Heuristic p-Adaptivity for Discontinuous Galerkin Method

Authors:
Alberto Chiappa, Riccardo Peli

Supervisor:
Luca Bonaventura

April 13, 2018
Contents

1 Introduction 2

2 Problem Statement 3
  2.1 Mathematical Formulation ........................................ 3
  2.2 Numerical Method .................................................. 4

3 Code 5
  3.1 Objectives .............................................................. 5
  3.2 Structure ............................................................... 6
    3.2.1 Domain ............................................................ 6
    3.2.2 Bases ............................................................... 6
    3.2.3 Operators ......................................................... 7
    3.2.4 Finite element problems ....................................... 8
    3.2.5 Polynomial adaptivity ......................................... 8
    3.2.6 Utilities .......................................................... 11
  3.3 Parallelization ...................................................... 12
  3.4 Tests ................................................................. 13

4 Validation of the algorithm 14
  4.1 Qualitative analysis ................................................. 14
  4.2 Convergence .......................................................... 18

5 Results 18
  5.1 Transport and Reaction Simulations ................................ 18
  5.2 Polynomial degrees adaptivity ....................................... 22
    5.2.1 Symmetric adaptivity ......................................... 24
    5.2.2 Asymmetric adaptivity ....................................... 27

6 Conclusions 30

7 Tutorial 31
Abstract

In this project we address the problem of solving diffusion-advection-reaction equations exploiting Discontinuous Galerkin finite element formulation. We limit our scope to simple geometry and rectangular meshes but we offer flexibility in setting the polynomial degrees independently among different elements. Furthermore, it is possible to construct meshes with elements of different sizes. These features allow us to implement polynomial degree adaptivity (p-adaptivity) and we prove its effectiveness in reducing the computational cost of the algorithm. Two different heuristic criteria are proposed.

1 Introduction

Since its introduction in the early 1970s ([1]), the Discontinuous Galerkin (from now on referenced as DG) method has been attracting the interest of the scientific community working with finite element methods because it boasts some very unique features. Indeed, unlike classical formulations of the finite element methods, it involves the use of discontinuous basis functions. In fact, the only requirement for the bases is to be polynomials locally to each element, but jumps at the edges are allowed. This gives independence properties to the different elements, because the intersection of the supports of their bases is empty.

Independence is a highly desirable property when dealing with polynomial degree adaptivity (p-adaptivity). Changing the degree of the bases on an element does not influence the neighbouring elements. Furthermore, independence eases parallel computation.

Hierarchical bases are a natural choice when dealing with p-adaptivity. Let us consider, for instance, Legendre polynomials: they are hierarchical in the sense that, given a set of polynomials up to degree n, the set of degree n+1 is obtained adding a new polynomial without modifying the existing ones. This allows to easily increase or decrease the local polynomial degree on an element, saving most of the computation.

The language of choice for the development of this project was C++. It is a modern object-oriented language that conciliates efficiency with high level of abstraction. An additional plus is the wide availability of high-performance libraries for scientific computing.

The aim of the project is to assess the feasibility of a p-adaptivity approach to reduce the computational demand of a DG algorithm for the resolution of a diffusion-advection-reaction differential problem on a bidimensional rectangular domain. The key features of the code are: it is flexible in setting different polynomial degrees across elements and along different axes on the same element; it can also treat element size in a general way, allowing for nonuniform meshes to be created. We implemented simple adaptivity criteria to test the capabilities of the implementation.

The report is structured as follows:

- firstly the mathematical formulation of the diffusion-advection-reaction problem and its DG discretization are introduced
• the code structure and the most relevant design choices are discussed from a high-level point of view
• the correctness of the code is tested comparing numerical and theoretical results
• finally, some sample problems are solved, exploiting the polynomial adaptivity and highlighting its advantages and limits

2 Problem Statement

In this section we formulate the mathematical problem addressed in the project. The goal is to provide a tool to solve steady advection-diffusion-reaction equations. We exploit discontinuous finite elements with a local adaptation of the basis degree to improve accuracy and reduce the computational time and memory allocation.

2.1 Mathematical Formulation

Given $\Omega \subset \mathbb{R}^2$, the advection-diffusion-reaction equation that we solve writes as follows:

$$\begin{aligned}
\left\{ & -\alpha \nabla^2 u(\mathbf{x}) + \nabla \cdot (\beta u(\mathbf{x})) + \sigma u(\mathbf{x}) = f(\mathbf{x}) \quad \mathbf{x} \in \Omega \\
& u(\mathbf{x}) = 0 \quad \mathbf{x} \in \partial \Omega
\end{aligned}$$  

(1)

with $\alpha, \sigma \in \mathbb{R}^+$, $\beta \in \mathbb{R}^2$ and $f \in L^2(\Omega)$. The solution of the equation is $u$ and it is unknown. From a physical point of view, $u$ can represent, for instance, the equilibrium temperature of a body or the concentration of a pollutant. We refer to [5] for further details on the equation and its physical meaning.

