A thread parallel implementation of the equilibrated flux in Julia

François Févotte, Ari Rappaport

To cite this version:

François Févotte, Ari Rappaport. A thread parallel implementation of the equilibrated flux in Julia. 2024. hal-04537446

HAL Id: hal-04537446
https://hal.science/hal-04537446

Preprint submitted on 8 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A thread parallel implementation of the equilibrated flux in Julia

François Févotte[*] Ari Rappaport[†‡]

April 8, 2024

Abstract

In this article, we present a novel implementation of the equilibrated flux, the key ingredient in designing a p-robust error estimator. Our algorithm involves two primary loops: one on mesh cells and one on nodal patches. The loop on cells provides a setup phase that allows to reduce memory allocations for the potentially more costly loop on patches. Both of these loops involve mutually independent operations and are therefore amenable to parallelization strategies. We present numerical results of both the correctness of our implementation as well as evidence of the speedup due to parallelism. Our implementation is available as a registered Julia package under an MIT open source license.

Keywords: finite elements, a posteriori error estimation, shared-memory parallelism, Julia programming language

1 Introduction

A posteriori error estimation has become an indispensable tool in numerical analysis. However, many popular error estimators such as the residual based estimator [22] or the so-called ZZ-estimator [24, 23] are not robust with respect to the polynomial degree of the discretization. The equilibrated flux is the key ingredient in designing an error estimator that is independent of the polynomial degree p related to the discretization (p-robust). In this paper we present an efficient parallel implementation of the equilibrated flux: a vector field $\boldsymbol{\sigma}_h \in \boldsymbol{H}(\mathrm{div}, \Omega)$ with a prescribed divergence where $\Omega \subset \mathbb{R}^d$ is a spatial domain in dimension $d \in \{1, 2, 3\}$. We consider a fitted conforming mesh $\mathcal{T}_h$ of Ω with vertex set $\mathcal{V}_h$. For each vertex $\mathbf{a} \in \mathcal{V}_h$, the algorithm consists in solving local minimization problems of the form

$$\boldsymbol{\sigma}_h^{\mathbf{a}} := \underset{\substack{\boldsymbol{v}_h \in \boldsymbol{V}_h^{\mathbf{a}} \\ \nabla\cdot\boldsymbol{v}_h = g^{\mathbf{a}}}}{\arg\min} \|\boldsymbol{\tau}_h^{\mathbf{a}} + \boldsymbol{v}_h\|_{\omega_{\mathbf{a}}}, \tag{1}$$

where $\omega_{\mathbf{a}} \subset \Omega$ is the domain corresponding to a small subset of elements in $\mathcal{T}_h$ (a nodal patch), $\boldsymbol{V}_h^{\mathbf{a}}$ is a finite-dimensional $\boldsymbol{H}(\mathrm{div}, \omega_{\mathbf{a}})$-conforming space, $-\boldsymbol{\tau}_h^{\mathbf{a}}$ is the local target vector field, and $g^{\mathbf{a}}$ is the local prescribed divergence. The global equilibrated flux is then built by summing over the local contributions, i.e.,

$$\boldsymbol{\sigma}_h(\boldsymbol{x}) := \sum_{\mathbf{a}\in\mathcal{V}_h} \boldsymbol{\sigma}_h^{\mathbf{a}}(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \Omega. \tag{2}$$

To solve the local minimization problem (1), the minimum is equivalently characterized as the solution of a mixed finite element problem of the form: find $(\boldsymbol{\sigma}_h^{\mathbf{a}}, r_h) \in \boldsymbol{V}_h^{\mathbf{a}} \times Q_h^{\mathbf{a}}$ satisfying

$$(\boldsymbol{\sigma}_h^{\mathbf{a}}, \boldsymbol{v}_h)_{\omega_{\mathbf{a}}} - (r_h, \nabla\cdot\boldsymbol{v}_h)_{\omega_{\mathbf{a}}} = -(\boldsymbol{\tau}_h^{\mathbf{a}}, \boldsymbol{v}_h)_{\omega_{\mathbf{a}}} \qquad \forall \boldsymbol{v}_h \in \boldsymbol{V}_h^{\mathbf{a}} \tag{3a}$$

$$(\nabla\cdot\boldsymbol{\sigma}_h^{\mathbf{a}}, q_h)_{\omega_{\mathbf{a}}} = (g^{\mathbf{a}}, r_h)_{\omega_{\mathbf{a}}} \qquad \forall q_h \in Q_h^{\mathbf{a}}, \tag{3b}$$

[*]Triscale Innov, 7 rue de la Croix Martre, 91120 Palaiseau, France.
[†]Inria, 2 rue Simone Iff, 75589 Paris, France.
[‡]Université Paris-Est, CERMICS (ENPC), Marne-la-Vallée, 77455, France.

where $(\cdot,\cdot)_{\omega_{\mathbf{a}}}$ denotes the L^2-inner product on $\omega_{\mathbf{a}}$ and $Q_h^{\mathbf{a}} \subset L^2(\omega_{\mathbf{a}})$ is a finite-dimensional broken space. We refer to the systems (3) for each $\mathbf{a} \in \mathcal{V}_h$ as *patch problems.*

An important practical feature of this construction is the *mutual independence* of the patch problems (3). This implies that the computation of the patch solutions $\boldsymbol{\sigma}_h^{\mathbf{a}}$ is embarrassingly parallel. However, to the best of our knowledge, an efficient parallel implementation has not been realized. In this work, we present a concrete algorithm with minimal memory allocations that uses the shared memory parallelism paradigm in the Julia programming language. Our algorithm presents three main novelties that are not obvious from the mathematical construction:

1. a setup phase consisting of a loop on cells to compute all necessary data for the patch problems;
2. reusing the standard local-to-global degree of freedom (DOF) mapping to construct a local-to-patch DOF mapping;
3. pre-computing the largest patch system size and preallocating all necessary matrices and vectors to be reused.

The resulting code is available under an open source MIT license in the `EquilibratedFlux.jl` package[1] and can be easily installed via the Julia package manager `Pkg`. The library naturally interfaces with the `Gridap.jl` library [1, 21] both in terms of usage and reuse of basic finite element library utilities.

The theory of the equilibrated flux is based on principles first established in Prager and Synge [17] and more recently in the works of Ladevèze and Leguillon [14], Destuynder and Métivet [6], Braess and Schöberl [2], and Ern and Vohralík [10]. The main application of the equilibrated flux is in a posteriori error estimation. The equilibrated flux gives rise to an error estimator that is reliable (constant free upper bound on the error) and efficient (lower bound on the error). As demonstrated in [2] (in two spatial dimensions) and [11] (in three spatial dimensions) the efficiency constant is independent of the polynomial approximation degree p. In particular, this means that the quality of the estimation does not degrade as the polynomial degree increases.

The equilibrated flux has been used to provide error estimates in many contexts, notably for diffusion problems [4], reaction-diffusion problems [19], the heat equation [9, 8], poro-elasticity [18], the Richards equation [15], eigenvalue problems [5], and nonlinear elliptic partial differential equations (PDEs) [12, 13].

The rest of this work is organized as follows. In §2 we give a precise description of the equilibrated flux in the context of the Poisson equation. In §3 we give a description of a naive algorithm for calculating the flux and discuss the associated trade-offs. In §4 we discuss our proposed efficient algorithm, with examples from the source code. In §5 we present two possible use cases for the flux: error estimation and driving adaptive mesh refinement. In §6 we demonstrate the performance of shared memory parallelization for computing the flux on a test case with 10 million DOFs. Finally, we conclude in §7 we acknowledge further optimizations to improve performance.

2 Problem description

2.1 Notation

For a subdomain $\omega \subseteq \Omega$, we denote the $L^2(\omega)$ inner product by $(\cdot,\cdot)_\omega$ and $\|\cdot\|_\omega$ the associated norm. We omit the subscript when $\omega = \Omega$. We consider a fixed triangular mesh $\mathcal{T}_h = \cup_K \{K\}$ of Ω such that $\cup_{K\in\mathcal{T}_h} K = \overline{\Omega}$. We enforce that the mesh is conforming in the sense that there are no hanging nodes: the intersection of (the closure) two arbitrary elements $K, K' \in \mathcal{T}_h$ is either empty or an l-dimensional simplex for $0 \le l \le d-1$. Next, let $\mathcal{V}_h$ denote the set of vertices in the mesh $\mathcal{T}_h$. We introduce the nodal patch $\mathcal{T}_{\mathbf{a}}$ of a node $\mathbf{a} \in \mathcal{V}_h$ by $\mathcal{T}_{\mathbf{a}} := \{K \in \mathcal{T}_h : \mathbf{a} \in K\}$. We denote the corresponding domain of $\mathcal{T}_{\mathbf{a}}$ by $\omega_{\mathbf{a}} \subset \Omega$. Finally, we define the nodes of a element $\mathcal{V}_K := \{\mathbf{a} \in \mathcal{V}_h : \mathbf{a} \in K\}$.

