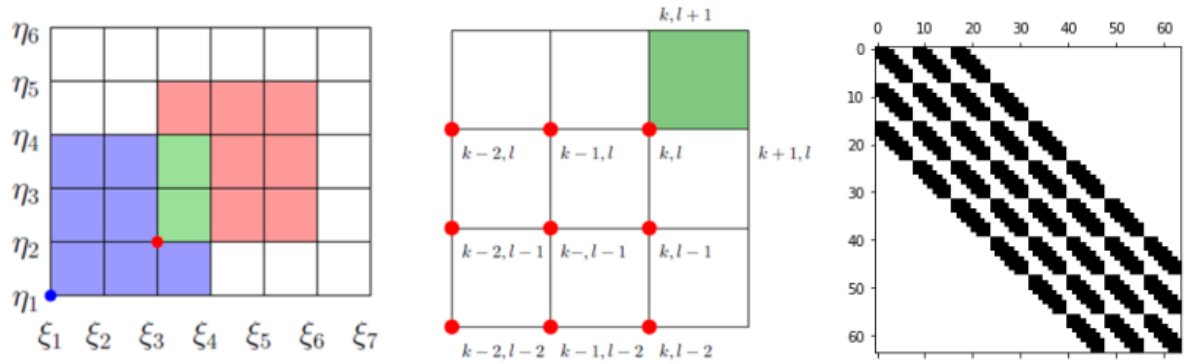
$$A_{i,j} = \int_\Omega \nabla\phi_i \nabla\phi_j \, \mathrm{d}\Omega = \sum_{k=1}^{N_{\text{el}}} \int_{\Omega_k} \nabla\phi_i \nabla\phi_j \, \mathrm{d}\Omega,$$

$$f_i = \int_\Omega f\phi_i \, \mathrm{d}\Omega = \sum_{k=1}^{N_{\text{el}}} \int_{\Omega_k} f\phi_i \, \mathrm{d}\Omega.$$

2.3.1. Support of basis functions

In section 2.2.1 the support structure of the basis functions was already mentioned. They are zero over most elements, making the integrals over those elements zero by default, thus reducing the amount of integrals that have to be calculated.

The support of these basis functions can be visualized in several ways [2]. The first being: given a basis function, in what elements is this basis function unequal to zero (see figure 2.5a)? The blue area is the support of basis function $\phi_{1,1}$, the red area the support of $\phi_{3,2}$, the green area is where both basis functions are supported. $p = 2$ in this image.
Another way to look at it: given an element, what basis functions are supported on this element (see figure 2.5b)? Again for $p = 2$, the red dots show what basis functions are relevant for integrating over the green area.
Finally, in figure 2.5c the nonzero pattern of the matrix A is shown for $p = 2$ and $N_{\text{el}} = 8$ in both directions. As $p = 2$ we see two non-zero entries to the left and two non-zero entries to the right of the diagonal. As well as two blocks to the left and on the right respectively.



(a) bsplinesupport interpretation 1 (b) bsplinesupport interpretation 2 (c) sparsitypattern A($p = 2, N_{\text{el}} = 8$)

Figure 2.5: Support of B-spline basis functions visualization. (a),(b) from: [2]

2.3.2. Integral approximation

The selected approximation method is Gauss quadrature.

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

An n-point Gauss quadrature rule is exact for polynomials of degree $2n-1$, by suitable choice of quadrature points x_i and weights w_i [10]. These points and weights for the standard interval $[-1,1]$ can be found in table **??**.

Considering for example the expressions for $A_{i,j}$ earlier this section, degree of basis functions $p = 2$. Then both $\nabla\phi_i$ and $\nabla\phi_j$ would have have degree 1, hence the integrand having polynomial order 2. 2 quadrature points would be sufficient to exactly calculate the integral.

Generally, the geometry function $\mathbf{F}$ isn't simply the identity, the integrand isn't a polynomial. Still, Gauss quadrature works quite well for these functions.

Number of points, n	Points, x_i		Weights, w_i	
1	0		2	
2	$\pm\frac{1}{\sqrt{3}}$	$\pm 0.57735...$	1	
3	0		$\frac{8}{9}$	0.888889...
	$\pm\sqrt{\frac{3}{5}}$	$\pm 0.774597...$	$\frac{5}{9}$	0.555556...
4	$\pm\sqrt{\frac{3}{7}-\frac{2}{7}\sqrt{\frac{6}{5}}}$	$\pm 0.339981...$	$\frac{18+\sqrt{30}}{36}$	0.652145...
	$\pm\sqrt{\frac{3}{7}+\frac{2}{7}\sqrt{\frac{6}{5}}}$	$\pm 0.861136...$	$\frac{18-\sqrt{30}}{36}$	0.347855...
5	0		$\frac{128}{255}$	0.568889...
	$\pm\frac{1}{3}\sqrt{5-2\sqrt{\frac{10}{7}}}$	$\pm 0.538469...$	$\frac{322+13\sqrt{70}}{900}$	0.478629...
	$\pm\frac{1}{3}\sqrt{5+2\sqrt{\frac{10}{7}}}$	$\pm 0.90618...$	$\frac{322-13\sqrt{70}}{900}$	0.236927...

Table 2.1: Gauss quadrature points and weights. From: [11]

Change of interval

Table 2.1 gives the points and weights for an integral over $[-1,1]$, however this isn't the domain we are interested in. This is easily accounted for using a simple transformation [12]. An integral over $[\xi_i,\xi_{i+1}]$ is approximated

$$\int_{\xi_i}^{\xi_{i+1}} f(\xi)d\xi = \frac{\xi_{i+1}-\xi_i}{2}\int_{-1}^{1} f\Big(\frac{\xi_{i+1}-\xi_i}{2}x+\frac{\xi_i+\xi_{i+1}}{2}\Big)dx$$

$$\approx \frac{\xi_{i+1}-\xi_i}{2}\sum_{k=1}^{N_{\text{el}}} w_i f\Big(\frac{\xi_{i+1}-\xi_i}{2}x_i+\frac{\xi_{i+1}-\xi_i}{2}\Big).$$

The quadrature rule is easily expanded to squares or cubes. For example, for $p=3$ and integrating over $[-1,1]^2$, the quadrature points are $\{-\sqrt{\frac{3}{5}},0,\sqrt{\frac{3}{5}}\}\times\{-\sqrt{\frac{3}{5}},0,\sqrt{\frac{3}{5}}\}$. The weights are $(\frac{5}{9},\frac{8}{9},\frac{5}{9})\otimes(\frac{5}{9},\frac{8}{9},\frac{5}{9})$.
For a rectangle $[\xi_i,\xi_{i+1}]\times[\eta_j,\eta_{j+1}]$ we again will have to use a simple transformation.

2.4. Multipatch

So far we have pretended all domains can be mapped onto either the unit square (2D) or the unit cube (3D). But this is of course not the case, in almost all practical circumstances it will be necessary to describe domains using multiple patches.
Most common reason is the situation where the domain simply differs topologically from a square or cube. For example the shape in figure 2.6, it is clear we can't find invertible $\mathbf{F}$ that maps this domain to the unit square. Instead all coloured areas are mapped separately to their own unit squares.

Figure 2.6: Example of a geometry where multiple patches are required. From: [3]

Other reasons to use multiple patches include when different materials or different physical models are used in different parts of the domain. Finally it may be computationally efficient to assemble different subdomains in parallel, which is more convenient when using multiple patches [2].

In multipatch, Ω is divided into a collection of non-overlapping subdomains Ω^k such that $\bar{\Omega} = \bigcup_{k=1}^{K} \bar{\Omega}^k$.
We call Ω a multipatch geometry consisting of k patches. For each Ω^k a geometry function $\mathbf{F}^k$ is then defined to parametrize each subdomain individually

$$\mathbf{F}^k : \Omega_0 \to \Omega^k \qquad \mathbf{F}^k(\boldsymbol{\xi}) = \mathbf{x} \in \Omega^k, \ \forall \boldsymbol{\xi} \in \Omega_0.$$

For illustration, consider the multipatch geometry consisting of 4 patches as shown in figure 2.7. In the same figure you can also see the resulting block structure of the system matrix A.
The first 4 diagonal blocks A_{ii} are associated with the interior degrees of freedom on Ω_i. $A_{\Gamma\Gamma}$ denotes the degrees of freedom at the interface Γ. Finally, the off-diagonal blocks denote the coupling between degrees of freedom at the interior and the interface.
The resulting block structure is called a block arrowhead matrix. This particular block structure is of course only achieved for particular numberings of the degrees of freedom. The blocks $A_{11}, ..., A_{44}$ will look similar to the matrix A in figure 2.5c.

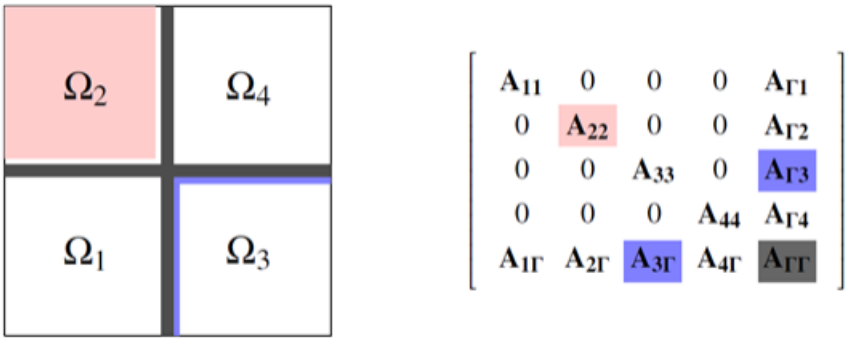


Figure 2.7: A multipatch geometry, consisting of 4 patches and the resulting block structure of the system matrix. From: [3]

Chapter 3

Multigrid Methods

3.1. Introduction Multigrid

In chapter 2 we described how a boundary value problem leads to a linear system of equations, using IgA. This chapter focusses on solving the resulting linear system. In particular, we focus on multigrid methods.
Multigrid methods aim to solve linear systems by combinining a basic iterative method and a correction which is based on a connected and easier problem. Multigrid is shown to be very efficient, especially for large systems [13],[14].

The goal is to determine $\mathbf{u}$ in $A\mathbf{u} = \mathbf{f}$, but the dimensions of this problem are too big to do so effectively with just a basic iterative method.
Therefore consider the connected system $\tilde{A}\tilde{\mathbf{u}} = \tilde{\mathbf{f}}$. Here $\tilde{A}$, $\tilde{\mathbf{u}}$ and $\tilde{\mathbf{f}}$ are connected to the original matrix and vectors via a prolongation operator $\mathcal{I}_P$ ($\mathbf{u} = \mathcal{I}_P\tilde{\mathbf{u}}$) and restriction operator $\mathcal{I}_R$ ($\tilde{\mathbf{u}} = \mathcal{I}_R\mathbf{u}$).
$\tilde{A}$ can be obtained by either rediscretizing the bilinear form or by applying so-called Galerkin coarsening: $\tilde{A} = \mathcal{I}_R A \mathcal{I}_P$.

In the remainder of this section various multigrid algorithms are shown, starting with the most basic application of multigrid. As more and more is added these alterations will be explained and motivated. $\mathbf{v}$ will be used to denote the approximation to $\mathbf{u}$.
Multigrid algorithm 1 shows the use of two different grids and can be used to determine a reasonable initial guess $\mathbf{v}$. It is however of no use in trying to iteratively improving our solution.

ALGORITHM 3.1

Goal: Determine $\mathbf{u}$ in $A\mathbf{u} = \mathbf{f}$.
1. Determine $\tilde{A}$ and $\tilde{\mathbf{f}}$, using $\tilde{A} = \mathcal{I}_R A \mathcal{I}_P$ and $\mathbf{f} = \mathcal{I}_P\tilde{\mathbf{f}}$.
2. Solve for $\tilde{\mathbf{v}}$, using $\tilde{A}\tilde{\mathbf{v}} = \tilde{\mathbf{f}}$.
3. Determine $\mathbf{v}$, by prolongating $\mathbf{v} = \mathcal{I}_P\tilde{\mathbf{v}}$.

To further improve on an initial guess we use the residual equation. That is, instead of working on $\tilde{A}\tilde{\mathbf{u}} = \tilde{\mathbf{f}}$ we work on $\tilde{A}\tilde{\mathbf{e}} = \tilde{\mathbf{r}}$. We see (approximately) how big the error is and thus how much our guess should be corrected. This leads to multigrid algorithm 2. The sequence can be repeated iteratively to further improve your guess $\mathbf{v}$.

ALGORITHM 3.2 (Coarse grid correction)

Goal: Determine $\mathbf{u}$ in $A\mathbf{u} = \mathbf{f}$.
1. Compute the residual $\mathbf{r} = \mathbf{f} - A\mathbf{v}$.
2. Restrict the residual $\tilde{\mathbf{r}} = \mathcal{I}_R\mathbf{r}$.
3. Solve $\tilde{A}\tilde{\mathbf{e}} = \tilde{\mathbf{r}}$.
4. Prolongate the error $\mathbf{e} = \mathcal{I}_P\tilde{\mathbf{e}}$.
5. Update the guess $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{e}$.

It is known from literature that coarse grid correction (algorithm 2) is effective in reducing low frequency components of the error. On the other hand basic iterative methods such as Gauss Seidel or (damped) Jacobi reduce the high frequency components. Hence, they are often used together. A typical multigrid method contains both smoothing steps and a coarse grid correction. Algorithm 3 is an example of this. The number of pre- and postsmoothing steps is denoted by ν_1 and ν_2 respectively.

ALGORITHM 3.3 (Two-grid cycle)

Goal: Determine $\mathbf{u}$ in $A\mathbf{u} = \mathbf{f}$.
1. Relax ν_1 times on $A\mathbf{u} = \mathbf{f}$ with initial guess $\mathbf{v}$.
2. Compute the residual $\mathbf{r} = \mathbf{f} - A\mathbf{v}$.
3. Restrict the residual $\tilde{\mathbf{r}} = \mathscr{I}_R\mathbf{r}$.
4. Solve $\tilde{A}\tilde{\mathbf{e}} = \tilde{\mathbf{r}}$.
5. Prolongate the error $\mathbf{e} = \mathscr{I}_P\tilde{\mathbf{e}}$.
6. Update the guess $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{e}$.
7. Relax ν_2 times on $A\mathbf{u} = \mathbf{f}$ with initial guess $\mathbf{v}$.

The most effective way of solving $\tilde{A}\tilde{\mathbf{e}} = \tilde{\mathbf{r}}$ may be to restrict to an even easier grid. That is, we want to apply the multigrid approach recursively.
This recursive use of multigrid leads to a V-cycle and is described in multigrid program 4. Figure 3.1a shows the V-cycle scheme in a more visual way.

ALGORITHM 3.4 (V-cycle)

Goal: Determine $\mathbf{u}$ in $A\mathbf{u} = \mathbf{f}$.
Relax ν_1 times on $A\mathbf{u} = \mathbf{f}$ with initial guess $\mathbf{v}$.
1. Compute $\tilde{\mathbf{f}} = \mathscr{I}_R\mathbf{r}$.
2. Relax ν_1 times on $\tilde{A}\tilde{\mathbf{u}} = \tilde{\mathbf{f}}$ with initial guess $\tilde{\mathbf{v}} = \mathbf{0}$.
3. Compute $\tilde{\tilde{\mathbf{f}}} = \mathscr{I}_R\tilde{\mathbf{r}}$.