We report the weak formulation of equation 1, as it is necessary for finite element methods

$$\begin{aligned}
\alpha \int_{\Omega} \nabla u \cdot \nabla v - \int_{\Omega} \beta u \cdot \nabla v + \sigma \int_{\Omega} uv &= \int_{\Omega} fv \\
& \forall v \in H^1_0(\Omega)
\end{aligned}$$  

(2)

where $H^1_0(\Omega)$ is a Sobolev Space defined as the space of functions with zero trace on $\partial \Omega$ which belong to $H^1(\Omega)$, where

$$H^1(\Omega) = \left\{ f \in L^2(\Omega) : D^1 f \in L^2(\Omega) \right\}$$  

(3)

$D^1$ denotes the first distributional derivative.

Under the assumptions on the parameters listed above, the weak formulation has a unique solution in $H^1_0(\Omega)$. 

3
2.2 Numerical Method

We exploit a Discontinuous Galerkin (DG) method to solve the equation 1. It is based on a weak formulation and we refer to [4] for its derivation. Here we report just the final formulation implemented in the code.

Given the rectangular domain $\Omega \subset \mathbb{R}^2$, it is subdivided into rectangular disjoint elements $\Omega_m$. We introduce the discrete space $W_\delta$ defined as

$$W_\delta = \{ v_\delta \in L^2(\Omega) : v_\delta|_{\Omega_m} \in Q_r(\Omega_m) \}$$

where $Q_r(\Omega_m)$ is the space of polynomials of degree less or equal to $r$ with respect to each variable. Note that the functions in $W_\delta$ are not necessarily continuous and they can jump through an edge. Defining the union of the internal edges as $E_\delta$, each of them belongs to two elements and we denote with $\text{"+"}$ and $\text{"-"}$ the two different elements. We define

$$[v] = v^+ n^+ + v^- n^-$$

the jump of a function through an edge and its mean gradient. The DG formulation is the following: find $u_\delta \in W_\delta$ such that

$$\alpha \sum_{m=1}^{M} \int_{\Omega_m} \nabla u_\delta \cdot \nabla v_\delta - \alpha \sum_{e \in E_\delta} [v] \cdot \{ \nabla u_\delta \} - \tau \alpha \sum_{e \in E_\delta} [u_\delta] \cdot \{ \nabla v_\delta \} + \gamma \sum_{e \in E_\delta} |e|^{-1} \int_{e} [v_\delta] \cdot [v_\delta]$$

$$- \sum_{m=1}^{M} \int_{\Omega_m} \beta u_\delta \cdot \nabla v_\delta + \sum_{e \in E_\delta} \{ \beta u_\delta \} \cdot [v_\delta] + \sigma \sum_{m=1}^{M} \int_{\Omega_m} u_\delta v_\delta - \alpha \sum_{e \subseteq \partial \Omega} \int_{e} v_\delta \nabla u_\delta \cdot n$$

$$- \tau \alpha \sum_{e \subseteq \partial \Omega} \int_{e} u_\delta \nabla v_\delta \cdot n + \gamma \sum_{e \subseteq \partial \Omega} |e|^{-1} \int_{e} u_\delta v_\delta + \sum_{e \subseteq \partial \Omega_{\text{inflow}}} \int_{e} \beta \cdot n u_\delta v_\delta = \sum_{m=1}^{M} \int_{\Omega_m} f v_\delta \quad \forall v_\delta \in W_\delta$$

The formulation above is composed of several operators, deriving from the weak formulation, localised to each element. There are operators which act on an element and others on an edge. In particular, the operators which involve an integral over an element are the operators present in the weak formulation 2, but split on the elements.

On the other hand, the edge operators involve the jump or the mean of the solution (and
its gradient) and of the test functions through the edges of $\mathcal{E}_3$. They appear because of the integration by part localised on each element and they do not vanish because the considered functions are not a-priori continuous.

The operators multiplied by $\tau$ and $\gamma$ are added for numerical reasons. More specifically, the one multiplied by $\tau$ is the symmetric version of the previous one and, in the case of pure diffusion-reaction, it can symmetrize the matrix of the linear system (if $\tau$ is 1). Furthermore, the parameter $\tau$ influences the stability and the convergence rate of the algorithm. The operator multiplied by $\gamma$ is a stabiliser and it penalises the jumps of the solution, preventing the appearance of spurious oscillations. Adding these two operators preserves the strong consistency of the numerical scheme, i.e. the exact solution still satisfies the formulation.

Concerning the transport terms, the edge operator considers the upwind flux (5), getting the information from the element which is upstream. This approach makes the formulation stable. The boundary conditions are imposed in a weak sense, i.e. penalising the discrepancy between $u$ and the zero function on $\partial \Omega$. So, the solution $u_3$ is not fixed to zero on the boundary (as it happens imposing the boundary conditions in the strong form) and if the penalisation is not large enough, the solution will be far from zero on the boundary.

3 Code

This section is dedicated to the description of the code and to the justification of the technical choices that we made while writing it. We do not want it to serve as a documentation: it is available as a separate .pdf file dealing with each class and method in detail.

3.1 Objectives

Before the beginning of our work, the Mathematics Department of Politecnico di Milano already owned a code to solve differential problems with the Discontinuous Galerkin method; however, it was written in Fortran, with a procedural programming style. We were asked to build a new piece of software, inspired by the already available one, that made use of the object-oriented programming technique. Additionally, it should be able to manage nonuniform polynomial degree on the different elements of the mesh, in order for different polynomial adaptation techniques to be supported.

