Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Discontinuous Galerkin discretization of magnetic convection

Term paper

Jonas Sukys

supervisors:
Prof. Dr. Ralf Hiptmair
Holger Heumann

Last update: July 21, 2010

Abstract

The main aim of this paper is a numeric solution to magnetic convection equation in 3 dimensions. Since Discontinuous Galerkin Finite Element method leads to an unstable variational problem, upwinding technique is used. For the sake of simplicity (and ease of implementation), the upwind numerical flux is rewritten using the usual average combined with a jump penalty [BMS04]. Even though on conventional non-parallel architectures the curse of dimensionality limits the accuracy of the numerical solution, the results were sufficient for simple error analysis. In most cases, improvements in convergence rates for higher polynomial degree were observed. All numerical experiments were implemented using FEniCS finite element library; a brief motivational description of its features is also provided.

Keywords: magnetic convection, Discontinous Galerkin, hyperbolic equations, jump-stabilization, upwind.

AMS Subject Classification: 65N12 65N22 65N30

Contents

List of Figures

Introduction

Let M be a smooth manifold embedded into Euclidean space $\mathbb{R}^d$. For a smooth vector field $v \in C^\infty(M, \mathbb{R}^d)$, the *Lie derivative* w.r.t. v of a p-form $\alpha \in \Omega^k(M)$ for $0 < p < d$ is again a p-form defined as:

$$\mathcal{L}_v \alpha = \mathrm{d}(\iota_v \alpha) + \iota_v \,\mathrm{d}\, \alpha \tag{0.1}$$

where:

- $\iota_v : \Omega^p(M) \to \Omega^{p-1}(M)$ is the contraction w.r.t. v (an **algebraic** operation)
- $\mathrm{d} : \Omega^p(M) \to \Omega^{p+1}(M)$ is the exterior derivative

Rigorous definitions and additional properties of contraction map and exterior derivative can be found in [Hit03].

PDEs of the form (0.1) show up in variational formulations of magnetic convection problems. In particular, interesting cases in the 3 dimensional Euclidean space occur for $p = 1$, i.e. for Lie derivatives of a 1-form.

For dimension $d = 3$, the exterior derivatives are consistent with the conventional differential operators:

$$\mathrm{d} : \Omega^0(M) \overset{\mathrm{grad}}{\to} \Omega^1(M) \overset{\mathrm{curl/rot}}{\to} \Omega^2(M) \overset{\mathrm{div}}{\to} \Omega^3(M) = \Omega^0(M) \tag{0.2}$$

Hence, for $d = 3$ and $p = 1$ identity (0.1) becomes:

$$\mathcal{L}_v \alpha \overset{(0.1),(0.2)}{=} \mathrm{d}(\underbrace{\alpha \cdot v}_{\text{0-form}}) + \iota_v \underbrace{\mathrm{curl}\, \alpha}_{\text{2-form}} \overset{(0.2)}{=} \mathrm{grad}(\alpha \cdot v) + \mathrm{curl}\, \alpha \times v \tag{0.3}$$

Here we leave the differential geometry setting aside and continue with numerical analysis of the boundary value problem associated with the differential operator in the equation (0.3).

1 Preliminaries

1.1 Problem statement

For a bounded Lipschitz domain $\Omega \in \mathbb{R}^3$ with boundary $\Gamma := \partial\Omega$, given a velocity field $v \in C^1(\overline{\Omega}, \mathbb{R}^3)$ and a function $c \in C(\Omega, \mathbb{R})$, consider a stationary PDE for a 1-form $A \in \Omega^1(\overline{\Omega})$:

$$\left.\begin{aligned} cA + \mathcal{L}_v A = cA + \operatorname{grad}(A \cdot v) + \operatorname{curl} A \times v = &\ \ 0 \quad \text{in } \Omega \\ A \times \eta = &\ \ h \quad \text{on } \Gamma_- \\ A \cdot v = &\ \ g \quad \text{on } \Gamma_- \end{aligned}\right\} \tag{1.1}$$

where $\Gamma_- \subset \partial\Omega$ is the *inflow boundary* defined as:

$$\Gamma_- := \{x \in \partial\Omega : \eta(x) \cdot v(x) < 0\} \subset \Gamma \tag{1.2}$$

For the completeness of notations, also define the *outflow boundary* as:

$$\Gamma_+ := \{x \in \partial\Omega : \eta(x) \cdot v(x) \geq 0\} \subset \Gamma \tag{1.3}$$

Remark: zero order term cA with a uniformly bounded sufficiently large c is required to ensure ellipticity. General CG and DG theory regarding well-posedness of continuous and discrete variational problems is available in [HS08].

Remark: notation $\Omega^1(M)$ denotes the space of 1-forms on a manifold M and is **not** related to Ω which is used to denote the domain for PDE.

1.2 Primal variational formulation

For simplicity and ease of notation, assume that Ω is a polygonal domain. Then let $\mathcal{M}$ be the discretization of Ω into open tetrahedra. Notice, that then Γ coincides with triangulation boundary $\overline{\partial\mathcal{M}}$ (which is not always the case for arbitrary non-polygonal domains).