$\vdots$

4. Solve $A^*\mathbf{u}^* = \mathbf{f}^*$.

$\vdots$

5. Update $\tilde{\mathbf{v}} \leftarrow \tilde{\mathbf{v}} + \mathscr{I}_P\tilde{\tilde{\mathbf{v}}}$.
6. Relax ν_2 times on $\tilde{A}\tilde{\mathbf{u}} = \tilde{\mathbf{f}}$ with initial guess $\tilde{\mathbf{v}}$.
7. Update $\mathbf{v} \leftarrow \mathbf{v} + \mathscr{I}_P\tilde{\mathbf{v}}$.
8. Relax ν_2 times on $A\mathbf{u} = \mathbf{f}$ using initial guess $\mathbf{v}$.

Next to V-cycles there are several other ways of going through the hierarchy, the main ones being W-cycles, F-cycles and FMG (full multigrid) and shown in figure 3.1. Trade-off is speed of doing a single iteration versus the expected number of iterations necessary.

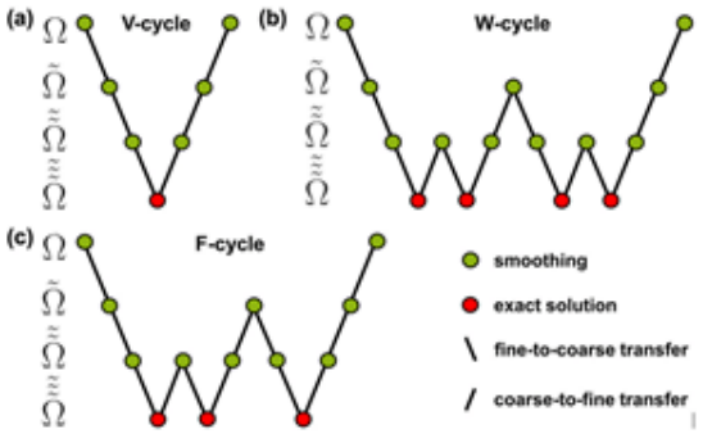


Figure 3.1: Visual description of a V,W and F-cycle scheme using 4 levels.

3.2. h-multigrid

The most common choice for the operator $\tilde{A}$ is the operator corresponding to a coarser discretization. $\tilde{A}$ is the operator resulting from a discretization with a mesh width $2h$ instead of h. Hence why we call it h-multigrid. As a result of the increased mesh width, the matrices and vectors on the coarser level are significantly smaller.
These are its main advantages: it is well known, well researched and we get to work on significantly smaller systems on the coarser levels.

For notation we use A^h, A^{2h}, $\mathbf{v}^h$, $\mathbf{v}^{2h}$, etc. for matrices and vectors on the different grids.
The prolongation operator is denoted $\mathcal{I}_{2h}^h$. This is to be read from the bottom to the top; it transforms vectors over a grid with mesh width $2h$ to vectors over a grid with mesh width h. The restriction operator is denoted $\mathcal{I}_h^{2h}$.
We will show how these operators look like for both 1- and 2-dimensional problems.

3.2.1. Prolongation operator

To prolongate information from the coarse grid to the fine grid within h-multigrid methods, linear interpolation is typically adopted. For a 1-dimensional problem this means $\mathcal{I}_{2h}^h \mathbf{v}^{2h} = \mathbf{v}^h$, where [13]

$$\begin{aligned} v_{2j}^h &= v_j^{2h}, \\ v_{2j+1}^h &= \frac{1}{2}(v_j^{2h} + v_{j+1}^{2h}), \qquad 0 \le j \le \frac{1}{2}n - 1. \end{aligned}$$

In matrix-vector form a small example would look like

$$\mathcal{I}_{2h}^h \mathbf{v}^{2h} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{pmatrix} = \mathbf{v}^h.$$

When expanding to a 2-dimensional problem, we see $\mathcal{I}_{2h}^h \mathbf{v}^{2h} = \mathbf{v}^h$, where [13]

$$\begin{aligned} v_{2i,2j}^h &= v_{ij}^{2h}, \\ v_{2i+1,2j}^h &= \frac{1}{2}(v_{ij}^{2h} + v_{i+1,j}^{2h}), \\ v_{2i,2j+1}^h &= \frac{1}{2}(v_{ij}^{2h} + v_{i,j+1}^{2h}), \\ v_{2i+1,2j+1}^h &= \frac{1}{4}(v_{ij}^{2h} + v_{i+1,j}^{2h} + v_{i,j+1}^{2h} + v_{i+1,j+1}^{2h}), \qquad 0 \le i, j \le \frac{1}{2}n - 1. \end{aligned}$$

3.2.2. Restriction operator

For restriction it may be tempting to simply keep the information on those grid points which are also on the coarser grid, discard the information on those nodes only on the fine grid. This is known as injection: $\mathbf{v}_j^{2h} = \mathbf{v}_{2j}^h$.
But the most common and the better choice is full weighting, where a weighted average is taken of fine grid nodes. For a 1-dimensional problem this means $\mathcal{I}_h^{2h} \mathbf{v}^h = \mathbf{v}^{2h}$, where [13]

$$v_j^{2h} = \frac{1}{4}(v_{2j-1}^h + 2v_{2j}^h + v_{2j+}^h), \qquad 1 \le j \le \frac{1}{2}n - 1.$$

Or in case of a small example in matrix-vector form

$$\mathcal{I}_h^{2h} \mathbf{v}^h = \begin{pmatrix} 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_7 \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \mathbf{v}^{2h}.$$

For a 2-dimensional problem we can write the operator $\mathcal{I}_h^{2h}$ in stencil notation as [14]

$$\mathcal{I}_h^{2h} = \frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}_h^{2h}.$$

With this choice of prolongation and restriction operators the variational condition $\mathcal{I}_{2h}^h = c(\mathcal{I}_h^{2h})^T$ is satisfied.

3.3. p-multigrid

In p-multigrid the coarser level is based on a lower order discretization. That is, the hierarchy is based on a different order p of the basis functions [6]. A vector $\mathbf{v}^p$ contains the coefficients of the order p basis functions to some B-spline curve, surface or solid. $\mathbf{v}^{p-1}$ contains the coefficients to the same curve, surface or solid, but now using the order $p-1$ basis functions.

3.3.1. L_2-projection

This transformation is a bit more complicated than the transformations we have seen for h-multigrid. Here the prolongation and restriction operators are based on a L_2-projection. To demonstrate the change of bases, suppose we have 2 bases $\mathscr{B}_1 = \{\phi_1, ..., \phi_n\}$ and $\mathscr{B}_2 = \{\psi_1, ..., \psi_m\}$. Also suppose we have a function $\sum_i a_i \phi_i$ lying in the span of basis $\mathscr{B}_1$, which we want to rewrite using the basisfunctions ψ_j.
This means we have to find coefficients b_j such that $\frac{1}{2}||\sum_j b_j \psi_j - \sum_i a_i \phi_i||^2_{L^2}$ is minimized [9].

$$\frac{1}{2}||\sum_j b_j \psi_j - \sum_i a_i \phi_i||^2_{L^2} = \frac{1}{2}\int (\sum_j b_j \psi_j - \sum_i a_i \phi_i)^2 \, \mathrm{d}\Omega.$$

For this to be minimized, we need $\frac{d}{db_k} = 0$ for all k.

$$\int (\sum_j b_j \psi_j - \sum_i a_i \phi_i)\psi_k \, \mathrm{d}\Omega = 0 \qquad \forall k,$$

$$\int (\sum_j b_j \psi_j)\psi_k \, \mathrm{d}\Omega = \int (\sum_i a_i \phi_i)\psi_k \, \mathrm{d}\Omega \qquad \forall k,$$

$$\sum_j b_j \int \psi_j \psi_k \, \mathrm{d}\Omega = \sum_i a_i \int \phi_i \psi_j \, \mathrm{d}\Omega \qquad \forall k,$$

$$\left(\int \psi_j \psi_k \, \mathrm{d}\Omega\right)_{j,k} \mathbf{b} = \left(\int \phi_i \psi_k \, \mathrm{d}\Omega\right)_{i,k} \mathbf{a},$$

$$M\mathbf{b} = P\mathbf{a},$$

$$\mathbf{b} = M^{-1}P\mathbf{a}.$$

Note that M is an $n \times n$-matrix, whereas P is an $n \times m$-matrix.

Now to apply this on the different order B-spline basis functions. It has been shown that it is effective to directly project to the level $p = 1$ [5]. Therefore only the operators between level p and level 1 need to be specified.
The prolongation operator is given by

$$\mathscr{I}_1^p = (M_p)^{-1} P_1^p,$$

the restriction operator is given by

$$\mathscr{I}_p^1 = (M_1)^{-1} P_p^1.$$

Here the mass matrices M and transfer matrices P are defined as

$$(M_p)_{i,j} = \int_\Omega \phi_{i,p}\phi_{j,p} \, \mathrm{d}\Omega, \qquad (P_1^p)_{i,j} = \int_\Omega \phi_{i,p}\phi_{j,1} \, \mathrm{d}\Omega,$$

$$(M_1)_{i,j} = \int_\Omega \phi_{i,1}\phi_{j,1} \, \mathrm{d}\Omega, \qquad (P_p^1)_{i,j} = \int_\Omega \phi_{i,1}\phi_{j,p} \, \mathrm{d}\Omega.$$

To avoid having to invert the mass matrices M, its lumped variant $M^L_{i,i} = \sum_j M_{i,j}$ is used. Numerical experiments show this barely effects the convergence behaviour of the p-multigrid method. By the properties (in particular the partition of unity property, see section 2.2) of B-spline basis functions the lumped matrix is easily calculated.

$$M^L_{i,i} = \sum_j M_{i,j} = \sum_j \int_\Omega \phi_{i,p}\phi_{j,p} \, \mathrm{d}\Omega = \int_\Omega \sum_j \phi_{i,p}\phi_{j,p} \, \mathrm{d}\Omega = \int_\Omega \phi_{i,p} \, \mathrm{d}\Omega.$$

3.3.2. Motivation for p-multigrid

On the level $p = 1$ the system will not be significantly smaller as it was when we used h-multigrid. However, the system will be a lot more sparse and will also look a lot more like other better researched systems. This because the level $p = 1$ equals FEM, which is widely known and widely researched.

As our system on the coarse grid is very similar to a system resulting from a FEM discretization, we can use methods which are known to perform well for FEM. This means we can use something simple like h-multigrid and Gauss Seidel.
That is, after we have gone from level $p = p$ to level $p = 1$, we can go and do h-multigrid on the level $p = 1$ [6]. This scheme is visualized in figure 3.2.

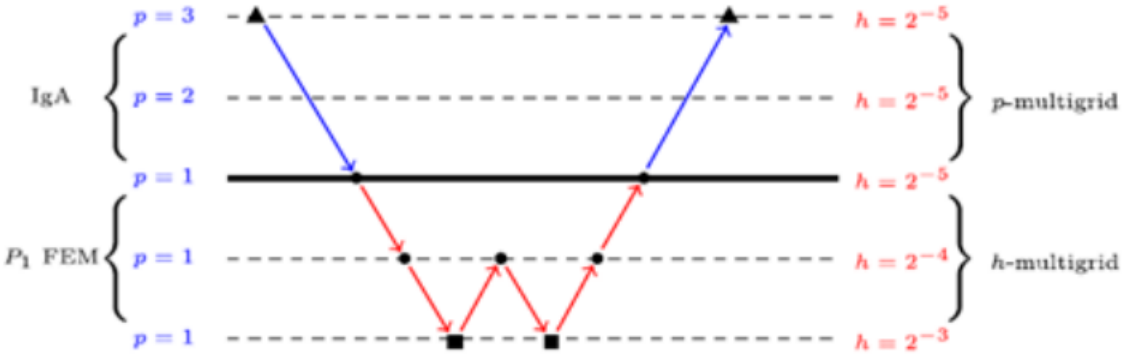


Figure 3.2: Scheme for combined use of p- and h-multigrid. From: [6]

3.4. Smoothers

Within Multigrid methods, a basic iterative method is typically used as a smoother. However, in IgA the performance of classical smoothers such as (damped) Jacobi or Gauss Seidel decreases significantly for higher values of p [15]. Therefore other methods are suggested, such as a smoother using an Incomplete LU factorization. This is the approach we choose for the remainder of this report, but this is by far the only option.

3.4.1. Alternative smoothers

As mentioned we decided to focus on the ILU type smoothers in this report, however other methods have also been suggested. Recent research includes the use of multiplicative Schwarz smoothers [16], subspace corrected mass smoothers [17], hybrid smoothers [18] and symbol based multigrid smoothers [19].
These aren't further elaborated upon in this report.

Chapter 4

ILU type smoothers - Part I

4.1. Incomplete LU factorizations and smoothers

The goal of an incomplete LU factorization is to find sparse lower- and upper triangular matrices L, U such that $A \approx LU$. This incomplete factorization is significantly cheaper compared to full LU factorization, both for computation and storage [4].
Various strategies exist for computing an incomplete LU factorization. Of these we will show two in this chapter (ILU(0) and ILUT) and others in upcoming chapters (Block ILUT, etc.).

Once the factorization is obtained, a single smoothing step is applied as follows [4]:
Calculate error: $\mathbf{e} = U^{-1}L^{-1}(\mathbf{f} - A\mathbf{u})$.
Update guess: $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{e}$.

We will see that these ILU type smoothers will cost significant setup time, however one should need only one or few smoothing steps.

4.1.1. ILU(0)

First incomplete factorization we look at is ILU(0) [20],[21]. It is the simplest incomplete factorization there is and is based on the ikj-version of Gaussian elimination. It only allows for nonzero entries on L and U on those positions A contains nonzero entries. Any element in L and U that falls outside the sparsity pattern of A is dropped.
We also call this zero fill in. The algorithm is as follows.

ALGORITHM 4.1: ILU(0)

Goal: Determine sparse lower and upper triangular L, U such that $LU \approx A$.
1. Start with $L = I$, $U =$ copy of A.
2. for $i = 2, ..., n$:
3. for $k = 1, ..., i-1$ and if $(i,k) \in NZ(A)$:
4. Compute pivot $l_{ik} = u_{ik}/u_{kk}$, set $u_{ik} = 0$.
5. for $j = k+1, ..., n$ and if $(i,j) \in NZ(A)$:
6. Compute $u_{ij} = u_{ij} - l_{ik}u_{kj}$.
7. end for
8. end for
9. end for

Here $NZ(A)$ denotes those points where the matrix A contains nonzero entries.

4.1.2. ILUT

The accuracy of the ILU(0) incomplete factorization may be insufficient to yield an adequate rate of convergence. Allowing for more fill in, that is more nonzero entries, may give better approximations of L and U. Methods that allow for more fill in can be more efficient as well as more reliable.
One such method is ILUT, a factorization where thresholds are used to determine which elements are kept [20],[21]. Similar to ILU(0), except the condition $(i,j) \in NZ(A)$ is replaced by another dropping rule, which is explained below the algorithm.

ALGORITHM 4.2: ILUT

Goal: Determine sparse lower and upper triangular L, U such that $LU \approx A$.
1. Start with L=I, U=copy of A.