Our language of choice was C++. It has great object oriented capabilities and boasts the high efficiency of compiled languages. Moreover, it is widely used in high-performance computing: a variety of highly optimised linear algebra libraries are available.

Thanks to the intrinsic properties of object-oriented programming, the code is very modular and its fundamental parts can be subdivided in sections. In the following paragraphs, a description of each of them will be provided. For a detailed documentation of the interface and for the source code, see the Doxygen documentation.
3.2 Structure

3.2.1 Domain

The necessary first step when writing a finite elements code is to define the classes that model the domain on which the differential problem is defined. We chose to limit our scope to rectangular domains. This leads to the natural choice of discretizing them with a quadrangular mesh. The advantages of this approach are many. First of all, the definition and the memorisation of the domain are simple; but most importantly, the basis polynomials defined on each element are separable, which means that adaptivity can be performed independently along the two axes. Finally, 1-dimensional quadrature formulas can be easily extended to a 2-d domain if it is a rectangle.

The physical domain translates into the QuadMesh class. It is responsible for both storing the geometric properties of the mesh, by means of collections of Node and QuadElement objects, and providing the necessary methods to access this information. Furthermore, an Edge class models the boundaries of each mesh element.

The physical domain is closely related to the finite element space, as it defines the properties of the basis functions over each element. In order to keep the interface as simple as possible and not to expose to the user too much about the internal implementation, we decided to use containment to connect FeSpace and QuadMesh classes. Thus, QuadMesh is an attribute of FeSpace and most of the information it contains can be accessed through the FeSpaceElement class. In fact, vector of fespace elements is also an attribute of FeSpace: they not only store the information about the maximum degree of the basis polynomials along each axis, but they also offer an interface to the most recurrent methods of the QuadMesh class, like the getters for the domain size and coordinates. This way the user only has to access the QuadMesh object only when he needs to know some global information, like the elements in the neighbourhood of another one.

3.2.2 Bases

When choosing which basis to use for a differential problem, the most popular alternatives are two: a nodal basis or a modal basis. Nodal bases are also known as Lagrangian interpolant polynomials. They have the property that each polynomial takes value 1 on exactly one node and 0 on all the others. This means that the projection coefficient relative to, e.g., the basis \( i \), which has value 1 on node \( i \), is the value of the whole projection on the mentioned node. This fact establishes a 1-to-1 relationship between the nodes and the projection coefficients, allowing for an immediate pointwise evaluation of the solution.

Lagrangian interpolants, however, have one major drawback when polynomial adaptivity comes into play: changing the polynomial degree on one element means having to redefine all the basis functions, as they are totally different. This is the reason why we chose to use modal bases for the resolution of the problems, despite also defining nodal bases for future extensions of the
As far as the code is concerned, because of the presence of differential operators, also the bases derivatives have to be defined. Both bases and their derivatives share the same interface: they implement a method that, given a vector of nodes, evaluates the basis functions over them. This structure can be translated into the code defining a class hierarchy with inheritance. We defined a base Basis class, whose evaluate method is virtual, that forces all its children to implement it. This design allowed us to reuse large parts of the code multiple times, especially in the operator definitions. In fact, most of them differ one from the other for the kind of basis used. For instance, combining a derived basis for both the solution and the test function we obtain a diffusion operator, while considering their non-derived version we define a reaction operator.

### 3.2.3 Operators

As just mentioned, using inheritance to define the different bases was the key to reuse most of the code of the Operator class. What we did was to create an abstract Operator class, whose only purpose was to provide an interface for the specific operators. It declares two pure virtual methods, one that is be used by the specialised classes to create the local matrix for the reference element, and the other one to update the global matrix. However, the specialised operators do not directly inherit from the base Operator class. Indeed, operators can be further grouped into two families: the ones that are defined inside the element and the ones that are defined on its edges. This reflects into two intermediate classes, ElementOperator and EdgeOperator. Figure 2 visually shows the hierarchy of the operator classes. Both ElementOperator and EdgeOperator contain the implementation of the creation of the local matrix, that is almost identical for operators of the same kind. The only difference is the choice of the basis (if to use the basis polynomials or their derivatives) and some operator-specific coefficients, such as the diffusion coefficient, the reaction coefficient and the transport coefficient. In order to keep them as general as possible, and to allow for the code to be extended in the future with new operators with possibly non-constant coefficients, we decided to delegate the coefficient definition to separate class methods. This way, by defining a subclass of an existing operator that overrides the method that returns the coefficient, it is possible to create entirely new operators with non-constant coefficients.
3.2.4 Finite element problems

A consequence of how the operators are defined is the role that the Problem classes act in our code: they are simply collections of operators. As all the specific operators inherit from the same Operator base class, differential problems can be defined as a collection of operators, whose updateSystemMatrix methods are called one by one on a SystemMatrix object owned by the Problem class. This very general implementation of a problem relies on the fact that each operator can update the global system matrix independently.