[1] `https://github.com/aerappa/EquilibratedFlux.jl`

We assume that we work with a fixed polynomial degree $p \geq 1$. We introduce the continuous Galerkin (cG) finite element space of of degree p by

$$V_h^p := H_0^1(\Omega) \cap \mathcal{P}_p(\mathcal{T}_h). \tag{4}$$

We denote the hat function basis by $\{\psi_{\mathbf{a}}\}_{\mathbf{a}\in\mathcal{V}_{\mathbf{a}}} \subset V_h^1$. A given hat function $\psi_{\mathbf{a}}$ is fully determined by its action on the vertices $\mathcal{V}_h$ of the mesh; indeed for $\mathbf{a}' \in \mathcal{V}_{\mathbf{a}}$ we have

$$\psi_{\mathbf{a}}(\mathbf{a}') = \begin{cases} 1 & \text{if } \mathbf{a}' = \mathbf{a}, \\ 0 & \text{otherwise.} \end{cases}$$

See Figure 1 for an illustration for a nodal patch domain $\omega_{\mathbf{a}}$ and the associated hat function $\psi_{\mathbf{a}}$.

We also introduce, for an element $K \in \mathcal{T}_h$, the Raviart–Thomas–Nédélec [3] mixed finite element space,

$$\boldsymbol{V}_h^p(K) := [\mathcal{P}_p(K)]^d + \boldsymbol{x}\mathcal{P}_p(K), \tag{5}$$

as well as the $\boldsymbol{H}(\text{div})$-conforming counterpart for a collection of elements $\mathcal{T} \subseteq \mathcal{T}_h$, and its corresponding subdomain $\omega \subseteq \Omega$

$$\boldsymbol{V}_h^p(\omega) := \{\boldsymbol{v}_h \in \boldsymbol{H}(\text{div}, \omega) : \boldsymbol{v}_h\big|_K \in \boldsymbol{V}_h^p(K) \quad \forall K \in \mathcal{T}\}, \tag{6}$$

as well as the broken space

$$Q_h^p(\omega) := \{v \in L^2(\omega) : v\big|_K \in \mathcal{P}_p(K) \quad \forall K \in \mathcal{T}\}. \tag{7}$$

We denote the dimensions of the element-wise spaces by

$$n_K := \dim(\boldsymbol{V}_h^p(K)), \quad m_K := \dim(\mathcal{P}_p(K)). \tag{8}$$

2.2 Poisson equation

To fix ideas, in this paper we consider the Poisson equation with homogeneous boundary conditions, i.e., for a polygonal domain $\Omega \subset \mathbb{R}^2$,

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{9}$$

where we make a simplifying assumption that $f \in \mathcal{P}_p(\mathcal{T}_h)$. The finite element approximation $u_h \in V_h^p$ of (9) satisfies

$$(\nabla u_h, \nabla v_h) = (f, v_h) \quad \forall v \in V_h^p, \tag{10}$$

This problem is well-posed thanks to the Lax–Milgram theorem. However, this method results in a “nonconforming flux”, i.e.,

$$-\nabla\cdot(\nabla u_h) \neq f. \tag{11}$$

2.3 Equilibrated flux for the Poisson equation

We now describe the post-processing procedure to produce the so-called equilibrated flux starting from the approximate cG solution u_h of (10).

Definition 1 (Equilibrated flux)**.** *Let $u_h \in V_h^p$ solve* (10)*. For each node $\mathbf{a} \in \mathcal{V}_h$, determine $\boldsymbol{\sigma}_h^{\mathbf{a}} \in \boldsymbol{V}_h^{\mathbf{a}}$ and $r_h \in Q_h^{\mathbf{a}}$ by solving*

$$(\boldsymbol{\sigma}_h^{\mathbf{a}}, \boldsymbol{v}_h)_{\omega_{\mathbf{a}}} - (r_h, \nabla\cdot\boldsymbol{v}_h)_{\omega_{\mathbf{a}}} = -(\psi_{\mathbf{a}}\nabla u_h, \boldsymbol{v}_h)_{\omega_{\mathbf{a}}}, \tag{12a}$$

$$(\nabla\cdot\boldsymbol{\sigma}_h^{\mathbf{a}}, q_h)_{\omega_{\mathbf{a}}} = (f\psi_{\mathbf{a}} - \nabla u_h \cdot \nabla\psi_{\mathbf{a}}, q_h)_{\omega_{\mathbf{a}}}, \tag{12b}$$

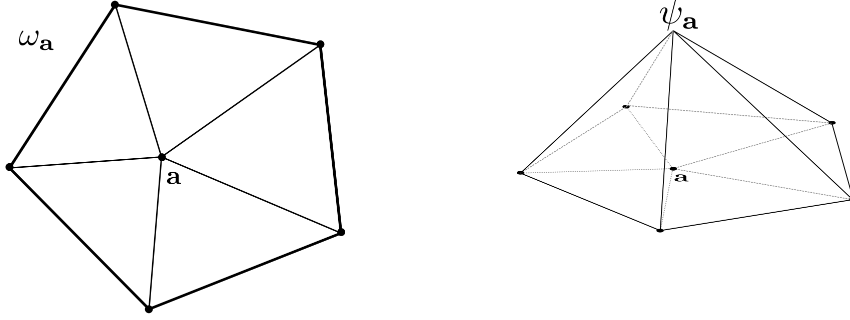

Figure 1: A nodal patch and its associated hat function.

for all $(\boldsymbol{v}_h, q_h) \in \boldsymbol{V}_h^{\mathbf{a}} \times Q_h^{\mathbf{a}}$. *The spaces are defined as*

$$\begin{aligned} \boldsymbol{V}_h^{\mathbf{a}} &:= \{\boldsymbol{v}_h \in \boldsymbol{V}_h^p(\omega_{\mathbf{a}}) : \boldsymbol{v}_h \cdot \boldsymbol{n}_{\omega_{\mathbf{a}}} = 0 \text{ on } \partial\omega_{\mathbf{a}}\}, \\ Q_h^{\mathbf{a}} &:= \{q_h \in Q_h^p(\omega_{\mathbf{a}}) : (q_h, 1)_{\omega_{\mathbf{a}}} = 0\}, \end{aligned} \qquad \mathbf{a} \in \mathcal{V}_h^{\text{int}}, \tag{13a}$$

$$\begin{aligned} \boldsymbol{V}_h^{\mathbf{a}} &:= \{\boldsymbol{v}_h \in \boldsymbol{V}_h^p(\omega_{\mathbf{a}}) : \boldsymbol{v}_h \cdot \boldsymbol{n}_{\omega_{\mathbf{a}}} = 0 \text{ on } \partial\omega_{\mathbf{a}} \setminus \partial\Omega\}, \\ Q_h^{\mathbf{a}} &:= Q_h^p(\omega_{\mathbf{a}}), \end{aligned} \qquad \mathbf{a} \in \mathcal{V}_h^{\text{ext}}. \tag{13b}$$

The global flux is then defined by

$$\boldsymbol{\sigma}_h := \sum_{\mathbf{a} \in \mathcal{V}_h} \boldsymbol{\sigma}_h^{\mathbf{a}}. \tag{14}$$

Remark 1 (Link with the general problem)**.** *The local flux given in* (12) *can be seen as a special case of the general flux given in* (3) *with* $\boldsymbol{\tau}_h^{\mathbf{a}} = \psi_{\mathbf{a}} \nabla u_h$ *and* $g^{\mathbf{a}} = f\psi_{\mathbf{a}} - \nabla u_h \cdot \nabla \psi_{\mathbf{a}}$. *Furthermore, the global flux* (14) *satisfies*

$$\boldsymbol{\sigma}_h \in \boldsymbol{H}(\text{div}, \Omega), \quad \nabla \cdot \boldsymbol{\sigma}_h = f. \tag{15}$$