2. for $i = 1, ..., n$:
3. $w := a_{i\star}$
4. for $k = 1, ..., i-1$ and if $w_k \neq 0$:
5. $w_k = w_k / a_{kk}$
6. Apply a dropping rule to w_k.
7. if $w_k \neq 0$:
8. Update $w := w - w_k u_{k\star}$.
9. end if
10. end for
11. Apply a dropping rule to row w.
12. $l_{ij} = w_j$ for $j = 1, ..., i-1$.
13. $u_{ij} = w_j$ for $j = i, ..., n$.
14. Set $w = 0$.
15. end for

In the factorization ILUT(m, τ) the following rules are used [4]:
1. In line 6, an element w_k is dropped (i.e. replaced by zero) if it is less then the relative tolerance τ_i, obtained by multiplying τ by the original norm of the i-th row.
2. In line 11, drop again any element in the row with a magnitude below τ_i. Then, only keep the M ($M = m \cdot$ the average number of nonzeros per row) largest elements in both the L part and the U part of the row. The diagonal element is also always kept.
Here $a_{i\star}$ denotes the i-th row of matrix A.

In this project we typically use $m = 1$, $\tau = 10^{-13}$.

4.1.3. Computational Costs

In table 4.1 we see the cost of factorization and smoothing using ILU(0) and ILUT compared to the cost of a full factorization or a single step or Gauss Seidel.

Full LU factorization	$\mathcal{O}(N_{\text{dof}}^3)$
Forward- and back substitution	$\mathcal{O}(N_{\text{dof}}^2)$
Single step Gauss Seidel	$\mathcal{O}(N_{\text{dof}}^2)$
Setup ILU(0)	$\mathcal{O}(N_{\text{dof}} p^{2d})$
Apply ILU(0)	$\mathcal{O}(N_{\text{dof}} p^{d})$
Setup ILUT	$\mathcal{O}(N_{\text{dof}} p^{2d})$
Apply ILUT	$\mathcal{O}(N_{\text{dof}} p^{d})$

Table 4.1: Complexity of ILU(0) and ILUT smoothers.

To explain the complexities of the ILU type smoothers keep in mind the structure of the matrix A. We know that matrices resulting from the IgA discretizations have a limited number of nonzero entries. This amount is dependent on the order of basis functions p and the dimension d. In fact, the number of nonzero entries per row or column is at most $(1+2p)^d$. Or easier $\mathcal{O}(p^d)$.

Using this we can verify the complexities for the ILU(0) smoother. On line 2 we loop over $\mathcal{O}(N_{\text{dof}})$ elements, on line 3 over $\mathcal{O}(p^d)$ elements and finally on line 5 over $\mathcal{O}(p^d)$ elements. The operations are simple scalar operations, hence a cost of $\mathcal{O}(1)$. Multiplying N_{dof} with p^d, p^d and 1 gives a complexity of $\mathcal{O}(N_{\text{dof}} p^{2d})$ for the incomplete factorization.
For applying the smoother we need to do forward- and back substitution over $\mathcal{O}(N_{\text{dof}})$ rows with $\mathcal{O}(p^d)$ nonzeros per row. Hence this costs $\mathcal{O}(N_{\text{dof}} p^d)$.

Using similar reasoning for the ILUT(m, τ) smoother. On line 2 we loop over $\mathcal{O}(N_{\text{dof}})$ elements, on line 4 over $\mathcal{O}(p^d)$ elements. All operations are at most $\mathcal{O}(p^d)$, because the number of nonzeros in w or rows or columns of U and L is always bounded. Hence a complexity of $\mathcal{O}(N_{\text{dof}} p^{2d})$ for the factorization.
And a complexity of $\mathcal{O}(N_{\text{dof}} p^d)$ for the smoothing step, for the same reason as seen before with ILU(0).

It should be noted that though we see the same complexities here for both ILU(0) and ILUT, that an ILUT factorization will contain more nonzero entries. And thus will be slightly more expensive both for computation and storage.

4.1.4. Numerical Results

To show how the different smoothers perform consider the following test problem. The benchmark chosen is a Laplace equaiton on the unit square. All system matrices, transfer matrices, RHS vectors, etc. have been generated through my own programming using Python. The code for factorization and smoothing is also self made.

BENCHMARK: Laplace equation on the unit square

Consider the boundary value problem

$$-\Delta u = f \text{ on } [0,1]^2,\ u = 0 \text{ on the boundary.}$$

f is chosen as $f = 2\pi^2 \sin(\pi x)\sin(\pi y)$ such that the solution is known to be $u = \sin(\pi x)\sin(\pi y)$.

We will use IgA discretizations (single patch) for different values of p and h. Then we will use a two level p-multigrid method with on the fine grid either a GS, ILU(0) or ILUT smoother and a direct solve on $p = 1$.

We are mostly interested in how many steps it takes for a method to converge. For this we use the stopping criterium $||\mathbf{r}^k||/||\mathbf{r}^0|| < 10^{-8}$, where $\mathbf{r}^k$ and $\mathbf{r}^0$ denote the residuals after k and 0 steps respectively.
Tables 4.2 till 4.4 show the number of steps till convergence in this particular setup.

	$p=2$	$p=3$	$p=4$	$p=5$
$h=2^{-3}$	8	22	84	347
$h=2^{-4}$	7	23	68	254
$h=2^{-5}$	5	21	63	204

Table 4.2: Steps till convergence (GS smoother, $\nu_1 = \nu_2 = 1$)

	$p=2$	$p=3$	$p=4$	$p=5$
$h=2^{-3}$	3	2	2	2
$h=2^{-4}$	3	3	3	3
$h=2^{-5}$	3	3	3	3

Table 4.3: Steps till convergence (ILU(0) smoother)

	$p=2$	$p=3$	$p=4$	$p=5$
$h=2^{-3}$	1	1	1	1
$h=2^{-4}$	2	2	1	1
$h=2^{-5}$	2	2	2	2

Table 4.4: Steps till convergence (ILUT($m = 1, \tau = 10^{-13}$) smoother)

Table 4.2 shows the results for using Gauss Seidel as a smoother. As was mentioned already in the introduction the performance drops significantly if we are using higher order basis functions.
Tables 4.3 and 4.4 show the same tests but for the ILU(0) and ILUT smoother respectively. For the ILUT smoother the parameters $m = 1$ and $\tau = 10^{-13}$ are used. Here the amount of steps required to hit the stopping criterium does not seem to be dependent on the order of basis functions p.
We also see that there are less steps needed if using ILUT compared to ILU(0). Therefore we will continue elaborating on the ILUT smoother in the following sections.

4.2. Block ILUT

4.2.1. Matrix structure for multipatch geometries

In previous sections we have seen that when using multipatch geometries the system matrix A will have the following block structure.

$$A = \begin{pmatrix} A_{11} & & & A_{1\Gamma} \\ & \ddots & & \vdots \\ & & A_{KK} & A_{K\Gamma} \\ A_{\Gamma 1} & \cdots & A_{\Gamma k} & A_{\Gamma\Gamma} \end{pmatrix}.$$

Given this block structure we can tell how the LU factorization of A should look like. Through block matrix multiplication it can be verified that

$$A = \begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ B_1 & \cdots & B_K & I \end{pmatrix} \begin{pmatrix} U_1 & & & C_1 \\ & \ddots & & \vdots \\ & & U_K & C_K \\ & & & D \end{pmatrix} = LU.$$

Here L_i, U_i are based on a complete LU factorization $A_{ii} = L_i U_i$.
The B_i, C_i and D are given by

$$B_i = A_{i\Gamma} U_i^{-1},\ C_i = L_i^{-1} A_{\Gamma i},\ D = A_{\Gamma\Gamma} - \sum_{i=1}^{K} B_i C_i.$$

Here D is not a diagonal submatrix as the symbol might suggest. It is not even upper- or lower triangular. It is simply the first letter available after the A_{ii}, B_i and C_i.

4.2.2. Block ILUT factorization

This information gives for a new method of approximating the LU factorization for A.
The L_i and U_i are approximated by calculating the ILUT factorization on the blocks A_{ii}. These approximations are denoted $\tilde{L}_i$ and $\tilde{U}_i$. Calculations of the other blocks are then based on the $\tilde{L}_i$, $\tilde{U}_i$.

$$A \approx \begin{pmatrix} \tilde{L}_1 & & & \\ & \ddots & & \\ & & \tilde{L}_K & \\ \tilde{B}_1 & \cdots & \tilde{B}_K & I \end{pmatrix} \begin{pmatrix} \tilde{U}_1 & & & \tilde{C}_1 \\ & \ddots & & \vdots \\ & & \tilde{U}_K & \tilde{C}_K \\ & & & \tilde{D} \end{pmatrix} = \tilde{L}\tilde{U}.$$

$$\tilde{B}_i = A_{i\Gamma} \tilde{U}_i^{-1},\ \tilde{C}_i = \tilde{L}_i^{-1} A_{\Gamma i},\ \tilde{D} = A_{\Gamma\Gamma} - \sum_{i=1}^{K} \tilde{B}_i \tilde{C}_i.$$

Here we used tildes to emphasize we are talking about approximations. In the remainder of this text the tildes are occasionally dropped to avoid clutter in notation. Every time we use one of the symbols L_i, U_i, B_i, C_i, D we do indeed mean the approximation.
From this moment we will refer to this factorization as Block ILUT, applying ILUT on the full matrix A will from this point onward be called Global ILUT.
The factorization is also presented in pseudo-code in algorithm 4.3. Note that big parts of the factorization can be performed in parallel. Lines 2, 5 and 6 could have been combined in a single for-loop, however we decided to show it this way as the method for calculating L_i, U_i significantly differs from the method for calculating B_i, C_i.

ALGORITHM 4.3: Block ILUT factorization

Goal: Determine sparse lower and upper triangular L, U such that $LU \approx A$.
1. for $i = 1, ..., K$ (in parallel):
2. $\quad L_i, U_i = \text{ILUT}(A_{ii})$.
3. end for
4. for $i = 1, ..., K$ (in parallel):
5. $\quad$ Solve for B_i in $U_i^T B_i^T = A_{i\Gamma}^T$.
6. $\quad$ Solve for C_i in $L_i C_i = A_{\Gamma i}$.
7. end for
8. $D = A_{\Gamma\Gamma} - \sum_{i=1}^{K} \tilde{B}_i \tilde{C}_i$.

4.2.3. Using Block ILUT as a smoother

Also in applying the smoother a lot can be parallelized. Remember that $\mathbf{u}$ in $A\mathbf{u} = \mathbf{f}$ is approximated by first solving $L\mathbf{y} = \mathbf{f}$, then solving $U\mathbf{u} = \mathbf{y}$.

$$\begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ B_1 & \cdots & B_K & I \end{pmatrix} \begin{pmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_K \\ \mathbf{y}_\Gamma \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_K \\ \mathbf{f}_\Gamma \end{pmatrix}, \qquad \begin{pmatrix} U_1 & & & C_1 \\ & \ddots & & \vdots \\ & & U_K & C_K \\ & & & D \end{pmatrix} \begin{pmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_K \\ \mathbf{u}_\Gamma \end{pmatrix} = \begin{pmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_K \\ \mathbf{y}_\Gamma \end{pmatrix}.$$

If we write it out we can see it is possible to determine $\mathbf{y}_1, ..., \mathbf{y}_K$ in parallel using forward substitution on each of the blocks. After this we can determine $\mathbf{y}_\Gamma$, again by forward substitution.
In the second system we first have to determine $\mathbf{u}_\Gamma$ through solving $D\mathbf{u}_\Gamma = \mathbf{y}_\Gamma$. D is not triangular, thus requiring an expensive exact solve or a further approximation. Once $\mathbf{u}_\Gamma$ is obtained $\mathbf{u}_1, ...\mathbf{u}_\Gamma$ can be determined in parallel using backward substitution on each of the blocks.

ALGORITHM 4.4: Smoothing using Block ILUT factorization

Goal: Update guess $\mathbf{u}$.
1. for $i = 1, ..., K$ (in parallel):
2. Solve for $\mathbf{y}_i$ in $L_i\mathbf{y}_i = \mathbf{f}_i$ using forward substitution.
3. end for
4. Solve for $\mathbf{y}_\Gamma$ in $L\mathbf{y} = \mathbf{f}$ using forward substitution.
5. Solve for $\mathbf{u}_\Gamma$ in $D\mathbf{u}_\Gamma = \mathbf{y}_\Gamma$ using ... .
6. for $i = 1, ..., K$ (in parallel):
7. Solve for $\mathbf{u}_i$ in $U_i\mathbf{u}_i = \mathbf{y}_i$ using backward substitution.
8. end for

4.2.4. Computational Costs

In table 4.5 we see the computational cost for different parts of the Block ILUT smoother as well as a comparison to the cost of the global smoother.
N_{patch} denotes the degrees of freedom within a single patch, i.e. the size of one block corresponding to one patch. $N_{\text{interface}}$ denotes the degrees of freedom at the interface, i.e. the size of the right column and bottom row. These relate to N_{dof} via $N_{\text{dof}} = KN_{\text{patch}} + N_{\text{interface}}$.

Global ILUT (setup)	$\mathcal{O}(N_{\text{dof}}p^{2d})$
Global ILUT (apply)	$\mathcal{O}(N_{\text{dof}}p^{d})$
Block ILUT (setup L_i, U_i)	$\mathcal{O}(N_{\text{patch}}p^{2d})$
Block ILUT (setup B_i, C_i)	$\mathcal{O}(N_{\text{patch}}p^{d})$
Block ILUT (setup D)	$\mathcal{O}(N_{\text{interface}}^3)$
Block ILUT (apply)	$\mathcal{O}(N_{\text{patch}}p^{d})^*$

Table 4.5: Complexity of Block ILUT smoother.

Doing an ILUT factorization on a submatrix A_{ii} is of course cheaper than on the full matrix A. It is easy to see that computing a pair L_i, U_i can be done at a cost of $\mathcal{O}(N_{\text{patch}}p^{2d})$, compared to the $\mathcal{O}(N_{\text{dof}}p^{2d})$ for the full matrix.
The blocks B_i, C_i are determined through solving the systems $U_i^T B_i^T = A_{i\Gamma}^T$, $L_i C_i = A_{\Gamma i}$. These only require forward substitution and can therefore be performed at the cost of $\mathcal{O}(N_{\text{patch}}p^{d})$ per patch.
Computing D requires matrix multiplication. In the worst case scenario (for a full matrix D) this will cost $\mathcal{O}(N_{\text{interface}}^3)$.

Next to considering how expensive each step is, also consider what could be parallelized.
Determining L_i, U_i is independent of determining L_j, U_j and can be performed in parallel.
Determining B_i is independent of determining C_i, which is independent of determining B_j, C_j. Again can be performed in parallel. For determining D the matrix multiplications can be performed in parallel.

Also in applying the smoother a lot can be parallelized. Just before we have seen how $L\mathbf{y} = \mathbf{f}$ and $U\mathbf{u} = \mathbf{y}$ look when

written out.

$$\begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ B_1 & \cdots & B_K & I \end{pmatrix} \begin{pmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_K \\ \mathbf{y}_\Gamma \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_K \\ \mathbf{f}_\Gamma \end{pmatrix}, \qquad \begin{pmatrix} U_1 & & & C_1 \\ & \ddots & & \vdots \\ & & U_K & C_K \\ & & & D \end{pmatrix} \begin{pmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_K \\ \mathbf{u}_\Gamma \end{pmatrix} = \begin{pmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_K \\ \mathbf{y}_\Gamma \end{pmatrix}.$$

We can determine $\mathbf{y}_1, ..., \mathbf{y}_K$ in parallel using forward substitution on each of the blocks. This at the cost of $\mathcal{O}(N_{\text{patch}} p^d)$. The cost of determining $\mathbf{y}_\Gamma$, again by forward substitution, is dependent on the amount of nonzeros in $B_i, ..., B_K$.
In the second system we first have to determine $\mathbf{u}_\Gamma$ through solving $D\mathbf{u}_\Gamma = \mathbf{y}_\Gamma$. D is not triangular, either making this quite expensive at a cost of $\mathcal{O}(N^3_{\text{interface}})$ for an exact solve, or using a further approximation. Once $\mathbf{u}_\Gamma$ is obtained $\mathbf{u}_1, ...\mathbf{u}_\Gamma$ can be determined in parallel using backward substitution on each of the blocks. This at a cost of $\mathcal{O}(N_{\text{patch}} p^d)$.