The Problem class does not have to care for an efficient implementation of the system matrix assembly. This aspect of the algorithm is delegated to the SystemMatrix class. Its public interface, consisting of the methods addBlock, buildSparseMatrix and getMatrix, makes sure that memory and access efficiency are guaranteed. In fact, until the buildSparseMatrix method is called, all the elements are stored as a vector of triplets, indicating the position and the value of each element. This format is the one that guarantees the best efficiency when it is passed to an Eigen sparse matrix class for construction. We have chosen the Eigen library to run all the matrix operations as it provides an easy interface, it has the advantages of an header-only library and it provides very efficient implementations of some of the most popular linear algebra algorithms, including solvers for sparse systems. However, we do not exclude that for some specific problem it would be necessary, in the future, to use a different library to solve the system. Such a change would not have a huge impact on our code, as it would limit the modifications to the SystemMatrix class, which would have to adapt the matrix assembly, and to the problem class, which would have to call a suitable solver for the new matrix class.

3.2.5 Polynomial adaptivity

The main distinguishing feature of our implementation of the discontinuous Galerkin finite elements implementation is its possibility to accept different maximum polynomial degree on its elements, and to adapt the system matrix accordingly. The idea behind polynomial adaptivity is that, given a source term characterized by a regular enough function, increasing the local polynomial degree on the elements where the approximation is bad improves the overall accuracy. The adaptation process can be lead by different criteria. A popular choice is to use an a-posteriori estimator for the discretization error. This is, however, problem-dependent and not always available. A quite general approach would be to implement a solver that imitates the iterative approach of multigrid for mesh adaptation. The idea is to start with a low-degree approximation of the solution, identify the elements on which the error is larger and increase
the polynomial degree on them. This is, however, beyond the scope of this project.
We decided to keep it simple on the Mathematical side, as this is before anything else a pro-
gramming project. We implemented two heuristic methods, that work fairly well for diffusion
problems and provide relevant test cases for our software architecture. The implementation
of the two methods is largely inspired by the works [7] and [8], who first proposed such an
approach.
Both methods tune the polynomial degree on each element based on the supposed complexity
of the solution. In fact, diffusion is a symmetric operator, whose effect is to smooth the source
term, without moving the high gradients to some other parts of the domain. Furthermore, dif-
fusion alone makes the transitions less abrupt, easing the numerical approximation. However,
when the source term presents some local complexity, an adaptation is still beneficial, as we
will show in the section dedicated to the results.
Both the implemented methods work as follows:

- Compute a high-degree approximation of the source term.
- For each element of the domain, compute a low-degree approximation of the source term.
- Compare the low-degree approximation to the high-degree one. If the norm of their differ-
ence is larger than a fixed tolerance, increase the degree of the low-degree approximation
until it satisfies the tolerance.