Remark 2 (Estimator based on the flux)**.** *We can define an error estimator* η *using the equilibrated flux via*

$$\eta_K := \|\nabla u_h + \boldsymbol{\sigma}_h\|_K, \quad \eta := \left(\sum_{K \in \mathcal{T}_h} \eta_K^2 \right)^{1/2}. \tag{16}$$

We then have the following bound on the error in the H^1*-seminorm (reliability):*

$$\|\nabla(u - u_h)\| \leq \eta. \tag{17}$$

The estimator also provides a local lower bound (efficiency):

$$\eta_K \lesssim \sum_{\mathbf{a} \in \mathcal{V}_K} \|\nabla(u - u_h)\|_{\omega_{\mathbf{a}}}. \tag{18}$$

and the hidden constant is independent of (in particular) the polynomial degree p *for spatial dimension* $d \leq 3$*, see [2, 11].*

2.4 Linear algebra representation

Once a basis $\{\boldsymbol{v}_i^{\mathbf{a}}\}_{i=1}^{n_{\mathbf{a}}}$ of $\boldsymbol{V}_h^{\mathbf{a}}$ and $\{q_i^{\mathbf{a}}\}_{i=1}^{m_{\mathbf{a}}}$ of $Q_h^{\mathbf{a}}$ are chosen, we can also write the patch system (12) in matrix form,

$$\mathcal{A}_{\mathbf{a}} \mathcal{X}_{\mathbf{a}} = \mathcal{F}_{\mathbf{a}} \tag{19}$$

with the block decomposition

$$\mathcal{A}_{\mathbf{a}} = \begin{pmatrix} M_{\mathbf{a}} & B_{\mathbf{a}}^T \\ B_{\mathbf{a}} & 0 \end{pmatrix}, \mathcal{X}_{\mathbf{a}} = \begin{pmatrix} \boldsymbol{x}_{\mathbf{a}} \\ x_{\mathbf{a}} \end{pmatrix}, \mathcal{F}_{\mathbf{a}} = \begin{pmatrix} \boldsymbol{F}_{\mathbf{a}} \\ F_{\mathbf{a}} \end{pmatrix}. \tag{20}$$

The block matrices $M_{\mathbf{a}} \in \mathbb{R}^{n_{\mathbf{a}} \times n_{\mathbf{a}}}$ and $B_{\mathbf{a}} \in \mathbb{R}^{m_{\mathbf{a}} \times n_{\mathbf{a}}}$ are given by

$$[M_{\mathbf{a}}]_{ij} := (\boldsymbol{v}_j^{\mathbf{a}}, \boldsymbol{v}_i^{\mathbf{a}})_{\omega_{\mathbf{a}}} \tag{21a}$$

$$[B_{\mathbf{a}}]_{ij} := (\nabla \cdot \boldsymbol{v}_j^{\mathbf{a}}, q_i^{\mathbf{a}})_{\omega_{\mathbf{a}}}, \tag{21b}$$

while the right hand side vectors $\boldsymbol{F}_{\mathbf{a}} \in \mathbb{R}^{n_{\mathbf{a}}}$ and $F_{\mathbf{a}} \in \mathbb{R}^{m_{\mathbf{a}}}$ are given by

$$[\boldsymbol{F}_{\mathbf{a}}]_j := -(\psi_{\mathbf{a}} \nabla u_h, \boldsymbol{v}_j)_{\omega_{\mathbf{a}}}, \tag{22a}$$

$$[F_{\mathbf{a}}]_j := (f\psi_{\mathbf{a}} - \nabla u_h \cdot \nabla \psi_{\mathbf{a}}, q_j)_{\omega_{\mathbf{a}}}. \tag{22b}$$

The degrees of freedom for the local equilibrated flux $\boldsymbol{\sigma}_h^{\mathbf{a}}$ are contained in the vector $\boldsymbol{x}_{\mathbf{a}}$.

3 Naive implementation of the equilibrated flux

We first consider in Algorithm 1 a possible implementation of the equilibrated flux assembly Definition 1 at a high level of abstraction. The algorithm follows closely to the mathematical formalism: we are provided data $-\nabla u_h$ and f coming from the Poisson problem. We then initialize the final flux object $\boldsymbol{\sigma}_h$ and enter a loop on nodes/patches. Inside the loop, we rely on a finite element library that can create the required spaces $\boldsymbol{V}_h^{\mathbf{a}}$ and $Q_h^{\mathbf{a}}$ as well as evaluate the bilinear forms in (21) and (22). We then solve the linear problem (19) and scatter the local DOFs to the global DOFs. We also ignore details like the fact that the geometrical and topological patch information needs to be extracted from the mesh. Many libraries do not support this, as it is not part of the standard assembly procedure.

3.1 Disadvantage of the naive implementation

While the Algorithm 1 is very simple to write down and has a nice correspondence with the mathematical description, it is unfortunately inefficient. In particular, this algorithm inherently requires many dynamic memory allocations inside the loop on patches. Firstly, at each iteration of the loop on line 3, the finite element spaces are rebuilt completely from scratch, including, in particular, all the information pertaining to the reference element (which is often the most heavyweight in terms of allocation). In addition, on line 4 the matrix and vectors associated to the patch system are not reused between iterations, invoking more dynamic allocations.

Dynamic memory allocation can be a significant performance bottleneck due to overhead and data locality. Firstly, the process of allocating and deallocating memory dynamically may involve system calls, which are substantially slower compared to accessing stack or pre-allocated heap memory. This added computational overhead is particularly problematic in scenarios involving frequent allocations/deallocations, such as in tight loops or high-frequency function calls. Secondly, modern processors use a hierarchy of caches to speed up access to frequently used data. Good data locality means that the data a program needs is either already in the cache or close together in memory, allowing for efficient cache usage. Dynamic memory allocation can lead to poor data locality because it often allocates memory in a non-contiguous manner. Objects that are logically related in a program might end up being physically scattered in memory. This scattering can result in more cache misses (where the required data is not in the cache), leading to slower performance as the processor has to fetch data from the slower main memory.

These considerations are independent of the programming language, and apply just as much to a statically compiled language with manual memory management like C as to Julia which uses just-in-time (JIT) compilation and is garbage collected[2]. Thus, our approach to reduce dynamic memory allocations is applicable and relevant to any choice of language for someone interested in serial performance and even more so for parallel performance in a shared memory parallelism paradigm. This is because the memory buses can quickly become congested with requests from the various threads to main memory.

[2] This effect is more pronounced in the case of a garbage collected language, as frequent (de)allocations create pressure on the garbage collector.

3.2 Proposed solutions

As described in the introduction, we propose three main ways to reduce dynamic memory allocations:

1. Assemble cellwise versions of (21) and (22)
2. Introduce lightweight mappings from the cell DOF index space to the patch DOF index space
3. Reuse patch-local vectors and matrices

In the following section we show how these three steps lead to a much more efficient algorithm, with very few, or even zero memory allocations inside the main loop on patches.

Algorithm 1: Naive equilibrated flux assembly

Input: Mesh $\mathcal{T}_h$, approximate flux $-\nabla u_h$, source term f
Output: Global equilibrated flux $\boldsymbol{\sigma}_h$

1. Initialize $\boldsymbol{\sigma}_h \in \boldsymbol{V}_h^p(\Omega)$
2. **foreach** *node* $\mathbf{a} \in \mathcal{V}_h$ **do**
3. Instantiate the finite element spaces $\boldsymbol{V}_h^{\mathbf{a}}$ and $Q_h^{\mathbf{a}}$ of (13)
4. Allocate patch matrix and vector $\mathcal{A}_{\mathbf{a}}$ and $\mathcal{F}_{\mathbf{a}}$
5. Populate $\mathcal{A}_{\mathbf{a}}$ and $\mathcal{F}_{\mathbf{a}}$ using $\boldsymbol{V}_h^{\mathbf{a}}, Q_h^{\mathbf{a}}$
6. Solve $\mathcal{X}_{\mathbf{a}} = \mathcal{A}_{\mathbf{a}}^{-1}\mathcal{F}_{\mathbf{a}}$
7. Scatter $\boldsymbol{x}_{\mathbf{a}} = \texttt{DOFs}(\boldsymbol{\sigma}_h^{\mathbf{a}})$ to $\texttt{DOFs}(\boldsymbol{\sigma}_h)$
8. **return** $\boldsymbol{\sigma}_h$

4 Efficient algorithm

In this section we consider a more involved algorithm that relies less on the builtin functionality provided by a standard finite element library. However, the result is a more efficient algorithm that is also more amenable to shared memory parallelization.