Hence the asterix in table 4.5. Applying the smoother costs roughly $\mathcal{O}(N_{\text{patch}} p^d)$ if applied in parallel and the bottom row is small. Which is the case if using a smaller amount of patches, but certainly not the case for a large amount of patches.

4.2.5. Numerical Results

To show the performance of Block ILUT we need a new benchmark. Block ILUT shines when used on a discretization based on a multipatch geometry. In fact, I do not know what Block ILUT on a single patch geometry would look like. And the previous benchmark used only a single patch.

The benchmark chosen is a CDR equation on a unit square and is equal to a benchmark used in [3]. The system matrices and the necessary transfer matrices were provided by supervisor Ir. Roel Tielen. The code for factorization and smoothing was written by myself, using Python.
This benchmark is the benchmark used for the remainder of this text.

BENCHMARK: CDR equation on multipatch geometry

Consider the convection-diffusion-reaction (CDR) equation

$$-\nabla \cdot (D\nabla u) + \mathbf{v} \cdot \nabla u + Ru = f, \text{ on } \Omega,$$

$$\text{where } \Omega = [0,1]^2,\ D = \begin{pmatrix} 1.2 & -0.7 \\ -0.4 & 0.9 \end{pmatrix},\ \mathbf{v} = \begin{pmatrix} 0.4 \\ -0.2 \end{pmatrix},\ R = 0.3.$$

The right hand side vector f is chosen such that the exact solution u to this problem is given by

$$u(x,y) = \sin(\pi x)\sin(\pi y).$$

Next to varying h and p we also vary the amount of patches. We consider $K = 4$, $K = 16$ and $K = 64$.

In table 4.6 we see the steps required till convergence ($||\mathbf{r}^k||/||\mathbf{r}^0|| < 10^{-8}$) for this particular setup.
Also the number of nonzero elements in the factorization are provided, displayed as a fraction of the number of nonzero elements of A. That is nz(ILUT)/nz(A). We see an increase for the number of nonzeros from the system matrix A, to $L+U$ for a global ILUT factorization, to $L+U$ for a block ILUT factorization. But this increase is manageable.

From this table it is clear that the Block ILUT smoother performs remarkably well. The steps till convergence are very low, often the method converges in a single step.

Therefore we chose to use Block ILUT as a base for our research in the next chapter. The convergence is very fast as seen in this table. There are however some expensive steps involved, both in the factorization stage as well as the smoothing stage.

			steps Global ILUT	steps Block ILUT		nz A	nz Global ILUT	nz Block ILUT
K=4	p=2	h=3	3	2		7569	1.81	2.59
		h=4	2	2		27889	1.98	3.53
	p=3	h=3	2	1		16641	1.66	2.07
		h=4	2	2		58081	1.93	2.78
	p=4	h=3	2	1		30625	1.52	1.77
		h=4	2	1		101761	1.86	2.39
	p=5	h=3	2	1		50625	1.42	1.61
		h=4	2	1		160801	1.81	2.17
K=16	p=2	h=2	3	1		8649	1.51	1.84
		h=3	3	1		29929	1.79	2.93
	p=3	h=2	3	1		21025	1.37	1.51
		h=3	2	1		66049	1.68	2.23
	p=4	h=2	2	1		42025	1.26	1.36
		h=3	2	1		121801	1.57	1.87
K=64	p=2	h=1	3	1		11025	1.45	1.31
		h=2	3	1		34225	1.57	1.91
	p=3	h=1	3	1		31329	1.29	1.18
		h=2	3	1		83521	1.41	1.55
	p=4	h=1	3	1		70225	1.20	1.12
		h=2	3	1		167281	1.31	1.38

Table 4.6: Steps till convergence for the Block ILUT smoother compared to the Global ILUT smoother

Chapter 5

ILU type smoothers - Part II

5.1. Motivation for continued research

In chapter 4 the Block ILUT smoother was introduced, utilizing the block structure of the system matrix A and therefore knowing how its LU factorization should look like. The following approximation is used.

$$A \approx \begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ B_1 & \cdots & B_K & I \end{pmatrix} \begin{pmatrix} U_1 & & & C_1 \\ & \ddots & & \vdots \\ & & U_K & C_K \\ & & & D \end{pmatrix} = LU.$$

$$B_i = A_{i\Gamma} U_i^{-1},\ C_i = L_i^{-1} A_{\Gamma i},\ \tilde{S} = A_{\Gamma\Gamma} - \sum_{i=1}^{K} B_i C_i.$$

The Block ILUT smoother performed very well in our IgA context. The best smoother so far. But it also looked like there still is something to improve. As seen in section 4.2.4 the $L_1, U_1, ..., L_K, U_K$ are efficiently approximated, but especially D drives up the cost, both for factorization and smoothing.

This leads to the question: Can we find a factorization for A which does not require to calculate D (and possibly the B_i, C_i), or where these can be approximated using a different approach?

There are multiple approaches to tackle this, different fields to take inspiration from. From these we discuss two in this chapter, others are left for the section 'further research'.
A first obvious approach is to simply remove the off-diagonal entries. This is attempted in section 5.2. Unfortunately too much information is lost doing this, this does not give for a good method

In sections 5.3 to 5.6 we investigate a method which does show a lot more promise. Inspiration is taken from the work of Chow and Anzt ([22],[23]), which describe methods of construction of L and U via a fixed-point method.
This is of particular interest as it suggests we can calculate the 'easy' blocks via Block ILUT. Calculate the 'hard' blocks using this iterative process, only needing to update that part of the matrix. It also produces a factorization in the form below as seen below, where E and F are actually lower and upper triangular.

$$A \approx \begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ B_1 & \cdots & B_K & E \end{pmatrix} \begin{pmatrix} U_1 & & & C_1 \\ & \ddots & & \vdots \\ & & U_K & C_K \\ & & & F \end{pmatrix} = LU.$$

Next to this, the process is also parallelizable, making it even more attractive.

5.2. Removing Off-Diagonal Entries

The idea of this section is to attempt factorizations where we simply discard parts of the information. Certain blocks of the factorization are too difficult or costly to compute, that we rather do not do this.
We are aware that this will probably significantly impact the accuracy of the factorization, however it will also decrease the cost of the factorization by such an amount (as well as the cost per smoothing step) that we can accept quite an increase of multigrid iterations.

For comparison, remind the factorization following from Block ILUT:

$$\begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ B_1 & \cdots & B_K & I \end{pmatrix}\begin{pmatrix} U_1 & & & C_1 \\ & \ddots & & \vdots \\ & & U_K & C_K \\ & & & D \end{pmatrix} \approx \begin{pmatrix} A_{11} & & & A_{1\Gamma} \\ & \ddots & & \vdots \\ & & A_{KK} & A_{K\Gamma} \\ A_{\Gamma 1} & \cdots & A_{\Gamma k} & A_{\Gamma\Gamma} \end{pmatrix}.$$

An obvious approach would be the following factorization. It focuses entirely on the interior of the patches, the interfaces are completely ignored.

$$\begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ & & & I \end{pmatrix}\begin{pmatrix} U_1 & & & \\ & \ddots & & \\ & & U_K & \\ & & & I \end{pmatrix} = \begin{pmatrix} \tilde{A}_{11} & & & \\ & \ddots & & \\ & & \tilde{A}_{KK} & \\ & & & I \end{pmatrix}.$$

This factorization could be computed at a cost of $\mathcal{O}(N_{\text{patch}} p^{2d})$ and the smoother could be performed at a cost of $\mathcal{O}(N_{\text{patch}} p^{d})$. The reason for suggesting this factorization as a smoother was the hope that it could fix errors on the interior of the different patches, while keeping errors on the boundaries the same.
Unfortunately though, it turns out this is not what happens. The method diverges if you choose to use this as a smoother as can be seen in table 5.1 under the column 'Diag1'. Too much information is lost.

			ILU(0)	Global ILUT	Block ILUT	Diag1	Diag2	Diag3
K=4	p=2	h=3	4	3	2	89	-	-
		h=4	3	2	2	20	-	-
	p=3	h=3	3	2	1	-	-	-
		h=4	3	2	2	-	-	-
	p=4	h=3	4	2	1	-	-	-
		h=4	3	2	1	-	-	-
	p=5	h=3	4	2	1	-	-	-
		h=4	4	2	1	-	-	-
K=16	p=2	h=2	4	3	1	-	-	-
		h=3	4	3	1	-	-	-
	p=3	h=2	4	3	1	-	-	-
		h=3	3	2	1	-	-	-
	p=4	h=2	4	2	1	-	-	-
		h=3	4	2	1	-	-	-
K=64	p=2	h=1	5	3	1	-	-	-
		h=2	4	3	1	-	-	-
	p=3	h=1	6	3	1	-	-	-
		h=2	4	3	1	-	-	-
	p=4	h=1	6	3	1	-	-	-
		h=2	4	3	1	-	-	-

Table 5.1: Steps till convergence for several diagonal block limited smoothers.

Altogether, in table 5.1 we see only two instances where the method converges 'by accident' as the dashes imply that the method did not converge within 200 iterations and therefore the method is stopped. At the very best one could say the method does not work reliably. The more accurate description would be that it does not work.

Two other factorizations have been attempted that seemed reasonable. For 'Diag2' we chose:

$$\begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ B_1 & \cdots & B_K & I \end{pmatrix}\begin{pmatrix} U_1 & & & C_1 \\ & \ddots & & \vdots \\ & & U_K & C_K \\ & & & I \end{pmatrix} \approx \begin{pmatrix} A_{11} & & & A_{1\Gamma} \\ & \ddots & & \vdots \\ & & A_{KK} & A_{K\Gamma} \\ A_{\Gamma 1} & \cdots & A_{\Gamma k} & \sum B_i C_i \end{pmatrix}.$$

This does not require to compute D and the rest of the factorization matches. However, in the bottom right we end with quite a big submatrix that is far removed from $A_{\Gamma\Gamma}$. And thus not helping in reducing the error when applied as a smoother in the multigrid method.

Finally, for 'Diag3' we attempted the following:

$$\begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ & & & L_\Gamma \end{pmatrix}\begin{pmatrix} U_1 & & & \\ & \ddots & & \\ & & U_K & \\ & & & U_\Gamma \end{pmatrix} = \begin{pmatrix} A_{11} & & & \\ & \ddots & & \\ & & A_{KK} & \\ & & & A_{\Gamma\Gamma} \end{pmatrix}.$$

where $L_\Gamma U_\Gamma$ an incomplete LU factorization of $A_{\Gamma\Gamma}$. Also this method has no convergence when applied as a smoother in the multigrid method.

We can conclude that this approach is unsuccessful. It was very easy to try whether this could be a possibility, but unfortunately it is too much of a simplification.

5.3. Fixed-point ILU

5.3.1. Algorithm

All ILU type factorizations we have discussed so far are based on a Gaussian elimination process. But not this one, the fixed-point ILU algorithm is an iterative process based on the property

$$(LU)_{ij} = a_{ij},\ (i,j) \in S.$$

where S is the desired sparsity pattern of the ILU factorization. Instead of using a Gaussian elimination process, it is a problem of computing unknowns l_{ij} and u_{ij} using the above property as constraints. The unknowns to be computed are

$$l_{ij},\ i > j,\ (i,j) \in S, \qquad u_{ij},\ i \leq j,\ (i,j) \in S.$$

The total number of unknowns is $|S|$, the number of elements in the sparsity pattern S, as we can choose L to have a unit diagonal.

To determine these unknowns we rewrite the constraint. Using that L and U are lower- and upper triangular respectively we can write

$$a_{ij} = \sum_{k=1}^{N_{\text{dof}}} l_{ik}u_{kj} = \sum_{k=1}^{\min(i,j)} l_{ik}u_{kj},\ (i,j) \in S$$

As every constraint is associated with an element of S, we have $|S|$ equations to go with the $|S|$ unknowns. The equation for (i,j) gives an explicit formula for l_{ij} (if $i > j$) or u_{ij} (if $i \leq j$).

$$l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right),$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$$

Note that these equations are in the form $\mathbf{x} = G(\mathbf{x})$, where x is a vector containing the unknowns l_{ij} and u_{ij}. It is now natural to try solve these equations via a fixed-point iteration.

$$\mathbf{x}^{(p+1)} = G(\mathbf{x}^{(p)})$$

This leads to the following algorithm.

ALGORITHM 5.1: Fixed-point ILU

Goal: Determine sparse lower and upper triangular L, U such that $LU \approx A$.
1. Start with initial guess for L and U
2. for $sweep = 1, 2, \ldots$ until convergence:
3. for $(i,j) \in S$ (in parallel):
4. if $i > j$:
5. $l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right) / u_{jj}$
6. else:
7. $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
8. end if
9. end for
10. end for

An obvious choice for an initial guess for L and U would be to simply take the lower- and upper triangular parts of A. And let $L^{(0)}$ have unit diagonal

$$l_{ij}^{(0)} = a_{ij},\ i > j,\ (i,j) \in S$$

$$u_{ij}^{(0)} = a_{ij},\ i \leq j,\ (i,j) \in S$$

An obvious choice for S would be the sparsity pattern of A.

5.3.2. Remarks

There is a lot of structure in the function fixed-point function G. When the unknowns l_{ij} and u_{ij} are viewed as entries of matrices L and U, the formula for unknown (i, j) depends only on other unknowns in L and U that are to its left or above it. This is depicted in figure 5.1, where the L and U factors are shown superimposed into one matrix.

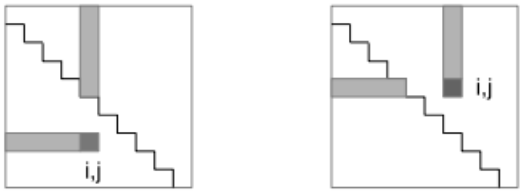


Figure 5.1: Formula for unknown at (i, j) (dark gray) depends on other unknowns left of (i, j) in L and above (i, j) in U (shaded). From: [22]

Therefore, different ways of performing the fixed-point iteration can give different methods. For example, if you decide to compute all components of $\mathbf{x}^{(k+1)}$ in sequence (and left to right and top to bottom). If while doing this you always use the latest values of $\mathbf{x}$, then we solve the equations in a single sweep. Also, the solution process corresponds exactly to performing a conventional ILU factorization.

This behavior is shown in table 5.2. Here the norm of the ILU residual $(A - LU)$ is given for a test matrix A ($K = 4, p = 2, h = 3$), with S the sparsity pattern of A and after different number of iterations and different ways of performing the fixed-point iteration. Fixed-point 1 is to always use $\mathbf{x}^{(k+1)}$ when it is known, to perform everything in sequence. Fixed-point 2 will always use the old $\mathbf{x}^{(k)}$. In Fixed-point 3 we pretend we are working in parallel using four cores.
This is compared to the ILU residual of the ILU(0) method seen in chapter 4.