The only difference between the two heuristic methods is that, while the first one increases the
degree along both x and y simultaneously, the second one checks at each iteration if it is more
beneficial to increase the degree along x or y. This makes the second method more suitable for
domain with stretched elements or for highly asymmetric source terms.
Here we report the core code that implements the two aforementioned symmetric and asym-
metric criteria.

```c
//
// Symmetric adaptivity criterion
//
double totalSquaredNorm = localProjection.squaredNorm();
double partialSquaredNorm = localProjection[0]*localProjection[0];
int curDegree = 0;
while (partialSquaredNorm < (1-tol)*totalSquaredNorm) {
    curDegree++;
    for (int j = 0; j < curDegree; ++j) {
        partialSquaredNorm += localProjection[curDegree*(maxDegree+1) + j] *
                              localProjection[curDegree*(maxDegree+1) + j];
        partialSquaredNorm += localProjection[j*(maxDegree+1) + curDegree] *
                              localProjection[j*(maxDegree+1) + curDegree];
    }
}```
partialSquaredNorm += localProjection[(curDegree)*(maxDegree+2)] * 
localProjection[(curDegree)*(maxDegree+2)];
}

// // Asymmetric adaptivity criterion //

double totalSquaredNorm = localProjection.squaredNorm();
double partialSquaredNorm = localProjection[0] * localProjection[0]
 + localProjection[maxDegree + 1] * 
localProjection[maxDegree + 1]
 + localProjection[maxDegree + 2] * 
localProjection[maxDegree + 2];

int curDegreeX = 1;
int curDegreeY = 1;

while (partialSquaredNorm < (1 - tol) * totalSquaredNorm) {
    double partialSquaredNormIncreasingX = partialSquaredNorm;
    double partialSquaredNormIncreasingY = partialSquaredNorm;

    if (curDegreeX < maxDegree) {
        // What happens if we increase x
        for (int j = 0; j < curDegreeY + 1; ++j) {
            partialSquaredNormIncreasingX += localProjection[(curDegreeX +
1) * (maxDegree + 1) + j] * localProjection[(curDegreeX + 1)
* (maxDegree + 1) + j];
        }
    }

    if (curDegreeY < maxDegree) {
        // What happens if we increase y
        for (int j = 0; j < curDegreeX + 1; ++j) {
            partialSquaredNormIncreasingY +=
localProjection[j * (maxDegree + 1) + curDegreeY + 1] *
localProjection[j * (maxDegree + 1) + curDegreeY + 1];
        }
    }

    // Increase the degree that brings more benefit
    if (partialSquaredNormIncreasingX > partialSquaredNormIncreasingY) {
        partialSquaredNorm = partialSquaredNormIncreasingX;
        curDegreeX++;
    } else {
        partialSquaredNorm = partialSquaredNormIncreasingY;
        curDegreeY++;
    }
}
3.2.6 Utilities

This group includes classes that do not have much in common, apart from the fact that they are used in a variety of contexts. In fact, they implement some useful functions (or small sets of functions) that are frequently used throughout the code. Here follows a comprehensive list of them:

- **GaussLobattoQuadrature.** Implements the Gauss-Lobatto quadrature rule, which is the interpolant quadrature rule with the maximum accuracy degree among the quadrature rules that use the domain boundaries as nodes. It is the most convenient quadrature rule for our case, as the basis evaluation always includes the element boundary, and most of the integrations are performed on the basis polynomials.

- **Error.** It accepts a 2-D function and the coefficients of the projection of a function onto a finite elements space and can compute the L2 or the energy norm of the difference. The class could be easily extended by implementing the computation of the errors with different norms, adding the relative method to the class. Convergence estimates for Discontinuous Galerkin methods come in the two available norms, thus we did not go further.

- **FunctionProjection.** Accepts a 2-D function and computes its projection on a bidimensional separable basis. It uses the quadrature formula defined in GaussLobattoQuadrature.

- **MatrixOperations.** Defines some useful operations between vectors and matrices that are not implemented in Eigen by default. They are vector-vector elementwise product and matrix-matrix tensor product.

- **SolutionEvaluation.** Given two vector of coordinates, the coefficients of a function on the elements of a finite element space and the finite element space itself, this class implements the pointwise evaluation of the said function on the grid generated by the cartesian product of the two coordinate vectors. It is very useful to plot the solution of a problem.

- **IntegralMatrix1D.** Accepts two bases and generates a matrix whose elements are the integrals of all the possible pairs of polynomials in which the first is chosen from one basis and the second from the other basis. It is mainly useful when computing the contribution of each element to the global system matrix. In fact, its contribution is the cartesian product of two matrices generated by IntegralMatrix1D.
3.3 Parallelization

The assembly of the system matrix is a very computationally expensive task. For small problems it takes longer than the solution of the linear system. We therefore saw the possibility to speedup the whole execution by parallelizing the creation of the local matrices relative to the different elements. In fact, these computations happen independently in a for cycle over all the elements of the mesh, which makes it suitable to be run in parallel with OpenMP. We only had to make sure that the updates to the vector of triplets necessary for the creation of the global matrix happened atomically.

Figure 3 shows the performance gain obtained by using multithreading in matrix creation. It shows the ratio between the execution time with n threads and the execution time with a single thread. Surprisingly, the speedup when using two cores is superlinear: our hypothesis is that a very important role in this section of the algorithm is played by communications with memory and the larger available cache when exploiting 2 cores boosts the performance. In the table 1 we report the execution times. The simulations were run on a laptop Dell XPS-15, Intel Quad Core I7 with 8 GB of RAM.
Table 1: Absolute execution time [s]

<table>
<thead>
<tr>
<th></th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 elements</td>
<td>0.122</td>
<td>0.071</td>
<td>0.051</td>
</tr>
<tr>
<td>32 elements</td>
<td>2.041</td>
<td>1.073</td>
<td>0.072</td>
</tr>
<tr>
<td>64 elements</td>
<td>54.12</td>