4.1 High level view

As explained in the previous section, one key to the efficient implementation will be to perform an initial cellwise assembly. For an element K and bases $\{\boldsymbol{v}_i^K\}_{i=1}^{n_K}$ of $\boldsymbol{V}_h^p(K)$ as well as $\{q_i^K\}_{i=1}^{m_K}$ of $Q_h^p(K)$, we define the cell block matrices $M_K \in \mathbb{R}^{n_K \times n_K}$ and $B_K \in \mathbb{R}^{m_K \times n_K}$ by

$$[M_K]_{ij} := (\boldsymbol{v}_j^K, \boldsymbol{v}_i^K)_K \tag{23a}$$

$$[B_K]_{ij} := (\nabla\cdot\boldsymbol{v}_j^K, q_i^K)_K. \tag{23b}$$

The right-hand sides of (12) are slightly more involved because they depend on the hat functions $\psi_{\mathbf{a}}$, of which there are $d+1$ for a d-dimensional simplex K. Thus, the right-hand sides are also indexed by the node $\mathbf{a}$,

$$[\boldsymbol{F}_K^{\mathbf{a}}]_j := -(\psi_{\mathbf{a}}\big|_K \nabla u_h, \boldsymbol{v}_j^K)_K, \tag{24a}$$

$$[F_K^{\mathbf{a}}]_j := (f\psi_{\mathbf{a}}\big|_K - \nabla u_h \cdot \nabla\left(\psi_{\mathbf{a}}\big|_K\right), q_j^K)_K. \tag{24b}$$

We now present Algorithm 2, which starts with a loop on cells creating the objects in (23) and (24). This loop is a common paradigm in finite element codes, and in certain cases this functionality is exposed. This is indeed the case in `Gridap.jl`, and in our implementation this part of the algorithm relies mostly on existing technology in the library. We discuss this in more detail in §4.3.

The next step of the algorithm is the pre-allocation of $\mathcal{A}_{\mathbf{a}}$ and $\mathcal{F}_{\mathbf{a}}$, on line 5. This is nontrivial since the dimensions of the patch problems are variable, and we will address this in §4.4. Next, we

see that we only perform read operations on the matrices M_K, B_K inside the loop on patches. This effectively reduces the number of allocations inside the loop on patches to zero. We also remark that many details are missing, and in particular we have not discussed the scatters on lines 10, 11, and 14. This requires careful handling of indices. We will give more details about managing indices in §4.5. We also need to remove allocations due to the resolution of the linear system on line 13, which we discuss in §4.4. A prerequisite for all the aforementioned steps is some topological information about the patches, which we discuss in §4.2.

Algorithm 2: Efficient equilibrated flux assembly

Input: Mesh $\mathcal{T}_h$, approximate flux $-\nabla u_h$, source term f
Output: Global equilibrated flux $\boldsymbol{\sigma}_h$

```
// Loop on cells
```
1 **foreach** *cell* $K \in \mathcal{T}_h$ **do**
2   Build the cellwise matrices M_K, B_K of (23)
3   **for** *each node* $\mathbf{a}$ *in the cell* K **do**
4   Build the cellwise vectors $\boldsymbol{F}_K^{\mathbf{a}}$, $F_K^{\mathbf{a}}$ of (24)
5 Allocate $\mathcal{A}_{\mathbf{a}}, \mathcal{F}_{\mathbf{a}}$
6 Initialize $\boldsymbol{\sigma}_h \in \boldsymbol{V}_h^p(\Omega)$
```
// Loop on patches
```
7 **foreach** *node* $\mathbf{a} \in \mathcal{V}_h$ **do**
8   Zero out $\mathcal{A}_{\mathbf{a}}$ and $\mathcal{F}_{\mathbf{a}}$
9   **foreach** *cell* $K \in \mathcal{T}_{\mathbf{a}}$ **do**
10   Scatter M_K, B_K to $\mathcal{A}_{\mathbf{a}}$
11   Scatter $F_K^{\mathbf{a}}, \boldsymbol{F}_K^{\mathbf{a}}$ to $\mathcal{F}_{\mathbf{a}}$
12 Impose boundary conditions on $\mathcal{A}_{\mathbf{a}}, \mathcal{F}_{\mathbf{a}}$
13 Solve $\mathcal{X}_{\mathbf{a}} = \mathcal{A}_{\mathbf{a}}^{-1}\mathcal{F}_{\mathbf{a}}$
14 Scatter $\boldsymbol{x}_{\mathbf{a}} = \texttt{DOFs}(\boldsymbol{\sigma}_h^{\mathbf{a}})$ to $\texttt{DOFs}(\boldsymbol{\sigma}_h)$
15 **return** $\boldsymbol{\sigma}_h$

4.2 Topological patch information

We construct a small type hierarchy, shown in Listing 1, that contains the necessary topological information pertaining to patches.

The two types of patches `DirichletPatch` and `InteriorPatch` correspond to the differences in the definitions of the spaces (13). The `PatchData struct` contains topological information related to the patch which is extracted through the `Gridap.Geometry.get_faces` function. For example, `node_to_cell = Geometry.get_faces(topo, 0, 2)` gives a vector of vectors for each node in the mesh to the ids of the cells it belongs to.

4.3 Cellwise assembly

As established in §4.1, we want to compute cellwise matrices and vectors of (23) and (24). To achieve this, we rely on existing machinery available in many finite element libraries, and in particular in the `Gridap.jl` library [3]. Therefore, the code in Listing 2 is meant only to illustrate the main steps of setting up the loop on cells in the setting of the `Gridap.jl` library in particular, but the main ideas can be carried over to other libraries. Furthermore, including this construction allows the final loop on patches to be self contained in the sense that all the data structures are defined. Finally, we note that here the loop on cells is not explicit, but comes implcitly through the definitions of objects that are defined cellwise on the entire mesh.

We first build the spaces $\boldsymbol{V}_h^p(K)$ and $Q_h^p(K)$ on the reference element and then extend it to the whole mesh $\mathcal{T}_h$ via the code in Listing 2, lines 1–8.

[3]see, e.g., the low-level developer tutorial `https://gridap.github.io/Tutorials/stable/pages/t013_poisson_dev_fe/`.

```
1   # T is meant to be an integer type large enough to store all cell/node
2   # indices in the mesh
3   abstract type Patch{T} where {T <: Integer} end
4
5   struct PatchData{T <: Integer}
6     node_to_offsets::Vector{T}
7     patch_cell_ids::Vector{T}
8     bdry_edge_ids::Vector{T}
9     all_edge_ids::Vector{T}
10  end
11
12  struct DirichletPatch{T <: Integer} <: Patch{T}
13    data::PatchData{T}
14  end
15
16  struct InteriorPatch{T <: Integer} <: Patch{T}
17    data::PatchData{T}
18  end
```

Listing 1: Topological patch information

With these spaces in hand, we can evaluate bilinear forms cell-wise on the mesh. The ingredients for this will be the basis of these respective spaces, as well as a `CellQuadrature` to perform cellwise integration at quadrature points. We create these objects at lines 10–13.

The interface for evaluating the bilinear forms cellwise is then very straightforward, and is used at lines 15–16 to construct the mass matrices $\{M_K\}_{K\in\mathcal{T}_h}$, and the mixed form matrices $\{B_K\}_{K\in\mathcal{T}_h}$ as in (23).

The right-hand sides are more involved due to the fact that in two dimensions each triangle has three vertices and therefore three hat functions are supported on it and must be accounted for. First, we consider two helper functions to get the hat function basis (lines 18–31). Once these two functions are in place to extract the hat functions, we can assemble the right-hand sides: one for each hat function. This is done in lines 33–42.

The arrays `cell_RHS` then correspond to $\{\boldsymbol{F}_K^{\mathbf{a}}\}_{K\in\mathcal{T}_h}$ and $\{F_K^{\mathbf{a}}\}_{K\in\mathcal{T}_h}$ of (24). We add a Lagrange multiplier row to handle the mean free condition in the case of interior patches (see (13a)) and then collect all the cellwise objects in a `NamedTuple` called `co` (lines 44-45).

4.4 Patch-level linear algebra

In this section, we detail our strategy for reusing the matrices and vectors of the patch system between iterations. The dimensions of the spaces on the reference element are known and calculated via the function `get_dofs_per_cell` in Listing 3, lines 1–5.

Next, we need to know the maximum number of cells in a patch. We obtain this information during the initial creation of the `Patch` objects of §4.2, which we call `max_patch_cells` (lines 7–17). We now collect all the patchwise matrices and vectors into a `NamedTuple` as we did for the cellwise data (line 19).