	1	2	3	4	5	6	7	8	9	10
ILU(0)	2.1064									
Fixed-point 1	2.1064	2.1064	2.1064	2.1064	2.1064	2.1064	2.1064	2.1064	2.1064	2.1064
Fixed-point 2	$2 \cdot 10^2$	$3 \cdot 10^3$	$1 \cdot 10^4$	$4 \cdot 10^5$	$8 \cdot 10^6$	$1 \cdot 10^7$	$1 \cdot 10^8$	$1 \cdot 10^9$	$2 \cdot 10^{10}$	$2 \cdot 10^{11}$
Fixed-point 3	13.049	2.1063	2.1064	2.1064	2.1064	2.1064	2.1064	2.1064	2.1064	2.1064

Table 5.2: ILU residuals after different amount of sweeps.

We see that Fixed-point 1 gives the same factorization as ILU(0). This is to be expected, given that the chosen sparsity pattern equals that of A and thus the sparsity pattern of the ILU(0) factorization. Fixed-point 3 converges to the same factorization, very quickly as well. Fixed-point 2 diverges.

There are other factors that could impact the method. For example, it has been shown in [22] that it is beneficial to diagonally scale the matrix A. That is, instead of working on A to work on a matrix TAT, where T a diagonal matrix such that TAT has unit diagonal. This has shown to improve convergence.
To illustrate this, table 5.3 shows the results for the same test as before, but now with a diagonally scaled matrix. Now we also see convergence for Fixed-point 2.

	1	2	3	4	5	6	7	8	9	10
ILU(0)	0.7757									
Fixed-point 1	0.7757	0.7757	0.7757	0.7757	0.7757	0.7757	0.7757	0.7757	0.7757	0.7757
Fixed-point 2	1.4283	0.9371	0.7932	0.7813	0.7753	0.7759	0.7756	0.7757	0.7757	0.7757
Fixed-point 3	0.9039	0.7755	0.7757	0.7757	0.7757	0.7757	0.7757	0.7757	0.7757	0.7757

Table 5.3: ILU residuals after different amount of sweeps (diagonally scaled).

5.3.3. Computational Cost

From algorithm 5.1 it is clear that the cost of a single sweep is dependent on the size of sparsity pattern S (the loop on line 3) and the number of nonzeros per row or column (the summations on line 5 and 7).

If we were to apply this on our IgA discretization, where for S we take the sparsity pattern of A, this would have a computational cost of $\mathcal{O}(N_{\text{dof}}p^{2d})$. The size of S is $\mathcal{O}(N_{\text{dof}}p^d)$, as A has N_{dof} rows with $\mathcal{O}(p^d)$ nonzeros per row. The maximum length of the summations in line 5 and 7 is thus also $\mathcal{O}(p^d)$. Giving a total of $\mathcal{O}(N_{\text{dof}}p^{2d})$ per iteration.
The total is the same as for example ILU(0) or the global ILUT method.

5.4. Block Fixed-point ILU

5.4.1. General info

Sections 5.3 discussed Fixed-point ILU by itself and showed results where Fixed-point ILU is performed on the full matrix. However, we do not aim to replace the Block ILUT method, we want to combine Fixed-point ILU with the Block ILUT method.
We want it to do the tasks where the current Block ILUT method struggles. Given its iterative structure it should perform pretty well when tasked with completing only part of the factorization.

Therefore consider various combinations of calculating certain blocks via Block ILUT (see section 4.2), other blocks with the newly proposed Fixed-point ILU method (see section 5.3). For example like in algorithm 5.2. Then look how they perform on the various test problems. Table 5.4 shows the number of multigrid iterations till convergence for different combinations of Block ILUT and Fixed-point ILU. This is compared to some of the previous smoothers.

ALGORITHM 5.2: Block Fixed-point ILU

Goal: Determine sparse lower and upper triangular L, U such that $LU \approx A$.
1. for $i = 1, ..., K$ (in parallel):
2. $\quad L_i, U_i = \text{ILUT}(A_{ii})$.
3. end for
4. for $i = 1, ..., K$ (in parallel):
5. $\quad$ Solve for B_i in $U_i^T B_i^T = A_{i\Gamma}^T$.
6. $\quad$ Solve for C_i in $L_i C_i = A_{\Gamma i}$.
7. end for

Determine E, F using the following, initial E and F are the lower- and upper triangular parts of $A_{\Gamma\Gamma}$. Sparsity pattern S is restricted to the block $\Gamma\Gamma$.
8. for $sweep = 1, 2, ...$ until convergence:
9. $\quad$ for $(i, j) \in S$ (in parallel):
10. $\quad\quad$ if $i > j$:
11. $\quad\quad\quad l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}\right) / u_{jj}$
12. $\quad\quad$ else:
13. $\quad\quad\quad u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
14. $\quad\quad$ end if
15. $\quad$ end for
16. end for

As mentioned, in table 5.4 we consider 4 such combinations of Block ILUT and Fixed-point ILU. They are elaborated on after table 5.4. Algorithm 5.2 matches combinations 3 and 4. This is also the combination we will work with in upcoming sections.

Here fixed-point 1 denotes the method of factorizing the entire matrix using Fixed-point ILU, for S use the sparsity pattern of A. As mentioned before, this constructs the same factorization as ILU(0). And indeed, it shows the same steps till convergence as in ILU(0).

In fixed-point 2 we calculate the L_{ii}, U_{ii} via Block ILUT, the B_i, C_i and D using Fixed-point ILU. In fixed-point 3 we calculate the L_{ii}, U_{ii}, B_i and C_i via Block ILUT. Only the final right bottom block using Fixed-point ILU. For both we take S the sparsity pattern of A. We see the method works, but not better then other methods discussed before.

Finally, in fixed-point 4 we again calculate everything but the right bottom block via Block ILUT, the final bit using Fixed-point ILU. But this time we allow for the entire bottom right block to be filled. That is, S equals this entire block. This time we see the same steps till convergence as Block ILUT. And the constructed L and U factorize to something very close to S. However as we will see in section 5.4.4 this method is not cheaper then the Block ILUT method.

			ILU(0)	Global ILUT	Block ILUT		fixed-point1	fixed-point2	fixed-point3	fixed-point4
K=4	p=2	h=3	4	3	2		4	4	4	2
		h=4	3	2	2		3	3	3	2
	p=3	h=3	3	2	1		3	3	3	1
		h=4	3	2	2		3	3	3	2
	p=4	h=3	4	2	1		4	4	3	1
		h=4	3	2	1		3	3	3	1
	p=5	h=3	4	2	1		4	4	4	1
		h=4	4	2	1		4	4	4	1
K=16	p=2	h=2	4	3	1		4	4	4	1
		h=3	4	3	1		4	4	4	1
	p=3	h=2	4	3	1		4	4	4	1
		h=3	3	2	1		3	3	3	1
	p=4	h=2	4	2	1		4	4	4	1
		h=3	4	2	1		4	4	4	1
K=64	p=2	h=1	5	3	1		5	5	5	1
		h=2	4	3	1		4	4	4	1
	p=3	h=1	6	3	1		6	6	6	1
		h=2	4	3	1		4	4	4	1
	p=4	h=1	6	3	1		6	6	6	1
		h=2	4	3	1		4	4	4	1

Table 5.4: Number of multigrid iterations till convergence for the combined Block ILUT Fixed-point method.

These results show that there is potential for a combined approach of Block ILUT and Fixed-point ILU. Especially the case referred to in the table as fixed-point 3. The number of multigrid iterations necessary is acceptable, the cost for factorization and smoothing is expected to be low.
Probably we are looking at a factorization somewhere between fixed-point 3 and fixed-point 4. Allowing for more nonzero entries gives chance for more accurate factorizations. If we choose these locations in a clever manner we may see both a lower number of necessary multigrid iteration, as well as factorization cost that remains low.

5.4.2. Sparsity pattern

Sparsity pattern strategies

In section 5.4.1 we have seen that the accuracy of a factorization is strongly influenced by the allowed number of nonzero entries. Of course the number of nonzeros also has an influence on the cost of the factorization. We have seen that if we take S to be the sparsity pattern of $A_{\Gamma\Gamma}$ that the approximation is not good enough. And if we take S to be the entire bottom right block that this is likely too expensive.

Therefore we want to look at a couple of different sparsity patterns that seem logical and lie somewhere in between these two extremes. Sorted from a smaller amount of nonzeros to very many nonzeros, consider the following options for S:
S_1: nonzero pattern of $A_{\Gamma\Gamma}$, i.e. the same as fixed-point3 in section 5.4.1.
S_2: the level-1 set of $A_{\Gamma\Gamma}$, i.e. the nonzero pattern of $L_\Gamma U_\Gamma$, the ILU(0) factorization of $A_{\Gamma\Gamma}$.
S_3: nonzero pattern of the bottom right block of A^2, that is the nonzero pattern of $\sum[A_{\Gamma i}A_{i\Gamma}] + A_{\Gamma\Gamma}$.
S_4: the level-1 set of A in the bottom right block, that is the nonzero pattern of $\sum[B_i C_i] + A_{\Gamma\Gamma}$.
S_5: the entire bottom right block, i.e. the same as fixed-point4 in section 5.4.1.
S_6: something dynamic (discussed later, section 5.6).

Some motivation for these candidates. S_1 and S_5 are obvious and discussed before. S_2 till S_4 take inspiration from a concept called level of fill, a concept regularly used in the field of ILU factorizations and explained among other places in [21]. S_2 till S_4 are variations on the level-1 set of A restricted to the bottom right block.
S_2 is the cheapest of these, only needing to consider $A_{\Gamma\Gamma}$. S_3 is included as we expect it to look similar to S_4, but cheaper to determine and cheaper to use. This because we expect $A_{\Gamma i}$ and $A_{i\Gamma}$ to have less nonzero entries then B_i and C_i. S_4 is arguably closest to this level-1 set. Also, the block D in Block ILUT has this nonzero pattern. D is not upper- or lower triangular, but one would expect it is possible to find a good approximation using this sparsity pattern.

Test Results

For these options for the sparsity pattern S we want to know a few things. Firstly we want to know how big these sets are, we want to know how expensive it is to determine these sets, we want to know the cost of factorization using these sparsity patterns. Finally we want to know the number of steps required for the multigrid method to converge, given the factorization based on these $S_1, S_2, ...$

The size of $S_1, S_2, ...$ and the steps for the multigrid method are easily presented in a table (see table 5.5). Determining the cost of deciding which (i, j) are in S can be more difficult, especially for the cases S_3 and S_4.

These tests are performed where each update is made in sequential order and thus the factorization is finished in a single step. Multiple reasons for this:
1. We do not want to introduce too many things at once, vary to many aspects at the same time. Rather first find out what is a suitable sparsity pattern, then find out what is a suitable number of parallel factorization step, then see whether these combine well.
2. If the factorization does not perform well now, it certainly will not perform well when the factorization is computed in parallel.
3. These tests were performed before the tests with parallel implementation.

Looking at these results we indeed see a pattern that more elements in S_i leads to less iterations required for the multigrid method. However not very convincing, the decrease in number of iterations is not very significant between S_1 and S_4. There is a big drop when we make the jump to S_5. But as mentioned before, S_5 is too expensive, especially for larger K.

These results are thus slightly disappointing. More was expected of S_4 in particular as this mimics the sparsity pattern of D in Block ILUT. This means S_1 is probably the best choice of sparsity pattern, compared to the others. Not only because of the increased cost as a result of the size of S_i, but also because of the additional cost of determining S_2,S_3,S_4.

			size S_1	size S_2	size S_3	size S_4	size S_5		steps S_1	steps S_2	steps S_3	steps S_4	steps S_5
K=4	p=2	h=3	205	221	421	1045	1369		4	4	3	2	2
		h=4	365	381	709	3605	4761		3	3	3	2	2
	p=3	h=3	329	365	725	1281	1681		3	3	2	2	2
		h=4	553	589	1141	4033	5329		3	4	2	3	1
	p=4	h=3	477	541	1101	1541	2025		3	3	2	3	1
		h=4	765	829	1645	4485	5929		4	4	2	3	1
	p=5	h=3	649	749	1549	1825	2401		4	4	2	3	1
		h=4	1001	1101	2221	4961	6561		4	4	3	3	1
K=16	p=2	h=2	837	981	2085	3277	13689		4	4	3	3	1
		h=3	1317	1461	2949	10509	45369		4	4	3	3	1
	p=3	h=2	1509	1995	5457	4701	19881		4	4	3	3	1
		h=3	2181	2505	5313	12957	56169		3	3	2	3	1
	p=4	h=2	2373	3165	7725	6381	27225		4	4	3	3	1
		h=3	3237	3813	8325	15661	68121		4	4	2	3	1
K=64	p=2	h=1	2989	4113	8165	6037	90601		5	5	4	5	1
		h=2	4109	4893	10717	17045	275625		4	4	4	4	1
	p=3	h=1	5957	8339	15633	10845	170569		6	5	5	6	1
		h=2	7525	10495	29425	24637	405769		4	4	3	3	1
	p=4	h=1	9933	14037	25557	17045	275625		6	5	5	6	1
		h=2	11949	16837	42133	33621	561001		4	4	3	3	1

Table 5.5: Size of sparsity pattern and number of multigrid iterations till convergence.

Determining S_2, S_3, S_4

S_1 and S_5 are easily determined from looking at the matrix A. S_2, S_3, S_4 require more work to establish these sets.
S_2 requires ILU(0) factorization on the block $A_{\Gamma\Gamma}$, costing $\mathcal{O}(N_{\text{interface}} p^d)$. For S_3 and S_4 matrix multiplications are required. Unfortunately determining the sparsity pattern of a matrix product is the same amount of work (at least in terms of order of magnitude) as actually performing the matrix product. Worst case (full matrices) determining S_3 and S_4 would thus have a cost of $\mathcal{O}(N^3_{\text{interface}})$. In practice, though the pattern of nonzeros is not very predictable, both $A_{i\Gamma}, A_{\Gamma i}$ and B_i, C_i consist of mostly zeros.
As we expect more nonzeros in B_i, C_i compared to $A_{i\Gamma}, A_{\Gamma i}$, determining S_4 will be slightly more expensive compared to S_3.

5.4.3. Parallel performance

In section 5.3.2 we have seen that if you perform the fixed-point method in a specific sequence, that the factorization converges in one step. It would not be fair to abuse this as in practice you will not use this. You will want to use the option to work in parallel, especially since the other steps of the factorization also lend itself to work in parallel.
The obvious approach would be to use as many cores as you have patches. Then every core can calculate one set of L_i, U_i, one set of B_i, C_i and have its contribution to the bottom right block.

Therefore in this section we will pretend to be calculating the bottom right block in parallel. Two reasons for this: 1. the test problems are not big enough for it to be efficient to work in parallel and 2. it would take a lot of time to learn how to program this. And we have reason to think our 'pretend situation' is close enough as to not expect any change in results.

The sparsity pattern S is stored as a list of indices (i, j). This list is then split in K pieces $S_1, ..., S_K$. Then S_1 makes its first update, S_2 makes its first update, and so on. Then they can all make a second update, a third update, ... This simulates asynchronous parallelism, where the updated value is used when available, old value otherwise.
This is a reasonable approximation of working on K cores using shared memory. Not quite, sometimes you can use an updated matrix entry that should not have been updated yet. For example, the first update of S_2 should not be able to use that first updated value of S_1. On the other hand, sometimes you have to use an old value where an updated value would have probably already been available. This because not every update take the same time, entries more to the left and the top are faster as the summation is shorter. But all together, this should not make too much a difference.

Test Results

Suppose we make a factorization using this method. What is the number of iterations needed to make a decent factorization? Does this number depend on the values of K, p and h? If so, how? Do we see different results for the case with diagonal scaling and the case without diagonal scaling?