<td>17.14</td>
<td>10.18</td>
</tr>
<tr>
<td>128 elements</td>
<td>904.63</td>
<td>373.24</td>
<td>245.98</td>
</tr>
</tbody>
</table>

3.4 Tests

Creating a finite elements code from scratch for the first time presents two main challenges: first, the final structure of the code is not entirely clear from the beginning; secondly, if the first version of the complete code does not work it is not trivial to find out where the errors are. Luckily, most of the components are quite independent from each other. For instance, integration, bases, meshes, operators, despite interacting with each other, make sense also if they are treated independently. This makes them perfectly suitable for unit testing. Unit testing eases refactoring and helps finding errors early, which was really helpful during the development.

As a framework for testing we chose Google Test, because it is modern and portable. Bases, mesh, quadrature and some integration functionalities, like projection of a function on a basis, were tested. This allowed us to keep sure that no regression hits the basic part of the code and we could focus on fixing bugs in the higher level classes, such as the operators.
4 Validation of the algorithm

In this section we assess the correctness of the algorithm evaluating the results, firstly in a qualitative way and then comparing the errors with the theoretical estimations.

Let us considering the following problem

\[
\begin{cases}
    -\nabla^2 u(x,y) = f(x,y) & (x,y) \in \Omega \\
    u(x,y) = 0 & (x,y) \in \partial \Omega 
\end{cases}
\]  

(6)

where \( f(x,y) = e^{3x} \sin(2\pi y)((9-4\pi^2)x^2+(3+4\pi^2)x-4) \) and the analytical solution is \( u(x,y) = (x - x^2)e^{3x} \sin(2\pi y) \).

The numerical formulation of the problem is the simplified version of 4, getting rid of the transport and the reaction. Depending on the value of \( \tau \), we introduce three formulations, which exhibit different stability and convergence properties. In particular

- \( \tau = 1 \), SIPG (Symmetric Interior Penalty Galerkin)
- \( \tau = 0 \), IIPG (Incomplete Interior Penalty Galerkin)
- \( \tau = -1 \), NIPG (Non-symmetric Interior Penalty Galerkin)

NIPG is stable \( \forall \gamma > 0 \), whereas SIPG and IIPG are conditionally stable, i.e. they require \( \gamma \) sufficiently large.

4.1 Qualitative analysis

We now show the plots of the solutions for specific values of parameters, confirming the stability or instability of the different numerical methods introduced in the previous section. In particular, as clearly visible in figures 4, 5 and 6, all the methods are stable for \( \gamma = 10 \) and they exhibit a good approximation of the exact solution. On the hand, choosing a small \( \gamma \), i.e. \( 10^{-4} \), only the NIPG (\( \tau = -1 \)) method shows similar results as in the previous case but the solution is not smooth, as the jumps between the elements are just slightly penalised. The SIPG (\( \tau = 1 \)) method is unstable, showing oscillations comparable to the magnitude of the solution whereas the IIPG (\( \tau = 0 \)) method does not exhibit oscillations but the values of the solution considerably increases (note the scale).
Figure 4: Solutions for $\tau = 1$ and $\gamma = 10$ (above) and $\gamma = 10^{-4}$ (below)
Figure 5: Solutions for $\tau = 0$ and $\gamma = 10$ (above) and $\gamma = 10^{-4}$ (below)
Figure 6: Solutions for $\tau = -1$ and $\gamma = 10$ (above) and $\gamma = 10^{-4}$ (below)
4.2 Convergence

At this point, we validate our algorithm comparing its results with the theoretical estimations of the error. We refer to [4] for further details.

We define the $L^2$-norm as
\[ \|u\|_{L^2(\Omega)} = \sqrt{\int_{\Omega} u^2} \]

and the energy norm
\[ \|u\| = \sqrt{\sum_{m=1}^{M} \int_{\Omega_m} |\nabla u_\delta|^2 + \sum_{e \in \mathcal{E}} \gamma_e |e|^{-1} \int_e |u_\delta|^2 + \sum_{e \subset \partial \Omega} \gamma_e |e|^{-1} \int_e |u_\delta|^2} \]

Defining $r$ as the degree of the basis (the same on all the elements) and $h$ as the length of the edge of the elements (in this case, we use squared elements), the following a-priori error estimation holds
\[ h\|u - u_\delta\| + \|u - u_\delta\|_{L^2(\Omega)} \leq C h^{r+1} |u|_{H^{r+1}(\Omega)} \]

This estimate requires the solution $u$ to be sufficiently regular ($u \in H^{r+1}(\Omega)$) and it establishes a minimum convergence rate of the errors, letting $h$ goes to 0. The estimate 7 holds for the SIPG method, whereas it is valid only for regular meshes and odd polynomial degrees in the cases of NIPG and IIPG methods (see [2]). In figures 7, 8 and 9 we show the errors obtained in our simulations. The results agree with the theory, showing optimal convergence for all the methods with odd polynomial degree. In the case of an even degree, only the SIPG method exhibit optimal convergence rate (in particular the $L^2$ norm), whereas NIPG and IIPG shows suboptimal convergence.

Changing the point view, in figure 10 we report the behaviour of the error varying the polynomial degree with fixed mesh refinement. As expected, we observe an exponential decreasing since using the log scale along $y$ results in a straight line.

5 Results

5.1 Transport and Reaction Simulations

In this section we show additional results of the algorithm, focusing on the general case of an advection-diffusion-reaction equation
\[ \begin{cases} -\nabla^2 u(x, y) + \nabla \cdot (\beta u(x, y)) + \sigma u(x, y) = f(x, y) & (x, y) \in \Omega \\ u(x, y) = 0 & (x, y) \in \partial \Omega \end{cases} \]

We take $f(x, y) = -2(x(x-1) + y(y-1))$ and we pick some values of the transport field and the reaction coefficient in order to qualitatively describe the results. Firstly, we evaluate the effects
Figure 7: Rate convergence for NIPG ($\tau = -1$) varying the mesh refinement
Figure 8: Rate convergence for IIPG ($\tau = 0$) varying the mesh refinement
Figure 9: Rate convergence for SIPG ($\tau = 1$) varying the mesh refinement
of the advection term. Referring to figure 11 we note that when a dominating field (\( \beta \) much bigger than \( \alpha \)) is present, a boundary layer appears and the solution has a sharp transition to zero in correspondence of the outflow part of \( \partial \Omega \), i.e. where \( \beta \cdot n > 0 \). In the case of an almost hyperbolic equation (figure 11 (b)), the boundary condition at the outflow is neglected, so the boundary layer is not present in the result. The solution does not exhibit oscillations even in this extreme case.