Finally, we note that we also instantiate a pivot vector `ws`. This is required for the library `FastLapackInterface.jl`[4] which allows efficient (in terms of allocation) calls to the underlying `LAPACK` solver. Listing 4 shows how these preallocated objects are ultimately re-used during the loop on patches, where `n_free_dofs` depends on the patch currently being considered.

4.5 The DOF manager

We now discuss the required bookkeeping for the patch indexed DOFs. In particular, the cell objects of §4.3 are all cell-locally indexed. The `Gridap.jl` library exposes the so-called element-local to mesh-global DOF map (accessed via `Gridap.FESpaces.get_cell_dof_ids`) that is standard in

[4] `https://github.com/DynareJulia/FastLapackInterface.jl`

```
# Assume we already have a desired polynomial degree p
# and Triangulation 𝒯ₕ
reffeRT = ReferenceFE(raviart_thomas, Float64, p)
reffeP = ReferenceFE(lagrangian, Float64, p)
# Raviart-Thomas space for the flux
RT_space = FESpace(𝒯ₕ, reffeRT, conformity = :HDiv)
# Broken L^2 space for the Lagrange multiplier
L2_space = FESpace(𝒯ₕ, reffeP, conformity = :L2)

Qₕ = CellQuadrature(𝒯ₕ, quad_order)
dvp = get_trial_fe_basis(L2_space)
duRT = get_fe_basis(RT_space)
dvRT = get_trial_fe_basis(RT_space)

cell_mass_mats = ∫(duRT ⋅ dvRT) * Qₕ
cell_mixed_mats = ∫((∇ ⋅ duRT) * dvp) * Qₕ

function _get_hat_function_cellfield(i, basis_data, 𝒯ₕ)
  cell_to_ith_node(c) = c[i]
  A = lazy_map(cell_to_ith_node, basis_data)
  return GenericCellField(A, 𝒯ₕ, ReferenceDomain())
end

function _get_hat_functions_on_cells(𝒯ₕ)
  # Always degree 1
  reffe = ReferenceFE(lagrangian, Float64, 1)
  V0 = TestFESpace(𝒯ₕ, reffe; conformity = :H1, dirichlet_tags = "boundary")
  fe_basis = get_fe_basis(V0)
  bd = Gridap.CellData.get_data(fe_basis)
  return bd
end

hat_fns_on_cells = _get_hat_functions_on_cells(𝒯ₕ)
RHS_RT_form(ψ) = ∫(-ψ * (∇uₕ ⋅ dvRT))*Qₕ
RHS_L²_form(ψ) = ∫((f * ψ - ∇uₕ ⋅ ∇(ψ))*dvp)*Qₕ
cur_num_cells = num_cells(𝒯ₕ)
nodes_per_cell = 3
for i = 1:nodes_per_cell
  ψᵢ = _get_hat_function_cellfield(i, hat_fns_on_cells, 𝒯ₕ)
  cell_RHS_RTs[i] = RHS_RT_form(ψᵢ)
  cell_RHS_L2s[i] = RHS_L²_form(ψᵢ)
end

cell_Λ_vecs = ∫(1 * dvp) * Qₕ
co = (; cell_mass_mats, cell_Λ_vecs, cell_mixed_mats, cell_RHS_RTs, cell_RHS_L²s)
```

Listing 2: Cellwise assembly

```julia
function get_dofs_per_cell(p, d)
  RT_dofs_per_cell = (p + d + 1) * binomial(p + d - 1, p)
  L²_dofs_per_cell = binomial(p + d, p)
  (RT_dofs_per_cell, L²_dofs_per_cell)
end

(RT_dofs_per_cell, L²_dofs_per_cell) = get_dofs_per_cell(p, d)
M = zeros(RT_dofs_per_cell * max_patch_cells, RT_dofs_per_cell * max_patch_cells)
B = zeros(RT_dofs_per_cell * max_patch_cells, L²_dofs_per_cell * max_patch_cells)
A = [M B; transpose(B) zeros(size(B)[2], size(B)[2])]
# Pre-allocate the pivot vector for the matrix A
ws = LUWs(A)
RHS_RT = zeros(max_patch_cells * RT_dofs_per_cell)
RHS_L² = zeros(max_patch_cells * L²_dofs_per_cell)
RHS = [RHS_RT; RHS_L²]
Λ = similar(RHS_L²)
σ_loc = similar(RHS)

linalg = (; M, B, A, ws, Λ, RHS_RT, RHS_L², RHS, σ_loc)
```

Listing 3: Patch-level linear algebra: pre-allocation of re-usable objects

```julia
A_free_dofs = @view linalg.A[1:n_free_dofs, 1:n_free_dofs]
RHS_free_dofs = @view linalg.RHS[1:n_free_dofs]
σ_free_dofs = @view linalg.σ_loc[1:n_free_dofs]
ldiv!(σ_free_dofs, LU(LAPACK.getrf!(linalg.ws, A_free_dofs)...), RHS_free_dofs)
```

Listing 4: Patch-level linear algebra: re-use of pre-allocated objects for each patch

```julia
struct DOFManager{T, M <: Matrix{T}, V <: Vector{T}}
  # All the cell dofs stored as a matrix
  all_cell_dofs_gl::M
  # The current patch dofs in the global ennumeration
  patch_dofs_gl::V
  # The current free dofs in patch local ennumeration for slicing
  # into the patch local objects
  free_patch_dofs_loc::V
  # The current cell's dofs in the patch local ennumeration
  cell_dofs_loc::V
end

function update_cell_local_dofs!(dm::DOFManager, cellid)
  empty!(dm.cell_dofs_loc)
  for id in cur_cell_dofs_gl
    new_id = findfirst(n -> n == id, dm.patch_dofs_gl)
    new_id isa Nothing && error("Cannot update cell local dofs!")
    push!(dm.cell_dofs_loc, new_id)
  end
end
```

Listing 5: DOF manager

finite element codes. This map allows the scatter procedure from the cell-locally indexed objects to objects with mesh global indexing. We will be able to reuse this information to create a local-to-global map where the global index space is a patch $\omega_{\mathbf{a}}$ and not the full mesh $\mathcal{T}_h$. To this end, we introduce the `struct` defined in Listing 5, lines 1–11.

The `DOFManager` is built on the cell-local to mesh-global map to construct cell-local to patch-global maps for each patch. The field `all_cell_dofs_gl` stores the cell-to-global DOF mapping which is a `ncells` × `ndofs` vector of vectors. Here, `ndofs` represents the number of DOFs on a cell. The other fields of the `DOFManager` are updated dynamically using geometric information, but with no memory allocations using the `empty!` and `push!` methods (the convention in Julia is to append non-allocating function names with an exclamation point and modify the first argument). For example, the function defined at lines 13–20 updates `cell_dofs_loc` in place.

4.6 The loop on patches

```
1  for patch in patches
2    # Change the local numbering for the current patch
3    update_patch_dofs!(dm_RT, patch.data)
4    update_patch_dofs!(dm_L², patch.data)
5    # Scatter the cell based matrices in cell_objects to linalg
6    matrix_scatter!(linalg.M, co.cell_mass_mats, dm_RT, dm_RT, patch.data)
7    matrix_scatter!(linalg.B, co.cell_mixed_mats, dm_L², dm_RT, patch.data)
8    # Idem for vectors
9    vector_scatter!(linalg.RHS_RT, co.cell_RHS_RTs, dm_RT, patch.data)
10   vector_scatter!(linalg.RHS_L², co.cell_RHS_L²s, dm_L², patch.data)
11   single_vector_scatter!(linalg.Λ, co.cell_Λ_vecs, dm_L², patch.data)
12   # Now that scatter to local system is complete, remove fixed dofs
13   remove_homogeneous_neumann_dofs!(dm_RT, patch.data, degree)
14   # Count the free dofs for this patch once the BCs are imposed
15   n_free_dofs_RT = count_free_dofs(dm_RT)
16   n_free_dofs_L² = count_free_dofs(dm_L²)
17   n_free_dofs = n_free_dofs_RT + n_free_dofs_L²
18   # Use the sub-matrices and vectors generated from the scatters to build
19   # the monolithic objects
20   setup_patch_system!(linalg.A, linalg.RHS, linalg, dm_RT, dm_L²)
21   # Handle the pure Neumann case
22   if patch isa InteriorPatch
23     add_lagrange!(linalg.A, dm_RT, dm_L², linalg.Λ)
24     n_free_dofs += 1
25   end
26   solve_patch!(linalg, n_free_dofs)
27   # Scatter to the global FE object's free_values
28   scatter_to_global_σ!(σ_gl, dm_RT, linalg.σ_loc, n_free_dofs_RT)
29 end
```

Listing 6: Loop on patches

We now consider the heart of the code, i.e., the loop on patches. The efforts up to this point have been so that this part of the algorithm can be completely free of memory allocations so that the shared memory parallelism is efficient. We only show in Listing 6 a serial version of this loop for simplicity.