Answers to these questions are given in tables 5.6 and 5.7. To test how much the factorization changes between steps we consider the norm of the ILU residual, that is the L^2 norm of the matrix $R = A - LU$. These tests use the sparsity pattern S_1 as in section 5.4.2, that is S is equal to the sparsity pattern of $A_{\Gamma\Gamma}$.
For readability we decided to leave spots in these tables empty from the moment this residual norm does no longer change. Or at least the first 4 decimals do no longer change. We consider this as 'the factorization has converged'.

From these tables we can conclude that the interesting number of iterations is between 2 and 4. Most likely 2 steps should be a good enough approximation to expect decent performance in the multigrid method. There is no reason to go beyond 4, the factorization has (as good as) converged at 4.
This number does not seem to depend on the values of K, p and h or on diagonal scaling.

It does show that problems with a higher number of patches benefit more from parallelization though. A problem with $K = 4$, where the amount of work per iteration is divided by 4, needing 2 or 3 iterations is not a big improvement. Especially taking into account the communication necessary. However for $K = 64$, needing 2 or 3 iterations using 64 cores is an enormous improvement over calculating it all in 1 step but on a single core.

			steps					
			0	1	2	3	4	5
K=4	p=2	h=3	673.99	0.7191	0.6386	0.6384	0.6385	
		h=4	1001.0	0.9923	0.9353	0.9352		
	p=3	h=3	1846.7	0.6296	0.4487	0.4477	0.4478	
		h=4	2844.0	0.7257	0.5751	0.5743	0.5744	
	p=4	h=3	3975.6	0.6966	0.367	0.3652	0.3651	
		h=4	6354.9	0.7416	0.4452	0.4436		
	p=5	h=3	7314.6	0.7263	0.3129	0.3102	0.3101	
		h=4	12204	0.7552	0.3738	0.3715		
K=16	p=2	h=2	600.33	1.6468	1.3907	1.3906		
		h=3	953.07	2.1038	1.8513	1.8512		
	p=3	h=2	1464.4	1.6979	1.0174	1.0153		
		h=3	2612.0	1.9229	1.3593	1.3576	1.3577	
	p=4	h=2	2697.2	1.8949	0.9470	0.9457		
		h=3	5622.9	1.9447	1.1217	1.1208		
K=64	p=2	h=1	422.33	4.1341	3.9572	3.9581		
		h=2	848.98	3.7425	3.2206	3.1297		
	p=3	h=1	765.45	4.0984	3.5734	3.5744	3.5745	
		h=2	2071.3	10.511	2.2672	2.2629		
	p=4	h=1	1221.5	6.4752	3.3156	3.2558	3.2559	
		h=2	3815.2	4.0288	2.1798	2.0082	2.0083	

Table 5.6: ILU residual after several steps of the parallel factorization process.

			steps					
			0	1	2	3	4	5
K=4	p=2	h=3	4.3680	0.3821	0.3366	0.3365		
		h=4	6.9250	0.52261	0.4901	0.4900		
	p=3	h=3	9.8189	0.3576	0.2511	0.2501		
		h=4	15.073	0.4025	0.3111	0.3103		
	p=4	h=3	16.805	0.3784	0.2148	0.2127		
		h=4	25.303	0.3979	0.2465	0.2447		
	p=5	h=3	25.169	0.3941	0.1973	0.1945	0.1944	
		h=4	37.227	0.405	0.2170	0.2143	0.2142	
K=16	p=2	h=2	6.2391	0.8948	0.733	0.7329		
		h=3	9.8007	1.1327	0.9779	0.9778		
	p=3	h=2	15.013	1.0515	0.5918	0.5903	0.5904	
		h=3	22.192	1.1585	0.7585	0.7570		
	p=4	h=2	27.363	1.2356	0.5400	0.5441		
		h=3	37.840	1.2759	0.6494	0.6483		
K=64	p=2	h=1	8.2274	1.5484	1.366	1.3669	1.3670	
		h=2	13.097	2.0251	1.7201	1.7195		
	p=3	h=1	23.094	2.1613	1.4314	1.4329	1.433	
		h=2	31.635	2.2553	1.3979	1.3951		
	p=4	h=1	46.817	4.6863	1.5029	1.5030		
		h=2	57.614	2.6927	1.3224	1.2923	1.2921	

Table 5.7: ILU residual after several steps of the parallel factorization process (including diag scaling).

5.4.4. Computational Costs

In section 5.3.3 we already went into the cost of applying Fixed-point ILU on the full matrix, this time we consider the case where we apply it only on the bottom right submatrix.
We have seen the cost is dependent of the size of the allowed sparsity pattern S as well as the number of nonzeros in the rows and columns of the guesses L and U (as these determine the length of the summations).

Sounds easy, but it is in fact very difficult to make sensible claims about the cost of this method. The reason this is so difficult is because the nonzero pattern of B_i, C_i and also E and F is unpredictable. The size of S_1 and S_5 are easily determined:

$$|S_1| = \mathcal{O}(N_{\text{interface}} p^d), \qquad |S_5| = \mathcal{O}(N^2_{\text{interface}})$$

The others (S_2, S_3, S_4) are less predictable. It is not possible to motivate similar claims through a proof, but it is possible to make an educated guess through numerical testing and counting number of nonzeros. These tests show the number of nonzeros lie closer to that in S_1, also that the number of nonzeros stay under control for increasing one of the parameters K,p or h.

For the length of the summations, again through numerical testing, these appear to be $\mathcal{O}(p^d)$. Finally there is the cost of determining S, but this is already discussed in section 5.4.2.

This leads to the following table. Note that these results stem from numerical tests, not from a logical proof. This is especially the case for S_2,S_3,S_4.

	cost factorization	cost determine S
S_1	$\mathcal{O}(N_{\text{interface}} p^{2d})$	-
S_2	$\mathcal{O}(N_{\text{interface}} p^{2d})$	$\mathcal{O}(N_{\text{interface}} p^d)$
S_3	$\mathcal{O}(N_{\text{interface}} p^{2d})$	$\mathcal{O}(N^3_{\text{interface}})$
S_4	$\mathcal{O}(N_{\text{interface}} p^{2d})$	$\mathcal{O}(N^3_{\text{interface}})$
S_5	$\mathcal{O}(N^2_{\text{interface}} p^d)$	-

Table 5.8: Complexity of Block Fixed-point ILU.

The cost of applying these factorizations as a smoother are $\mathcal{O}(N_{\text{patch}} p^d)$. Will be considerably cheaper then Block ILUT as this factorization is actually lower- and upper triangular rather then being only block lower- and block upper triangular. An expansive direct solve (or further approximation) is saved.

5.4.5. Discussion

This section stands to combine what is learned in sections 5.4.1 to 5.4.4. Then compare this 'ultimate Block Fixed-point ILU smoother' to the smoothers seen before, Block ILUT in particular. Discuss the cost of factorization, the amount of work per iteration of the multigrid method and of the amount of steps necessary in the multigrid method.

The variants of the Block Fixed-point ILU smoother we consider in this section are those with an allowed sparsity pattern of S_1 or S_4 (see section 5.4.2) and where the factorization is based on 2 or 4 parallel factorization steps.
The number of steps required for the multigrid method to converge is shown in table 5.9. The method that stands out is the one with sparsity pattern S_1 and 2 parallel factorization steps. It does not stand out because it has lower number of multigrid iterations, it stands out as it shows comparable number of multigrid iterations, while being significantly cheaper.
Not only does it show that 2 parallel factorization steps is enough (it scores just as well as the variant with 4 steps), it also compares very well to Block ILUT.

				Global	Block		S_1	S_1	S_4	S_4
			ILU(0)	ILUT	ILUT		2 steps	4 steps	2 steps	4 steps
K=4	p=2	h=3	4	3	2		4	4	3	2
		h=4	3	2	2		3	3	2	2
	p=3	h=3	3	2	1		3	3	3	3
		h=4	3	2	2		3	3	3	2
	p=4	h=3	4	2	1		3	3	3	3
		h=4	3	2	1		3	3	3	3
	p=5	h=3	4	2	1		4	4	3	3
		h=4	4	2	1		3	4	3	3
K=16	p=2	h=2	4	3	1		4	4	3	3
		h=3	4	3	1		4	4	3	3
	p=3	h=2	4	3	1		4	4	3	3
		h=3	3	2	1		3	3	3	3
	p=4	h=2	4	2	1		4	4	3	3
		h=3	4	2	1		4	4	3	3
K=64	p=2	h=1	5	3	1		5	5	5	5
		h=2	4	3	1		4	4	4	4
	p=3	h=1	6	3	1		6	6	6	6
		h=2	4	3	1		4	4	3	3
	p=4	h=1	6	3	1		6	6	6	6
		h=2	4	3	1		5	4	3	3

Table 5.9: Steps till convergence for some of the best Block Fixed-point ILU methods

The number of multigrid iterations in Block ILUT may be considerably lower than that in S_1 with 2 factorization steps. The cost of factorization is higher for Block ILUT, the cost of performing a smoothing step is more expensive for Block ILUT.

5.5. ParILUT

5.5.1. Adjusting sparsity pattern

To compute an ILU factorization where the sparsity pattern of the L and U factors are adapted to the values of A, we propose interleaving a method for computing an ILU factorization for a fixed sparsity pattern with a method for adjusting the sparsity pattern [23].
In such a pattern we do not need to compute the ILU factorization for a given sparsity pattern exactly, since that sparsity pattern will be further adjusted. Therefore, it is natural to use one sweep of the fixed-point ILU algorithm to cheaply approximate an ILU factorization in between adjusting the sparsity pattern.

To add nonzeros to the sparsity pattern S of the current L and U approximations, consider an entry r_{ij} of $R = A - LU$. If L and U are exact ILU factors for the pattern S, then r_{ij} would be zero. If r_{ij} is large in magnitude, then either L and U are very inaccurate incomplete factors, or (i, j) is not in the sparsity pattern of S. In the latter case, it is natural to consider (i, j) as a nonzero location to add to the sparsity pattern.
A candidate can be added to S if its corresponding r_{ij} is large in magnitude according to a threshold. Alternatively, all candidates can be added to S.
Note that when S corresponds to the level-0 ILU factorizations, the set of candidates corresponds to the pattern of the level-1 ILU factorization.

If you want to remove nonzeros from the sparsity pattern S, simply remove nonzeros in L and U if they are small in magnitude. This can be either a threshold on the size of the nonzeros and/or a threshold on the number of nonzeros that can be used.

5.5.2. Algorithm outline

ALGORITHM 5.3: ParILUT

Goal: Determine sparse lower and upper triangular L, U such that $LU \approx A$.
1. Start with initial guess for L and U
2. for $sweep = 1, 2, \ldots$ until convergence:
3. Identify candidate locations
4. Compute ILU residual at candidate locations
5. Estimate ILU residual norm
6. Add m_L nonzeros to L and m_U nonzeros to U
7. Do one sweep of the fixed-point ILU algorithm
8. Remove the m'_L and m'_U smallest magnitude elements from L and U, respectively
9. Do one sweep fo the fixed-point ILU algorithm
10. end for

This is one possibility, the variant displayed in [23]. However, several aspects to this can be adjusted.
In the algorithm it was decided to add nonzeros to the sparsity pattern before removing nonzeros. Compared to removing and then adding nonzeros, this gives somewhat more accurate L and U factors at the end of each step, although the cost is slightly higher because the fixed-point ILU sweep then operate on more nonzeros.
In this example it was also decided to, at each step, remove the same number of nonzeros as the number of nonzeros that were added earlier in the step. Another strategy could be to allow the number of nonzeros to grow with each step, which would be more suited to our current problem.

The initial approximations chosen for L and U are the lower- and upper triangular parts respectively. In the procedure for adjusting the sparsity pattern, we must specify the initial values of the selected candidate locations, i.e. the matrix locations added to L and U.
One natural choice for these initial values is zero. However, since we use a single fixed-point sweep to adjust the nonzero values, the zeros added do not contribute to adjusting existing nonzero values of L and U until they themselves have been updated to a nonzero value. An alternative choice for the initial value of a newly added l_{ij} or u_{ij} is

$$l_{ij} = r_{ij}/u_{jj}, \text{ if } i > j, \qquad \text{or } u_{ij} = r_{ij}/l_{ii}, \text{ if } i < j.$$

If only a single nonzero is added to the sparsity pattern, these are the values of l_{ij} or u_{ij} that would reduce r_{ij} to zero. However, adding such a nonzero will also generally change the residual at other locations besides (i, j). The strategy we will use is to set the initial value of newly added l_{ij}, u_{ij} at zero.

In section 5.6 we will discuss how we combine this method with the block structure of our matrix A.

5.6. Block ParILUT

5.6.1. General info

In section 5.4 we used a combination of Block ILUT and a Fixed-point ILU method where the sparsity pattern S was static. The pattern was determined beforehand and not allowed to change. Next to this we want to consider a variant where the sparsity pattern is more dynamic. Again, we calculate the L_i, U_i, B_i and C_i using Block ILUT, but this time we calculate the bottom right block using ParILUT, the approach of section 5.5. This approach will be called Block ParILUT.

From the above paragraph one might expect Block Fixed-point ILU (section 5.4) and Block ParILUT (current section) to be very similar. They are however very different in a key aspect. This difference is caused by the expression we encounter when needing to calculate the ILU residual matrix R. To show this, consider the following two factorizations of A.

$$A \approx \begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ B_1 & \cdots & B_K & I \end{pmatrix} \begin{pmatrix} U_1 & & & C_1 \\ & \ddots & & \vdots \\ & & U_K & C_K \\ & & & D \end{pmatrix} = LU.$$

$$A \approx \begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_K & \\ B_1 & \cdots & B_K & E \end{pmatrix} \begin{pmatrix} U_1 & & & C_1 \\ & \ddots & & \vdots \\ & & U_K & C_K \\ & & & F \end{pmatrix} = LU.$$

Block ILUT creates a factorization of the top kind. Block Fixed-point ILU creates a factorization of the bottom kind and does so without needing to calculate D (for an appropriately chosen sparsity pattern that is).
Block ParILUT also creates a factorization of the bottom kind, however here it is needed to first calculate D.

This is an important distinction. Not wanting to calculate D was one of the reasons we wanted to make alterations to Block ILUT in the first place. This does not mean that Block ParILUT is a useless method though, the cost of factorization may not be an improvement compared to Block ILUT, the cost of a single smoothing step will be cheaper.

We need to consider the ILU residual $R = A - LU$ in order to know which candidates to add to the sparsity pattern. More specifically we need to consider R limited to the bottom right block, since this is the only part of the matrix that has entries of significant magnitude and also the only part of the matrix that changes between different iterations of Block ParILUT. Comparing the expression for $R_{\Gamma\Gamma}$ to the expression for D seen in the Block ILUT method we see the following

$$R_{\Gamma\Gamma} = A_{\Gamma\Gamma} - (LU)_{\Gamma\Gamma} = A_{\Gamma\Gamma} - \sum_{i=1}^{K} B_i C_i - EF$$

$$D = A_{\Gamma\Gamma} - \sum_{i=1}^{K} B_i C_i$$

That is $R_{\Gamma\Gamma} = D - EF$, where E, F are the current approximations. Block ParILUT will therefore be calculating L_i, U_i, B_i, C_i and D as in Block ILUT. Afterwards it will determine E, F from D using ParILUT. Initial E and F are the lower- and upper triangular part of D, these are iteratively improved during ParILUT.