On the other hand, the reaction introduces an isotropic factor, modifying the magnitude and the shape of the solution. In particular, a positive reaction decreases the absolute value of the solution but it does not cause numerical issues, as shown in figure 12.

### 5.2 Polynomial degrees adaptivity

As previously mentioned when describing the code, we implemented two different polynomial adaptation criteria, based on the shape of the source term. When the function presents very high peaks with large gradients in certain parts of the domain, increasing the polynomial degree locally has beneficial effects on the approximation error. The advantage of this approach is that achieving the same levels of accuracy with a uniform degree would require the solution of a much larger system.

As a test case for our adaptivity approach we considered a purely diffusive problem whose solution is a Gaussian curve with very high mass concentration at the centre of the domain. The computation of the source term that generates such solution can be performed by hand,
Figure 11: Numerical solutions for two different values of the transport vector
Parameters: $h = 1/50, r = 2, \alpha = 1, \sigma = 0, \gamma = 1, \tau = -1$

Figure 12: Numerical solutions for two different values of the reaction coefficient
Parameters: $h = 1/50, r = 2, \alpha = 1, \beta = (0, 0)^T, \gamma = 1, \tau = -1$
which allowed us to compute the exact discretization error.
In order to evaluate the performance improvement brought by the adaptivity criterion we pro-
ceeded as follows: we first chose the number of elements and tuned the uniform degree so that
the approximation is visually satisfactory and the error relatively low; then we picked the adap-
tation parameters to obtain an approximation error as close as possible to the one obtained
with uniform degrees. At this point we compared the size of the system that was solved in the
two cases.

5.2.1 Symmetric adaptivity
The algorithm is entirely heuristic and is based on the assumption that the source term has a
similar shape to the solution of the differential problem. This hypothesis is satisfied only when
the transport term is very small, better if not present at all, and the diffusion is not too strong
either. In this specific scenario we can empirically demonstrate that the adaptation works fairly
well and fewer elements are necessary to achieve comparable accuracy to the solution obtained
using uniform polynomial degrees.
The algorithm works as follows:

1 Set a tolerance and a maximum degree.

2 Iterate through the elements of the mesh.

3 On each element compute the projection of the source term on the finite element space,
considering a local degree equal to the maximum degree. Also compute the norm of this
projection. We call the norm of the projection on the finite element space with maximum
degree $P_{MD}$

4 Set a temporary local degree to 1.

5 Compute the norm of the projection on the subspace of the finite element space of degree
equal to the temporary degree. We call it $P_{TD}$

6 If $P_{TD} > (1 - tol)P_{MD}$ or the temporary degree is equal to the maximum degree, the final
local degree is set to current local degree and the algorithm proceeds to the next element.
Otherwise increase the local degree and return to step 5

We tested this algorithm on a problem whose solution is a Gaussian with a very high concen-
tration at the centre of the domain, as we set $\sigma = 10^{-2}$ (9), multiplied by some factors that
make sure it respects the Dirichlet homogeneous boundary conditions (10).

$$g(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left[\frac{(x - 0.5)^2 + (y - 0.5)^2}{2\sigma^2}\right]$$

(9)
\[ u(x, y, \sigma) = xy(x - 1)(y - 1)g(x, y, \sigma) \] (10)

The corresponding source term is given by 12, whose expression is split into a few functions for clarity.

\[ h(x) = \left( -2 - \frac{1}{\sigma^2} (3x - 0.5 - 3x^2) - \frac{x - 0.5}{\sigma^2} \left( 1 - 2x - \frac{1}{\sigma^2} (1.5x^2 - 0.5x - x^3) \right) \right) \] (11)

\[ f(x, y, \sigma) = (y^2 - y)g(x, y, \sigma)h(x) + (x^2 - x)g(y, x, \sigma)h(y) \] (12)

Figures (14) and (15) show these two functions. The algorithm was able to detected that the complexity is located at the centre of the domain and selected the polynomial degrees accordingly, as showed in Figure (13). Close to the boundary, where the source term is flat, degree 1 was enough to satisfy the tolerance, which was set to \(5 \times 10^{-3}\), while a higher degree was required elsewhere. The approximation error with respect to the L2 norm is equal to \(1.01 \times 10^{-2}\) when the degree is set to 5 everywhere and equal to \(1.06 \times 10^{-2}\) if the adaptation with maximum degree 5 is used. The slightly lower precision has the advantage of reducing the linear size of the global matrix from 36864 to 12796, greatly reducing the computational complexity of the problem.

In figure 16 we show how the number of unknowns and the \(L^2\) error are influenced by the tolerance. The number of elements of the domain is \(32 \times 32\). We observe that when the tolerance is set to a low value (for instance \(10^{-4}\)) the error obtained with the adaptivity is identical to the one got with an uniform basis degree, but the gain in terms of number of unknowns is significant. We can also notice that the decrease in the approximation error happens abruptly when the tolerance parameter becomes small enough. This can be explained with the fact that most of the "mass" of the solution is concentrated on a small number of elements. When the criterion recognises that the degree has to be maximised on those elements the approximation error becomes almost the same as if the maximum degree was chosen everywhere. Therefore, although it is theoretically possible to get a big speedup due to the smaller number of elements, it is in practice difficult to select a proper value for the tolerance.
Figure 13: Distribution of the polynomial degrees after adaptation. The distribution can be interpreted thanks to figure 14: it is flat close to the boundary, where a low degree is enough to approximate the source term. The degree is then maximum close where the gradients are high and slightly decreases at the center of the domain.

Figure 14: Source term used to test the symmetric polynomial adaptivity, viewed from different perspectives. Despite the corresponding solution being a Gaussian, we can notice that the source term is characterised by a negative elongation around the central peak.