In this loop, it is assumed that `DOFManagers` `dm_RT` and `dm_L2` have been created out of their respective spaces. It is also important to note that the various scatter functions are non-allocating. This is illustrated in Listing 7, which details the implementation of `vector_scatter`.

```julia
function vector_scatter!(patch_vec, cell_vecs, dm, patch_data)
  fill!(patch_vec, 0)
  node_to_offsets = patch_data.node_to_offsets
  for (i, cellid) in enumerate(patch_data.patch_cell_ids)
    cell_vec_all_nodes = @view cell_vecs[cellid, :]
    offset = node_to_offsets[i]
    update_cell_local_dofs!(dm, cellid)
    cell_vec = cell_vec_all_nodes[offset]
    for i in axes(cell_vec, 1)
      patch_vec[dm.cell_dofs_loc[i]] += cell_vec[i]
    end
  end
end
```

Listing 7: Example of a non-allocating scatter function

5 Example use cases of the equilibrated flux

The following two sections closely follow the tutorials which can be found online in the documentation for the `EquilibratedFlux.jl` package[5].

5.1 Error estimation

In this tutorial we show how the equilibrated flux can be used to create an error estimator (16). We consider the following example data for the discrete Poisson problem (10):

- $\Omega = (0, 1)$
- $\mathcal{T}_h$ uniform $10 \times 10 \times 2$ triangular mesh
- $u(x, y) = \sin(2\pi x)\sin(\pi y)$
- $f(x, y) = 5\pi^2 \sin(2\pi x)\sin(\pi y)$
- Polynomial degree $p = 1$

We assume that the discrete solution `uh` has been computed via `Gridap.jl`. Then assembling an cellwise version of the estimator (16) can be achieved very easily using `EquilbratedFlux.jl` as demonstrated in the code snippet in Listing 8.

We then plot the square root of each of these quantities cellwise in Figure 2. We see that indeed, they are in very good agreement, even locally on each element. This property will be crucial for informing an adaptive mesh refinement procedure (as we will see in §5.2) where the local error distribution is used to bisect and generate new elements.

Next, we consider how well the flux we compute approximates the divergence, which we measure by $\|\nabla\cdot\boldsymbol{\sigma}_h - \Pi_h f\|$, where Π_h denotes the orthogonal projection onto the space $\mathcal{P}_1(\mathcal{T}_h)$. In fact, mathematically this quantity should be identically zero, and we show in Figure 3 that it is close to machine precision. This acts as a validation that the algorithm is working correctly.

5.2 Mesh adaptivity

In this section we show how the error estimator defined in the previous section can be used inside the standard adaptive finite element (AFEM) loop, i.e.,

$$\text{SOLVE} \to \text{ESTIMATE} \to \text{MARK} \to \text{REFINE}. \tag{25}$$

We demonstrate this for the classical corner singularity using the following data:

- $\Omega = (-1, 1) \setminus ([0, 1) \times (-1, 0])$

[5] `https://aerappa.github.io/EquilibratedFlux.jl/dev/`

```julia
# uh is the finite element solution on the mesh 𝒯ₕ
using EquilibratedFlux
σ = build_equilibrated_flux(-∇(uh), f, model, degree)
# Define some helper functions
L2_inner_product(f, g, dx) = ∫(f ⋅ g) * dx
L2_norm_squared(f, dx) = L2_inner_product(f, f, dx)
η²     = L2_norm_squared(σ_eq + ∇(uh), dx)
H1err² = L2_norm_squared(∇(u - uh), dx)
η_cellwise     = sqrt.(getindex(η_eq², 𝒯ₕ))  # Vector{Float64}
H1err_cellwise = sqrt.(getindex(H1err², 𝒯ₕ)) # Vector{Float64}
```

Listing 8: Example use case: error estimation

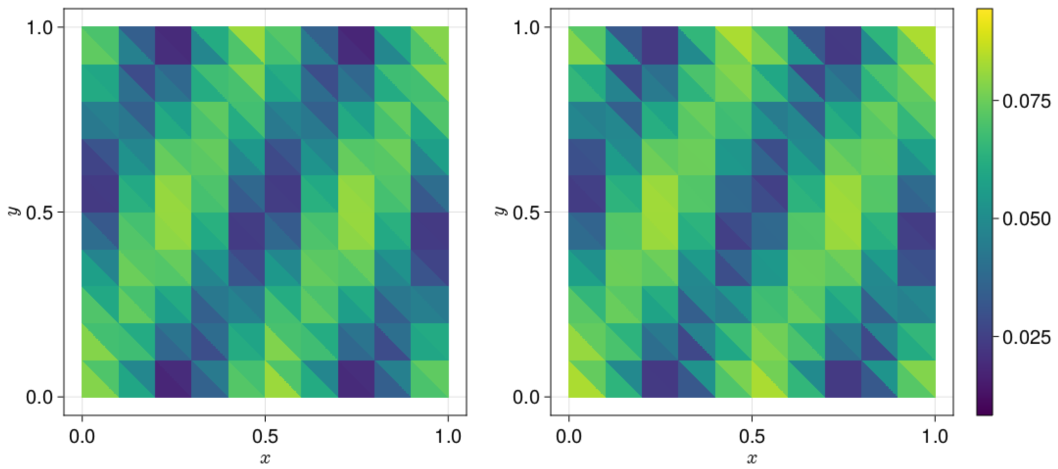

Figure 2: Element-wise error $\|\nabla(u_h - u)\|_K$ (left) and estimator $\|\nabla u_h + \boldsymbol{\sigma}_h\|_K$ (right).

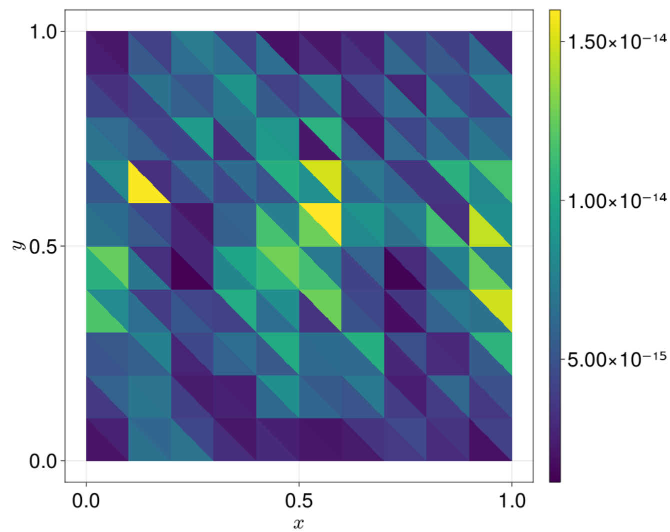

Figure 3: The element-wise divergence misfit $\|\nabla\cdot\boldsymbol{\sigma}_h - \Pi_h f\|_K$ for the equilibrated flux.

```julia
function estimate_laplace(uh, dx, model, degree)
    σ = build_equilibrated_flux(-∇(uh), x -> 0.0, model, degree)
    η² = L2_norm_squared(σ + ∇(uh), dx)
    Ω = Triangulation(model)
    getindex(η², Ω)
end

while η > tol
  # SOLVE
  uh, dx, dofs = solve_laplace(model, degree, g)
  # ESTIMATE
  η_arr = estimate_laplace(uh, dx, model, degree)
  # MARK
  cells_to_refine = dorfler_marking(η_arr)
  # REFINE
  model = refine(model, refinement_method = "nvb",
                 cells_to_refine = cells_to_refine)
end
```

Listing 9: Example use case: mesh adaptivity

- $u(r, \theta) = r^\alpha \sin(\alpha\theta)$, $\alpha = 2/3$
- $f = 0$

The AFEM loop takes the form presented in Listing 9 (removing some details for simplicity) where the `estimate_laplace` function uses the `build_equilibrated_flux` function to compute an error estimator. The other modules, e.g., `dorfler_marking` for a Dörfler marking strategy [7], are standard and are fully detailed in the online documentation[6].