5.6.2. Residual matrix

As we are using ParILUT on the bottom right block we need to determine what would be a good strategy for adding locations to the sparsity pattern S. For this we look at the ILU residual $R_{\Gamma\Gamma} = D - EF$. If $r_{ij} \neq 0$ we could consider adding position (i, j) to the pattern S. There are several ways of deciding what candidates to add:
- Take the m largest elements of $R_{\Gamma\Gamma}$.
- Take the m largest elements per row or column of $R_{\Gamma\Gamma}$.
- Take every candidate with absolute value bigger then a given threshold.
- A combination of the above.
- Add all candidates.

To make a better decision on which of these strategies would be most beneficial we need to take a closer look on $R_{\Gamma\Gamma}$. How many nonzeros does it contain? How many of those are 'small', how many are 'large'? How are the big entries spread over the matrix? Are they grouped, scattered, is there a pattern?
Table 5.10 and figure 5.2 help with this. In table 5.10 we see the total number of nonzeros in R, as well as the fraction of nonzeros which are greater then a given threshold. That is the number of nonzeros where $|r_{ij}| > \tau$ divided by the total number of nonzeros. Figure 5.2 shows how these entries are scattered over the matrix for a specific case $K = 16, p = 3, h = 2$.

			total nz	max	fraction of nonzeros where $\lvert r_{ij}\rvert > \tau$							
			R	$\lvert r_{ij}\rvert$	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-6}	10^{-8}	10^{-10}	10^{-12}
K=4	p=2	h=3	999	0.1205	0.008	0.262	0.564	0.721	0.835	0.849	0.849	0.849
		h=4	3541	0.1191	0.002	0.146	0.462	0.707	0.868	0.912	0.917	0.917
	p=3	h=3	1244	0.0885	0.000	0.163	0.445	0.611	0.756	0.780	0.780	0.780
		h=4	3980	0.0874	0.000	0.113	0.421	0.663	0.825	0.872	0.879	0.879
	p=4	h=3	1502	0.0684	0.000	0.107	0.359	0.550	0.705	0.730	0.730	0.730
		h=4	4429	0.0684	0.000	0.075	0.366	0.604	0.788	0.839	0.846	0.847
	p=5	h=3	1813	0.0705	0.000	0.069	0.288	0.485	0.639	0.676	0.676	0.676
		h=4	4931	0.0705	0.000	0.050	0.315	0.544	0.745	0.802	0.813	0.813
K=16	p=2	h=2	3076	0.1311	0.012	0.347	0.648	0.778	0.817	0.817	0.817	0.817
		h=3	10253	0.1447	0.007	0.215	0.578	0.763	0.891	0.904	0.904	0.904
	p=3	h=2	4624	0.1351	0.003	0.200	0.529	0.655	0.739	0.741	0.741	0.741
		h=3	12768	0.0879	0.000	0.151	0.473	0.668	0.832	0.855	0.857	0.857
	p=4	h=2	6437	0.1663	0.002	0.135	0.430	0.584	0.682	0.687	0.687	0.687
		h=3	15525	0.0682	0.000	0.098	0.401	0.613	0.791	0.818	0.819	0.819
K=64	p=2	h=1	5515	0.5045	0.017	0.454	0.643	0.649	0.664	0.664	0.664	0.664
		h=2	16003	0.1292	0.012	0.364	0.672	0.796	0.833	0.833	0.833	0.833
	p=3	h=1	10768	0.4760	0.004	0.329	0.502	0.571	0.583	0.583	0.583	0.583
		h=2	24569	0.1351	0.002	0.221	0.546	0.677	0.752	0.753	0.753	0.753
	p=4	h=1	17311	0.4387	0.003	0.217	0.448	0.530	0.553	0.553	0.553	0.553
		h=2	34276	0.1663	0.001	0.144	0.451	0.603	0.700	0.704	0.704	0.704

Table 5.10: Residual matrix: number of entries over a given threshold.

Based on these results we have a good intuition for how the bigger entries in R are spread. It does not seem like the bigger entries are grouping together. For example if we were to allow everything with $|r_{ij}| > 10^{-2}$, we improve over the entire matrix.
This suggests that it should be good to have a rule of this kind and that is not necessary to have different rules from row to row. Viable strategies therefore seem:
- Add everything $|r_{ij}| > 10^{-2}$, $|r_{ij}| > 10^{-4}$.
- Add everything $r_{ij} \neq 0$.

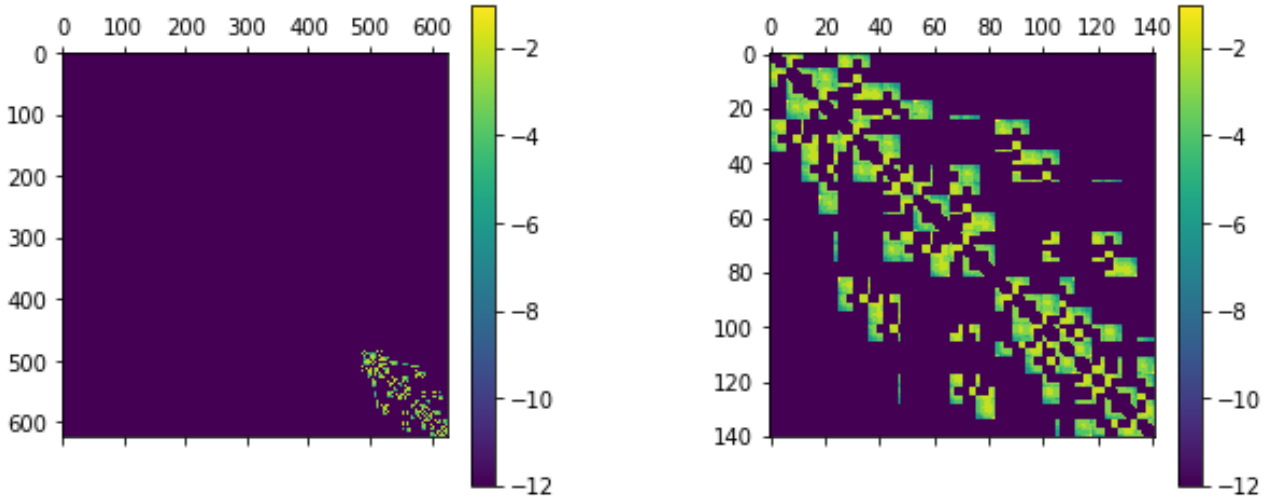


Figure 5.2: Residual matrix: order of magnitude of $|r_{ij}|$ for the large matrix R (left) or the bottom right block $R_{\Gamma\Gamma}$ (right). $A(K = 16, p = 3, h = 2)$.

5.6.3. Algorithm

With the insights from section 5.6.2 we know what could be good strategies for changing the sparsity pattern S. This leads to the Block ParILUT algorithm, a combination of Block ILUT and ParILUT.
The ParILUT part of the algorithm can be simplified slightly as we will allow the size of S to increase. We choose not to remove any (i, j) from S. The increase to the size of S is very manageable and this allows us to do only one Fixed-point ILU sweep per ParILUT sweep.

ALGORITHM 5.4: Block ParILUT

Goal: Determine sparse lower and upper triangular L, U such that $LU \approx A$.
1. for $i = 1, ..., K$ (in parallel):
2. $L_i, U_i = \text{ILUT}(A_{ii})$.
3. end for
4. for $i = 1, ..., K$ (in parallel):
5. Solve for B_i in $U_i^T B_i^T = A_{i\Gamma}^T$.
6. Solve for C_i in $L_i C_i = A_{\Gamma i}$.
7. end for

Determine E, F using the following. Initial E and F are the lower- and upper triangular parts of $A_{\Gamma\Gamma}$
8. for $sweep = 1, 2, ...$ until convergence:
9. Compute $R_{\Gamma\Gamma} = D - EF$.
10. Add all (i, j) with $|r_{ij}| > \tau$ to S.
11. Do one sweep of the fixed-point ILU algorithm.
12. end for

5.6.4. Numerical Results

We will test the Block ParILUT algorithm in multiple situations. S_4 will be the logical initial sparsity pattern, as this is the sparsity pattern of D. Initial E and F will be the lower- and upper triangular parts of D. We want to use both a stronger threshold $\tau = 10^{-2}$ as well as a more lenient threshold $\tau = 10^{-4}$.
Next to this we want to test what happens if we start with a smaller initial sparsity pattern S_1. Will this make a good approximation using a smaller amount of nonzeros? Initial E and F will have to be limited to this initial sparsity pattern as well.

Tables 5.11, 5.12 and 5.13 show the results to the tests using the Block ParILUT algorithm. In these tables we see both the size of the sparsity pattern as well as the number of multigrid operations necessary if you use that factorization as a smoother.
Table 5.11: Initial sparsity pattern S_1, threshold $\tau = 10^{-2}$.
Table 5.12: Initial sparsity pattern S_4, threshold $\tau = 10^{-2}$.
Table 5.13: Initial sparsity pattern S_4, threshold $\tau = 10^{-4}$.

Finally in table 5.14 we compare the Block ParILUT algorithm with the Block Fixed-point ILU algorithm in as fair a comparison possible. That is the same initial sparsity pattern, and the same number of ParILUT or Fixed-point ILU sweeps. This to show what difference the dynamical sparsity pattern makes compared to the static sparsity pattern.

Looking at these tables we can make some remarks. The results for initial sparsity pattern S_1 are very surprising. One would expect this sparsity pattern to grow rapidly towards S_4, the submatrix D we try to approximate has this pattern after all. This is not the case however, it makes a good approximation without needing all these nonzeros. The number of multigrid iterations are also very competitive.
This makes it unclear which initial sparsity pattern is preferred. S_1 requires less nonzeros S_4 sometimes needs a few less multigrid cycles for the same number of ParILUT sweeps.

The suggested number of ParILUT sweeps would be 2 if using S_4 as the initial pattern or 2 or 3 if using S_1 as the initial pattern. Also the stronger threshold performs better then the more lenient threshold. The more lenient threshold allows more additional entries, but these do not improve the number of multigrid iterations necessary.

Table 5.14 shows that Block ParILUT requires less multigrid iterations compared to the same number of factorization steps using the static sparsity pattern.

			|S|	|S|	|S|	|S|	|S|	|S|	MG	MG	MG	MG	MG	MG
			0	1	2	3	4	5	0	1	2	3	4	5
K=4	p=2	h=3	205	449	478	480	482	482	3	3	2	2	2	2
		h=4	365	786	837	854	861	862	3	2	2	2	2	2
	p=3	h=3	329	592	625	628	628	628	6	2	2	2	2	2
		h=4	553	964	1026	1042	1043	1043	5	2	2	2	2	2
	p=4	h=3	477	834	870	872	872	872	13	3	2	2	2	2
		h=4	765	1291	1362	1380	1383	1384	11	3	2	2	2	2
	p=5	h=3	649	1038	1109	1113	1113	1113	26	5	3	3	3	3
		h=4	1001	1581	1685	1704	1708	1709	23	4	3	3	2	3
K=16	p=2	h=2	837	2223	2405	2462	2470	2476	4	3	3	3	3	3
		h=3	1317	3071	3337	3440	3470	3475	4	3	3	3	3	3
	p=3	h=2	1509	3025	3353	3427	3456	3470	7	3	3	2	3	3
		h=3	2181	4083	4392	4438	4447	4449	6	3	2	2	2	2
	p=4	h=2	2373	4266	4662	4742	4757	4767	17	3	3	3	3	3
		h=3	3237	5848	6163	6205	6211	6211	13	3	3	2	2	2
K=64	p=2	h=1	2989	8023	8855	9179	9248	9269	7	5	4	4	4	4
		h=2	4109	11579	13046	13582	13667	13712	4	4	4	4	3	3
	p=3	h=1	5957	12364	13696	13890	13954	13974	11	5	5	4	4	4
		h=2	7525	15594	17457	18128	18386	18468	7	3	3	3	3	3
	p=4	h=1	9933	18043	20070	20491	20539	20541	23	6	5	5	5	5
		h=2	11949	21979	24639	25301	25482	25551	17	4	3	3	3	3

Table 5.11: Size of S and the number of multigrid iterations for factorizations after 0,...,5 steps of Block ParILUT (initial sparsity pattern S_1, $\tau = 10^{-2}$).

			|S|	|S|	|S|	|S|	|S|	|S|	MG	MG	MG	MG	MG	MG
			0	1	2	3	4	5	0	1	2	3	4	5
K=4	p=2	h=3	1045	1289	1290	1290	1290	1290	3	2	2	2	2	2
		h=4	3605	4026	4031	4033	4033	4033	3	2	2	2	2	2
	p=3	h=3	1281	1544	1550	1551	1551	1551	6	2	2	2	2	2
		h=4	4033	4444	4458	4458	4458	4458	5	2	2	2	2	2
	p=4	h=3	1541	1898	1900	1900	1900	1900	13	2	2	2	2	2
		h=4	4485	5011	5018	5018	5018	5018	11	2	2	2	2	2
	p=5	h=3	1825	2214	2228	2228	2228	2228	26	3	2	2	2	2
		h=4	4961	5541	5555	5557	5557	5557	23	3	2	2	2	2
K=16	p=2	h=2	3277	4663	4785	4816	4820	4820	4	3	3	3	2	2
		h=3	10509	12263	12465	12517	12526	12536	4	3	3	3	3	3
	p=3	h=2	4701	6217	6371	6387	6400	6400	7	3	2	2	2	2
		h=3	12957	14859	15054	15114	15123	15124	6	2	2	2	2	2
	p=4	h=2	6381	8274	8477	8509	8517	8518	17	4	3	3	3	3
		h=3	15661	18272	18443	18497	18518	18522	13	3	2	3	2	2
K=64	p=2	h=1	6037	11071	11839	12147	12222	12239	7	4	4	4	4	4
		h=2	17045	24515	25500	25722	25784	25800	4	4	3	3	3	3
	p=3	h=1	10845	17252	18157	18303	18339	18359	11	5	4	4	4	4
		h=2	24637	32706	33861	34093	34166	34176	7	3	3	3	3	3
	p=4	h=1	17045	25155	26234	26339	26359	26361	23	5	5	5	5	5
		h=2	33621	43651	44872	45163	45246	45269	17	3	3	3	3	3

Table 5.12: Size of S and the number of multigrid iterations for factorizations after 0,...,5 steps of Block ParILUT (initial sparsity pattern S_4, $\tau = 10^{-2}$).

			|S|	|S|	|S|	|S|	|S|	|S|	MG	MG	MG	MG	MG	MG
			0	1	2	3	4	5	0	1	2	3	4	5
K=4	p=2	h=3	1045	1946	2133	2156	2156	2156	3	2	2	2	2	2
		h=4	3605	6052	6505	6643	6643	6643	3	2	2	2	2	2
	p=3	h=3	1281	2285	2511	2531	2531	2531	6	2	2	2	2	2
		h=4	4033	6869	7396	7532	7532	7532	5	2	2	2	2	2
	p=4	h=3	1541	2801	3105	3116	3116	3116	13	2	2	2	2	2
		h=4	4485	7479	8020	8117	8118	8118	11	2	2	2	2	2
	p=5	h=3	1825	3293	3679	3694	3694	3694	26	3	2	2	2	2
		h=4	4961	8134	8774	8833	8834	8834	23	3	2	2	2	2
K=16	p=2	h=2	3277	7233	9452	10266	10517	10555	4	3	3	3	2	2
		h=3	10509	20702	25985	28239	29007	29386	4	3	3	3	3	3
	p=3	h=2	4701	9962	12550	13618	13942	14026	7	3	2	2	2	2
		h=3	12957	24614	30080	32373	33114	33507	6	2	2	2	2	2
	p=4	h=2	6381	13459	16735	18022	18453	18518	17	4	3	3	3	3
		h=3	15661	29831	36259	38716	39504	39881	13	3	2	3	2	2
K=64	p=2	h=1	6037	16499	24542	30123	33227	34504	7	4	4	4	4	4
		h=2	17045	39184	54944	65708	71841	74902	4	4	3	3	3	3
	p=3	h=1	10845	27821	41075	49209	53415	55130	11	5	4	4	4	4
		h=2	24637	53833	72621	85703	93526	97287	7	3	3	3	3	3
	p=4	h=1	17045	42449	59819	71076	76786	79053	23	5	5	5	5	5
		h=2	33621	72835	96458	112139	121255	125592	17	3	3	3	3	3

Table 5.13: Size of S and the number of multigrid iterations for factorizations after 0,...,5 steps of Block ParILUT (initial sparsity pattern S_4, $\tau = 10^{-4}$).