Figure 15: Gaussian curve with a high concentration at the centre of the domain.

5.2.2 Asymmetric adaptivity

(a) Source term  
(b) Analytical solution

Figure 17: Source term and corresponding solution used to test the asymmetric polynomial degree adaptation. Please observe that 17b is the same function as 17a rescaled by a multiplicative factor.
Figure 16: Number of unknowns and error varying the tolerance of the symmetric algorithm. Both the quantities are normalized, i.e. divided by their value obtained in a simulation with uniform degree equal to the maximum degree.
Figure 18: Distribution on the domain of the adapted polynomial degrees. The elements had
been stretched along the direction of the $y$ axis, so we can see how the adaptation leads to
higher degrees along that axis. The peculiar distribution of the degrees of the polynomials
along $x$ is due to the oscillations of the source term.

The asymmetric adaptivity algorithm is very closely related to its symmetric counterpart, with
a single major difference: at each step of the comparison between the norm of the low-degree
projection to the one of max-degree projection, the degrees of the polynomials along $x$ and $y$
axes are increased independently. This means that the algorithm can check at each step if the
source term function is better approximated increasing the polynomial degree along $x$ or $y$ and
update the local degree accordingly. The algorithm can be summarized as follows:

1 Set a tolerance and a maximum degree.

2 Iterate through the elements of the mesh.

3 On each element compute the projection of the source term on the finite element space,
   considering a local degree equal to the maximum degree. Also compute the norm of this
   projection. We call the norm of the projection on the finite element space with maximum
   degree $P_{MD}$

4 Set a temporary local degree to 1 on both $x$ and $y$.

5 Compute the norm of the projection on the subspace of the finite element space of degree
   equal to the temporary degree on each axis. We call it $P_{TD}$

6 If $P_{TD} > (1 - tol)P_{MD}$ or the temporary degree is equal to the maximum degree on both
   axes, the final local degree is set to current local degree and the algorithm proceeds to
   the next element. Otherwise proceed with the next step.
Compute the norms of the two projections $P_{TD,x}$ and $P_{TD,y}$ obtained by increasing the local degree on $x$ and $y$, respectively. Compare the two values: if $P_{TD,x} \geq P_{TD,y}$ and the temporary degree along $x$ is smaller than the maximum one, increase the local degree along $x$. Otherwise, increase it along $y$. Return to 5.

This approach is especially beneficial when the solution is asymmetric and presents higher complexity along one axis, or when the asymmetry is in the grid. For instance, we tested the asymmetric adaptivity on a high frequency sinusoidal function, solving the diffusion problem on a grid with 32 elements along $y$ and only 8 elements along $x$. The analytical expression of the source term is

$$f(x, y) = 200\pi^2 \sin(10\pi x) \sin(10\pi y)$$

and the corresponding solution is

$$u(x, y) = \sin(10\pi x) \sin(10\pi y)$$

In this case the heuristic approach works particularly well, because the solution has exactly the same shape of the source term, only rescaled by a factor. Figure (17) represents graphically the source term and the analytical solution to the problem.

The adaptation process leads to the selection of higher polynomial degrees along $x$, as fewer elements have to approximate the same complexity as along the other axis. The local degrees are shown in figure (18). The approximation error with respect to the L2 norm is equal to $4.52 \times 10^{-3}$ when the degree is set to 4 everywhere and equal to $4.60 \times 10^{-3}$ if the adaptation with maximum degree 4, with a tolerance equal to $10^{-5}$, is used. Again, a loss in the precision is traded to reduce the linear size of the global matrix from 6400 to 4480, noticeably reducing the computational complexity of the problem.

6 Conclusions

In this work we proved the effectiveness of our code in exploiting $p$-adaptivity in order to reduce the needed computational resources, maintaining the same accuracy. The proposed tool could be extended to deal with more general equations (for instance time dependent equations), boundary conditions (not homogeneous Dirichlet, Neumann or Robin conditions) or domain geometries (not rectangular domains).

In the framework of the $p$-adaptivity, more advanced methods have to be used when a transport term is present, because the shape of the solution can be very different from the source term in such a situation. A multigrid algorithm [6] can be a valuable approach for a general $p$-adaptivity in the contest of steady advection-diffusion-reaction equations.
7 Tutorial

In this tutorial we provide a short guide which describes the usage of the code. After the compilation (see README.txt for a detailed explanation), it is possible to run three executable which correspond to different samples problems: diffusion, diffusion advection reaction and p adaptivity. The first solves the laplace equation 6 and it is the problem used for the check of the convergence, the second solves a diffusion-advection-reaction equation using the data stated in 8 and the last one performs symmetric p-adaptivity on the laplace equation with source term 13. Each problem takes as input the correspondent parameters file in the folder input.

In order to define a new problem the main.cpp and the file input/parameters.txt should be modified. In the main it is possible to define the source term \( f \) and the analytical solution (if known). If the last one is defined, in order to measure the error, the two corresponding lines of code in the main function have to be uncommented. On the other hand, all the other parameters are defined in the parameters.txt file and it can be modified to define a different equations or to modify the numerical scheme.

Once the equation is solved, it is possible to plot the solution and the elements degrees exploiting Gnuplot with following commands:

- open Gnuplot terminal
- type `load 'plot_solution.plt'` to plot the solution
- type `load 'plot_degrees.plt'` to plot the element degrees resulting from the chosen adaptivity criterion (NONE, SYMMETRIC or ASYMMETRIC)
References