With this AFEM procedure, we can explore solving the problem for increasing polynomial degree p. The result of running this code for $p \in \{1, 2, 3, 4, 5\}$ is given in Figure 5 where we plot the estimator and error as a function of total degrees of freedom (DOFs) of the linear system. Instead of using this estimator as a stopping criterion as indicated in the AFEM loop, we simply stop the iteration when the mesh contains 3000 triangles. We observe the theoretically optimal rate of convergence with respect to DOFs, see e.g., [16]. Finally, we remark that the refinement strategy is markedly different with respect to the polynomial degree, as seen in Figure 4.

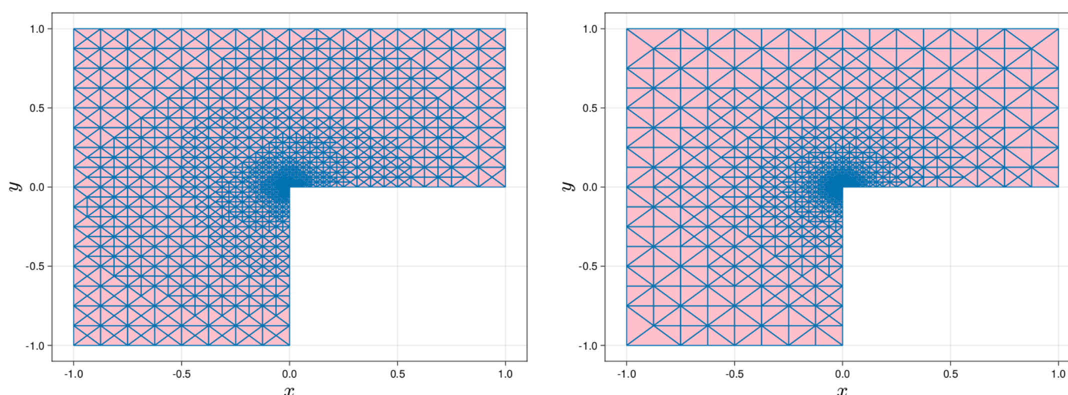

Figure 4: The final refined mesh with 3000 triangles using adaptive refinement for polynomial degrees $p = 1$ (left) and $p = 5$ (right).

[6] https://aerappa.github.io/EquilibratedFlux.jl/dev/examples/Lshaped/Lshaped/

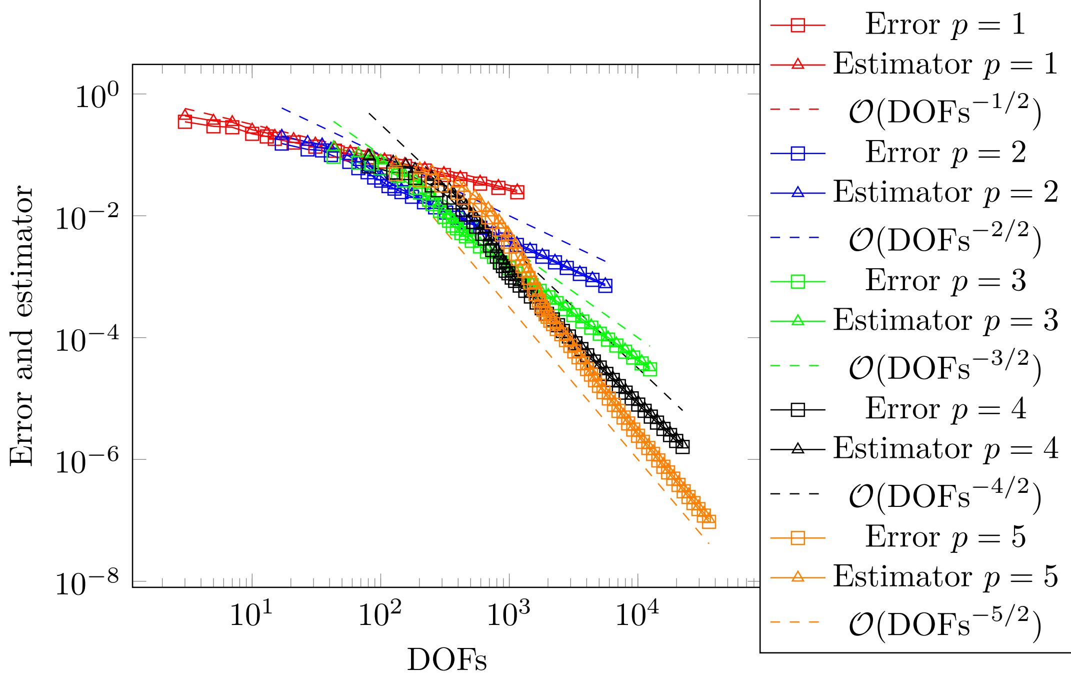

Figure 5: Optimal rate of convergence for both the estimator and the error for varying polynomial degree p on the L-shaped domain example using adaptive mesh refinement of §5.2.

6 Performance test

We now consider an experiment similar to the one conducted in [12, Appendix B] but with more details. In particular, we reconsider the data of §5.1 but with a variable mesh size so as to have approximately 10 millions DOFs for the Poisson problem (10) with polynomial degrees $p \in \{1, 2, 3, 4, 5\}$. We conduct our experiments on dual socket cluster node equipped with two Cacade Lake Intel Xeon 5218, each with 16 2.4GHz cores, and 192 GB of 2667 MHz RAM. We consider the following metric to evaluate the parallel scalability of Algorithm 2:

$$\text{Speedup} := \frac{\mathtt{T_1}}{\mathtt{T_P}} \leq \mathtt{P}, \tag{26}$$

where $\mathtt{T_P}$ is the time required for $\mathtt{P}$ processors and perfect scaling would result in Speedup $= \mathtt{P}$. In Figure 6 we plot the speedup for the two main loops described in Algorithm 2: the loop on cells starting on Line 1 and the loop on patches starting on Line 7. We first note that in both cases, the parallelization does indeed provide a speedup. The speedup is however much larger for the loop on patches which we have optimized to avoid allocations. At this stage, we have not fully optimized the setup phase involving the loop on cells. Nevertheless, a speedup is observed for the loop on cells, with a maximum speedup of 10x using 16 cores for polynomial degree $p = 5$. In contrast, the maximum speedup for the loop on patches is 13x using 16 cores for polynomial degree $p = 4$.

We also plot the wall time for the two loop in Figure 7. The times for the loop on cells vary between 1 minute for $p = 3$ and 16 threads and 1782.2 seconds (30 minutes) for $p = 1$ in serial. The loop on patches varies between 29 seconds for $p = 2$ with 16 threads, and 855 seconds (14 minutes) for $p = 5$ in serial. The overall time decreases with polynomial degree for the loop on cells, but increases for the loop on patches. However, the parallel scaling is also better for higher order for the loop on patches.

For both loops, we see that increasing the polynomial degree p improves the parallel scalability. We attribute this to the positive correlation between arithmetic intensity (number of floating point operations divided by the number of bytes accessed) and polynomial degree of discrete spaces. Higher arithmetic intensity typically leads to better performance both in serial and in parallel on

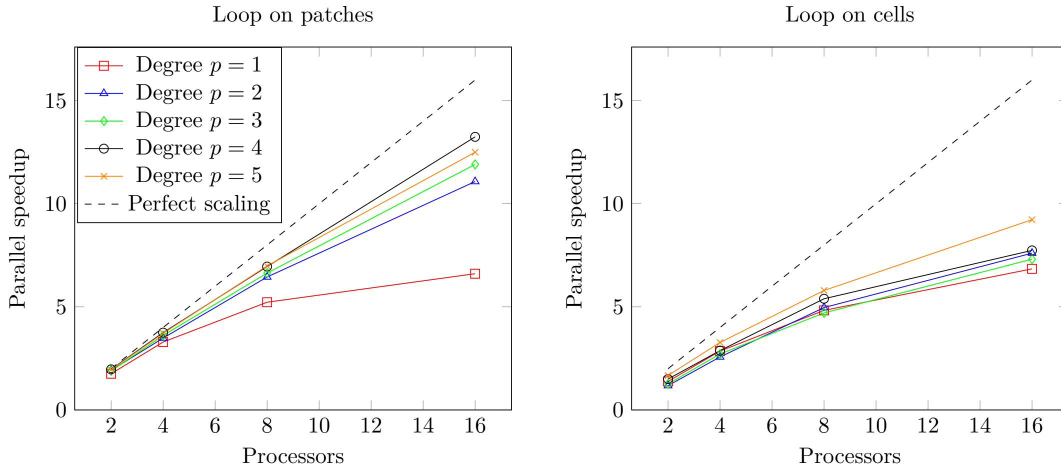

Figure 6: Parallel speedup for the two main loop in Algorithm: the loop on patches of Line 7 (left) and the loop on cells of Line 1 (right).

modern architectures. The increase of arithmetic intensity due to higher polynomial degree for matrix assembly has been studied in, e.g., [20]. For the loop on patches, the cost is dominated by the resolution of the local patch problem on Line 13. We use a direct solver (LU decomposition) for this step, and the complexity depends on the number of non-zeros in the matrix. As the polynomial degree increases, so does the matrix band b and direct solvers generally take $\mathcal{O}(b^2 n)$ where n is the number of unknowns. Thus, the arithmetic intensity increases asymptotically as b increases because the storage cost is only $\mathcal{O}(bn)$.