			Block Fixed-point ILU		Block ParILUT	
			2 steps	3 steps	2 steps	3 steps
K=4	p=2	h=3	3	2	2	2
		h=4	2	2	2	2
	p=3	h=3	3	3	2	2
		h=4	3	2	2	2
	p=4	h=3	3	3	2	2
		h=4	3	3	2	2
	p=5	h=3	3	3	2	2
		h=4	3	3	2	2
K=16	p=2	h=2	3	3	3	3
		h=3	3	3	3	3
	p=3	h=2	3	3	2	2
		h=3	3	3	2	2
	p=4	h=2	3	3	3	3
		h=3	3	3	2	3
K=64	p=2	h=1	5	5	4	4
		h=2	4	4	3	3
	p=3	h=1	6	6	4	4
		h=2	3	3	3	3
	p=4	h=1	6	6	5	5
		h=2	3	3	3	3

Table 5.14: Number of multigrid iterations for factorizations using 2 or 3 Block Fixed-point ILU or Block ParILUT steps respectively (initial sparsity pattern S_4, $\tau = 10^{-2}$).

5.6.5. Computational Costs

The Block ParILUT factorization continues from the Block ILUT factorization and will therefore be slightly more expensive. We have determined before that computing D will cost at the worst case $\mathcal{O}(N_{\text{interface}}^3)$, but in practice it will be cheaper as D still consists of many zeros.
The new part of calculating E and F from D consists of calculating the ILU residual and Fixed-point ILU sweeps. The cost of the Fixed-point ILU sweeps is $\mathcal{O}(N_{\text{interface}} p^{2d})$ as of section 5.4.4, calculating the ILU residual will be at the same cost as computing D.

Using ParILUT as a smoother will be significantly cheaper than Block ILUT, as now the factorization will actually be lower- and upper triangular rather then only block lower- and block upper triangular. The cost will be very comparable to the cost in Block Fixed-point ILU.

5.6.6. Discussion

Block ParILUT is a very viable method. The cost of factorization will be slightly more compared to the original Block ILUT method, also requiring a few more multigrid iterations. The cost per smoothing step will be considerably cheaper though. A good comparison between Block ParILUT and the previously discussed methods Block ILUT and Block Fixed-point ILU is given in section 5.7.

Both S_1 and S_4 are effective initial sparsity patterns, $\tau = 10^{-2}$ is a suitable threshold and it is advised to use 2 or 3 ParILUT sweeps on the bottom right block.

5.7. Block ILUT vs Block Fixed-point ILU vs Block ParILUT

Finally we have to give a full comparison between the Block ILUT method and the two newly proposed methods Block Fixed-point ILU and Block ParILUT. To give them a proper comparison we want to score them on 3 criteria.
- The cost of determining the factorization.
- The cost of a single smoothing step in the multigrid method.
- The number of necessary iterations for the multigrid method to converge given this factorization.

Other criteria that could be suggested are the amount of memory needed and the degree to which the methods can be parallelized. The memory needed is close to equal for the three methods and therefore irrelevant to include in the comparison. The degree of parallelization is taken into account within the criteria cost of smoothing and cost of factorization.

For the number of necessary multigrid iteration it is easy to look at concrete numbers, like the difference between 3 or 5 iterations. For the other two criteria it is not as easy to say this method costs $\mathcal{O}(...)$ or $\mathcal{O}(...)$. But we can make claims of the kind 'method A is cheaper then method B'.
Therefore we choose to present the performance of the smoothers per category by a score of 1 to 3, where 1 would be the best scoring method per criteria and 3 the worst scoring method. These scores are seen in table 5.15.

	cost factorization	cost smoothing	iterations multigrid
Block ILUT	2	3	1
Block Fixed-point ILU	1	1	3
Block ParILUT	3	2	2

Table 5.15: Comparing Block smoothers (1 best, 3 worst).

The scores in the 'iterations multigrid' column are easily determined from comparing tables of the previous sections. For the cost of factorization Block Fixed-point ILU is cheapest, not requiring the computation of D, followed by Block ILUT and finally Block ParILUT which continues from the factorization left by Block ILUT.
For the cost of a single smoothing step Block ILUT is clearly most expensive with the factorization not being fully lower- and upper triangular and therefore forward- and backward solve not being possible. Block Fixed-point ILU is slightly cheaper then Block ParILUT as it contains slightly less nonzeros, but this difference is negligible.
We can also make a small adjustment to this table where we contribute equal scores if the difference is very small. This gives rise to table 5.16.

	cost factorization	cost smoothing	iterations multigrid
Block ILUT	2	3	1
Block Fixed-point ILU	1	1.5	3
Block ParILUT	3	1.5	2

Table 5.16: Comparing Block Smoothers (1 best, 3 worst), allowing split scores.

It is clear from these two tables that the different methods have strengths and weaknesses in different areas. It is therefore difficult to say which of these methods is the best. For this one would need to further research how big the advantages or disadvantages are per criteria and one would need to make a choice on which criteria is considered to be most important.

Chapter 6

Conclusion and Further Research

The goal for this research project was to better understand the Block ILUT smoother and to make alterations to it in order to improve efficiency. This was attempted by combining the existing Block ILUT method with methods using fixed-point iterations to construct L, U, giving rise to the Block Fixed-point ILU and Block ParILUT methods. For both new methods several setups have seen investigated.
In Block Fixed-point ILU a particularly good setup is to use sparsity pattern S_1 with 2 parallel factorization steps. For Block ParILUT, S_4 is a good initial sparsity pattern, with threshold $\tau = 10^{-2}$ and again 2 parallel factorization steps.

Both new methods seem like viable alternatives, though it is difficult to compare them with each other or the original Block ILUT, as the methods have strengths in different areas. There is no obvious best method. For instance, the necessary number of multigrid iterations is lowest for the original Block ILUT method, but the other methods score better on computational cost per smoothing step. A longer summary is given in section 5.7.

For further research one could try to make a better comparison between the three methods. Currently these are very hard to compare; nonzero patterns are unpredictable, therefore the cost of the different operations is unpredictable, the theoretical bounds do not give a good indication. Analyzing the computation times of the numerical programs will not help either, they are not optimized enough. I did not have enough knowledge on programming sparse matrix operations or on parallel computing to do this effectively. Big improvements can be made here, it will give for some very interesting results.

Additionally, if one wants to make further alterations to the Block ILUT method, a good field to take inspiration from would be the field of Domain Decomposition methods. In terms of structure our problem with a domain split in multiple patches is very similar to a problem over multiple overlapping domains in Domain Decomposition methods. Problem setting is similar, resulting system matrix is similar, one could use similar tricks. An excellent book to learn about these methods is [24]. A more specific source within domain decomposition suggested by the supervisors, but not actively pursued is [25].

Finally, in [22] an exhaustive list is given of papers where ILU type factorizations are used in combination with parallel computing. Many of those on problems resulting from Domain Decomposition methods. Maybe one or more of those could be a good fit for the problems encountered in IgA.

List of symbols

In order of first appearance (roughly).

Chapter 2: Isogeometric Analysis
u the unknown in the PDE
f the RHS in the PDE

V function space
V_h finite dimensional subspace
ϕ_1,...,ϕ_n basis for V_h

Ω physical domain
Ω_0 parameter domain
$\mathbf{x}$ point in Ω
$\boldsymbol{\xi}$ point in Ω_0
$\mathbf{F}$ function linking physical and parameter domain

A the discretization matrix in the matrix equation
$\mathbf{u}$ the unknown in the matrix equation
$\mathbf{f}$ the RHS in the matrix equation
Can be specified further as $A_{h,p}$, $\mathbf{u}_{h,p}$, $\mathbf{f}_{h,p}$

Ξ, knot vector (if we need to consider multiple knot vectors also H and L)
ξ_i, the i-th knot of Ξ (η_j, ζ_k also as j-th and k-th knot for H and L respectively)

$\phi_{i,p}(\xi)$ the i-th order p basis function to knot vector Ξ
h distance between knots
p order of basis functions
N_{el} number of elements
N_{dof} degrees of freedom, i.e. the amount of basis functions

x_q the quadrature points
w_q the weights of said quadrature points

Ω^k, $\mathbf{F}^k$ multipatch variations of Ω and $\mathbf{F}$
K the number of patches in a multipatch discretization
Γ the interface between the different patches
N_{patch} number of basis functions contained within a single patch
$N_{\text{interface}}$ number of basis functions that interact with the interface

Chapter 3: Multigrid methods
$\mathbf{v}$ approximation to $\mathbf{u}$
$\mathbf{e}$ general error
$\mathbf{r}$ general residual
Can be specified further as $\mathbf{v}_{h,p}$, $\mathbf{e}_{h,p}$, $\mathbf{r}_{h,p}$, etc.

$\mathcal{I}_h^{2h}$ example of intergrid transfer

$\mathcal{I}_{2h}^{h}$ example of intergrid transfer

M mass matrix, $M_{i,j} = \int_\Omega \phi_i(x)\phi_j(x)\ \mathrm{d}\Omega$
M^L lumped mass matrix
P_1^p, P_p^1 transfer matrices used in p-multigrid methods

Chapter 4: ILU type smoothers part I
L, U lower- and upper triangular matrix with $A = LU$
$\tilde{L}, \tilde{U}$ approximations to L and U
$NZ(A)$ set of locations where $a_{ij} \neq 0$
m general threshold, relating to the number of allowed nonzeros
τ general threshold, relating to the allowed minimum size of nonzeros

A_{ii} submatrix of A, located on the diagonal
$A_{i\Gamma}$, $A_{\Gamma i}$ submatrix of A, located on the right column or bottom row respectively
$A_{\Gamma\Gamma}$ the bottom right submatrix of A
L_i, U_i ILUT factorization of A_{ii}, submatrices of L and U in Block ILUT factorizations
B_i, C_i, submatrices of L and U in Block ILUT factorizations
D, E, F, submatrices of L and U in Block ILUT factorizations

Chapter 5: ILU type smoothers part II
S general sparsity pattern
$S_1, ..., S_5$ sample sparsity patterns, defined in section 5.4.2

R ILU residual, $R = A - LU$
m_L, m_U the number of nonzeros added to the sparsity pattern of L and U in ParILUT
m'_L, m'_U the number of nonzeros removed from the sparsity pattern of L and U in ParILUT

Bibliography

[1] Thomas JR Hughes, John A Cottrell, and Yuri Bazilevs. "Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement". In: *Computer methods in applied mechanics and engineering* 194.39-41 (2005), pp. 4135–4195.

[2] J Austin Cottrell, Thomas JR Hughes, and Yuri Bazilevs. *Isogeometric analysis: toward integration of CAD and FEA*. John Wiley & Sons, Sept. 2009. ISBN: 0470748737. DOI: 10.1002/9780470749081.ch7.

[3] R.P.W.M. Tielen. "P-multigrid methods for Isogeometric analysis". English. PhD thesis. Delft University of Technology, 2021. ISBN: 978-94-6366-453-0. DOI: 10.4233/uuid:8445e901-e31e-4602-8630-5b39a3de7ff6.

[4] Roel Tielen et al. *Efficient p-Multigrid Methods for Isogeometric Analysis*. Jan. 2019.

[5] Roel Tielen, Matthias Möller, and Kees Vuik. "A direct projection to low-order level for p-multigrid methods in Isogeometric Analysis". In: *Numerical Mathematics and Advanced Applications ENUMATH 2019*. Springer, 2021, pp. 1001–1009.

[6] Roel Tielen et al. "p-multigrid methods and their comparison to h-multigrid methods within Isogeometric Analysis". In: (2020).

[7] Roel Tielen, Matthias Möller, and Kees Vuik. "A block ILUT smoother for multipatch geometries in Isogeometric Analysis". In: 2020.

[8] Bernd Simeon and Anh-Vu Vuong. *Isogeometric Analysis Primer*.

[9] JJIM van Kan, A Segal, and FJ Vermolen. *Numerical methods in scientific computing*. VSSD, 2005. ISBN: 90-71301-50-8.

[10] SP Venkateshan and Prasanna Swaminathan. *Computational methods in engineering*. Elsevier, 2013. ISBN: 978-0-12-416702-5. DOI: https://doi.org/10.1016/C2012-0-06128-5.

[11] *Gaussian quadrature*. Feb. 2022. URL: https://en.wikipedia.org/wiki/Gaussian_quadrature.

[12] Matthias Möller. "Assembly strategies in isogeometric analysis". In: 2015.

[13] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial*. SIAM, Jan. 2000. ISBN: 978-0-89871-462-3.

[14] C. Vuik and D.J.P. Lahaye. *Course WI4201, Scientific Computing*. 2017.

[15] Nathan Collier et al. "The cost of continuity: performance of iterative solvers on isogeometric finite elements". In: *SIAM Journal on Scientific Computing* 35.2 (2013), A767–A784.

[16] Alvaro Pe de la Riva, Carmen Rodrigo, and Francisco J Gaspar. "A robust multigrid solver for isogeometric analysis based on multiplicative Schwarz smoothers". In: *SIAM Journal on Scientific Computing* 41.5 (2019), S321–S345.

[17] Clemens Hofreither and Stefan Takacs. "Robust multigrid for isogeometric analysis based on stable splittings of spline spaces". In: *SIAM Journal on Numerical Analysis* 55.4 (2017), pp. 2004–2024.

[18] Jarle Sogn and Stefan Takacs. "Robust multigrid solvers for the biharmonic problem in isogeometric analysis". In: *Computers & Mathematics with Applications* 77.1 (2019), pp. 105–124.

[19] Marco Donatelli et al. "Symbol-based multigrid methods for Galerkin B-spline isogeometric analysis". In: *SIAM Journal on Numerical Analysis* 55.1 (2017), pp. 31–62.

[20] Yousef Saad. "ILUT: A dual threshold incomplete LU factorization". In: *Numerical linear algebra with applications* 1.4 (1994), pp. 387–402.

[21] Yousef Saad. *Iterative methods for sparse linear systems*. Second. SIAM, 2003. ISBN: 978-0-89871-534-7. DOI: 10.1137/1.9780898718003.

[22] Edmond Chow and Aftab Patel. “Fine-grained parallel incomplete LU factorization”. In: *SIAM journal on Scientific Computing* 37.2 (2015), pp. C169–C193.

[23] Hartwig Anzt, Edmond Chow, and Jack Dongarra. “ParILUT—a new parallel threshold ILU factorization”. In: *SIAM Journal on Scientific Computing* 40.4 (2018), pp. C503–C519.

[24] Victorita Dolean, Pierre Jolivet, and Frédéric Nataf. *An introduction to domain decomposition methods: algorithms, theory, and parallel implementation*. SIAM, 2015.

[25] Jason Frank and Cornelis Vuik. “On the construction of deflation-based preconditioners”. In: *SIAM Journal on Scientific Computing* 23.2 (2001), pp. 442–462.