7 Conclusions and future work

In this paper, we have introduced a novel algorithm for computing the equilibrated flux and demonstrated its correctness by reconstructing a vector field with a prescribed divergence (up to a polynomial projection) as well achieving the theoretical optimal rate of convergence for varying polynomial degree. We also present results for improved performance as measured in terms of parallel speedup. The corresponding open source library is written entirely in Julia and interfaces directly with the `Gridap.jl` finite element library.

The setup phase of the algorithm consisting in a loop on cells has not been fully optimized, and this is reflected in less dramatic speedup in the numerical results presented herein. This could therefore be an additional improvement at the implementation level, most likely by reducing allocations. Another improvement would be to cache information for elements shared between multiple patches.

References

[1] Badia, S., and Verdugo, F. Gridap: an extensible finite element toolbox in Julia. *Journal of Open Source Software 5*, 52 (2020), 2520.

[2] Braess, D., and Schöberl, J. Equilibrated residual error estimator for edge elements. *Math. Comp. 77*, 262 (2008), 651–672.

[3] Brezzi, F., and Fortin, M. *Mixed and hybrid finite element methods*, vol. 15 of *Springer Series in Computational Mathematics*. Springer-Verlag, New York, Heidelberg, 1991.

[4] Cai, Z., and Zhang, S. Robust equilibrated residual error estimator for diffusion problems: conforming elements. *SIAM J. Numer. Anal. 50*, 1 (2012), 151–170.

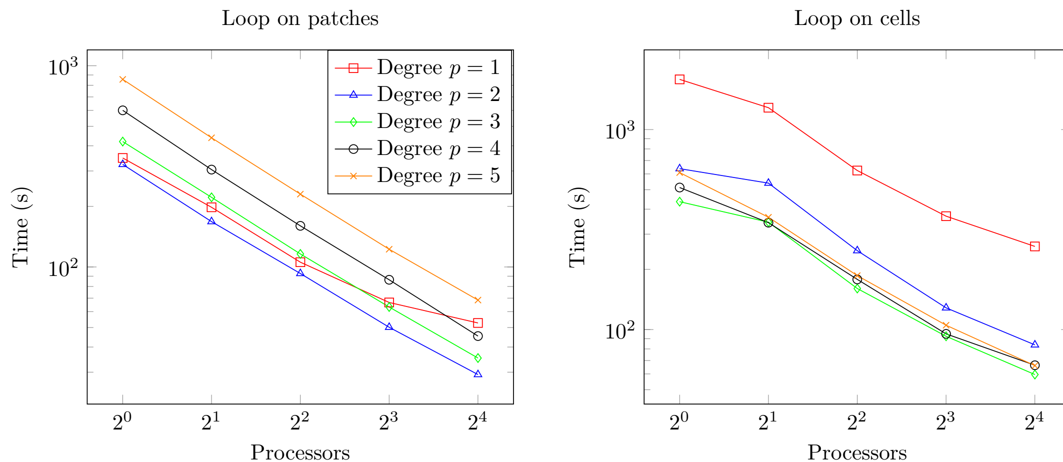

Figure 7: Wall time for the two main loop in Algorithm: the loop on patches of Line 7 (left) and the loop on cells of Line 1 (right).

[5] Cancès, E., Dusson, G., Maday, Y., Stamm, B., and Vohralík, M. Guaranteed a posteriori bounds for eigenvalues and eigenvectors: multiplicities and clusters. *Math. Comp. 89*, 326 (2020), 2563–2611.

[6] Destuynder, P., and Métivet, B. Explicit error bounds in a conforming finite element method. *Math. Comp. 68*, 228 (1999), 1379–1396.

[7] Dörfler, W. A convergent adaptive algorithm for Poisson's equation. *SIAM J. Numer. Anal. 33*, 3 (1996), 1106–1124.

[8] Ern, A., Smears, I., and Vohralík, M. Guaranteed, locally space-time efficient, and polynomial-degree robust a posteriori error estimates for high-order discretizations of parabolic problems. *SIAM J. Numer. Anal. 55*, 6 (2017), 2811–2834.

[9] Ern, A., Smears, I., and Vohralík, M. Equilibrated flux *a posteriori* error estimates in $L^2(H^1)$-norms for high-order discretizations of parabolic problems. *IMA J. Numer. Anal. 39*, 3 (2019), 1158–1179.

[10] Ern, A., and Vohralík, M. Polynomial-degree-robust a posteriori estimates in a unified setting for conforming, nonconforming, discontinuous Galerkin, and mixed discretizations. *SIAM J. Numer. Anal. 53*, 2 (2015), 1058–1081.

[11] Ern, A., and Vohralík, M. Stable broken H^1 and $H(\mathrm{div})$ polynomial extensions for polynomial-degree-robust potential and flux reconstruction in three space dimensions. *Math. Comp. 89*, 322 (2020), 551–594.

[12] Févotte, F., Rappaport, A., and Vohralík, M. Adaptive regularization, discretization, and linearization for nonsmooth problems based on primal–dual gap estimators. *Comput. Methods Appl. Mech. Eng. 418* (2024), 116558.

[13] Harnist, A., Mitra, K., Rappaport, A., and Vohralík, M. Robust energy a posteriori estimates for nonlinear elliptic problems. HAL preprint: hal-04033438, 2023.

[14] Ladevèze, P., and Leguillon, D. Error estimate procedure in the finite element method and applications. *SIAM J. Numer. Anal. 20*, 3 (1983), 485–509.

[15] Mitra, K., and Vohralík, M. A posteriori error estimates for the Richards equation, 2021.

[16] Morin, P., Siebert, K. G., and Veeser, A. A basic convergence result for conforming adaptive finite elements. *Math. Models Method Appl. Sci. 18*, 5 (2008), 707–737.

[17] Prager, W., and Synge, J. L. Approximations in elasticity based on the concept of function space. *Q. Appl. Math. 5* (1947), 241–269.

[18] Riedlbeck, R., Di Pietro, D. A., Ern, A., Granet, S., and Kazymyrenko, K. Stress and flux reconstruction in Biot's poro-elasticity problem with application to a posteriori error analysis. *Comput. Math. Appl. 73*, 7 (2017), 1593–1610.

[19] Smears, I., and Vohralík, M. Simple and robust equilibrated flux a posteriori estimates for singularly perturbed reaction-diffusion problems. *ESAIM: Mathematical Modelling and Numerical Analysis 54*, 6 (2020), 1951–1973.

[20] Sornet, G., Jubertie, S., Dupros, F., De Martin, F., and Limet, S. Performance analysis of SIMD vectorization of high-order finite-element kernels. In *2018 International Conference on High Performance Computing & Simulation (HPCS)* (2018), pp. 423–430.

[21] Verdugo, F., and Badia, S. The software design of Gridap: a finite element package based on the Julia JIT compiler. *Comput. Phys. Commun. 276* (2022), 108341.

[22] Verfürth, R. A posteriori error estimation and adaptive mesh-refinement techniques. In *Proceedings of the Fifth International Congress on Computational and Applied Mathematics (Leuven, 1992)* (1994), vol. 50, pp. 67–83.

[23] Zienkiewicz, O. C., Taylor, R. L., and Zhu, J. Z. *The finite element method: its basis and fundamentals*, seventh ed. Elsevier/Butterworth Heinemann, Amsterdam, 2013.

[24] Zienkiewicz, O. C., and Zhu, J. Z. A simple error estimator and adaptive procedure for practical engineering analysis. *Internat. J. Numer. Methods Engrg. 24*, 2 (1987), 337–357.