Fix conventional notations:

- $\mathcal{F}(\mathcal{M})$ - set of all facets of mesh $\mathcal{M}$
- $\mathcal{F}^o(\mathcal{M})$ - set of all interior facets of mesh $\mathcal{M}$
- $\mathcal{F}^\partial(\mathcal{M})$ - set of all exterior (boundary) facets of mesh $\mathcal{M}$

Additional notations:

- $\mathcal{F}^{\partial}_{-}(\mathcal{M})$ - set of all exterior inflow boundary facets of mesh $\mathcal{M}$

$$\mathcal{F}^{\partial}_{-}(\mathcal{M}) := \{f \in \mathcal{F}^{\partial}(\mathcal{M}) : f \subset \Gamma_{-}\} \tag{1.4}$$

- $\mathcal{F}^{\partial}_{+}(\mathcal{M})$ - set of all exterior outflow boundary facets of mesh $\mathcal{M}$

$$\mathcal{F}^{\partial}_{+}(\mathcal{M}) := \mathcal{F}^{\partial}(\mathcal{M}) \setminus \mathcal{F}^{\partial}_{-}(\mathcal{M}) \tag{1.5}$$

Let $K \in \mathcal{M}$. Choose the local shape functions to be polynomials of degree $p \geq 0$. Since **discontinuous** Galerkin discretization is of particular interest for this paper, the global shape functions for test space are **discontinuous** piece-wise polynomials $A'_N \in (\mathcal{P}_p(\mathcal{M}))^3$, i.e.:

$$\forall K \in \mathcal{M} \implies A'_N|_K \in (\mathcal{P}_p(K))^3 \tag{1.6}$$

Multiplying equation (1.1) by such test function A'_N and integrating over Ω gives the following variational formulation:

seek $A \in \Omega^1(\mathbb{R}^3)$ such that $\forall A'_N \in (\mathcal{P}_p(\mathcal{M}))^3$:

$$\sum_{K \in \mathcal{M}} \int_{\Omega} (cA + \operatorname{grad}(A \cdot v) + \operatorname{curl} A \times v) \cdot A'_N dx = 0 \tag{1.7}$$

Using integration by parts, every $\operatorname{grad}(A \cdot v)$ term of the summands in (1.7) can be rewritten as:

$$\int_K \operatorname{grad}(A \cdot v) \cdot A'_N dx = -\int_K (A \cdot v) \operatorname{div} A'_N dx + \int_{\partial K} (A \cdot v)(A'_N \cdot \eta) dS \tag{1.8}$$

Likewise, the $\operatorname{curl} A \times v$ term can be rewritten as:

$$\begin{aligned} \int_K (\operatorname{curl} A \times v) \cdot A'_N dx &= \int_K (v \times A'_N) \cdot \operatorname{curl} A dx = \\ = \int_K \operatorname{curl}(v \times A'_N) \cdot A dx &- \int_{\partial K} (A \times \eta) \cdot (v \times A'_N) dS \end{aligned} \tag{1.9}$$

Definition 1.2.1 (Jumps and averages)**.** *Let $K_+, K_- \in \mathcal{M}$ be adjacent, $\varphi_i : K_i \to \mathbb{R}$, $B_i : K_i \to \mathbb{R}^3$, η_i - the unit normal vector to K_i; $i \in \{+, -\}$. Define the following maps on interface facet $f = K_+ \cap K_- \in \mathcal{F}^o(\mathcal{M})$:*

(1) Averages:

$$\{\varphi\} := \frac{1}{2}(\varphi_+ + \varphi_-), \qquad \{B\} := \frac{1}{2}(B_+ + B_-);$$

(2) *Jumps:*

$$[[\varphi]] := \varphi_+ - \varphi_-, \qquad [[B]] := B_+ - B_-$$

(3) *Jumps w.r.t. normal:*

$$[[\varphi]]_\eta := \varphi_+\eta_+ + \varphi_-\eta_-, \qquad [[B]]_\eta := B_+ \cdot \eta_+ + B_- \cdot \eta_-;$$

(4) *Tangential jumps w.r.t. normal:*

$$[[B]]_\times := B_+ \times \eta_+ + B_- \times \eta_-;$$

On the boundary facets, the corresponding values of functions are defined as:

$$\{\varphi\} := \varphi_+, \quad \{B\} := B_+, \quad [[\varphi]]_\eta := \varphi_+\eta_+, \quad [[B]]_\eta := B_+ \cdot \eta_+,$$

$$[[\varphi]] := \varphi_+, \quad [[B]] := B_+, \quad [[B]]_\times := B_+ \times \eta_+.$$

Note: Justification for jumps w.r.t. normal ($[[\cdot]]_\eta$ and $[[\cdot]]_\times$) is the invariance under choice of cell K: interchanging K_+ and K_- produces the same result. Reference: [DAM01].

Proposition 1.2.1 (Magic DG formulas)**.** *Let $\mathcal{M}$ be some bounded mesh in $\mathbb{R}^3$, η - normal outward vector to $K \in \mathcal{M}$.*

(1) *Let $A : \mathcal{M} \to \mathbb{R}^3$, $\varphi : \mathcal{M} \to \mathbb{R}$, then:*

$$\sum_{K\in\mathcal{M}} \int_{\partial K} (A \cdot \eta)\varphi dS = \sum_{f\in\mathcal{F}^o(\mathcal{M})} \int_f \{A\} \cdot [[\varphi]]_\eta dS + \sum_{f\in\mathcal{F}(\mathcal{M})} \int_f [[A]]_\eta \{\varphi\} dS \tag{1.10}$$

(2) *Let $A, B : \mathcal{M} \to \mathbb{R}^3$, then:*

$$\sum_{K\in\mathcal{M}} \int_{\partial K} (A\times\eta)\cdot B dS = -\sum_{f\in\mathcal{F}^o(\mathcal{M})} \int_f \{A\}\cdot[[B]]_\times dS + \sum_{f\in\mathcal{F}(\mathcal{M})} \int_f [[A]]_\times \cdot \{B\} dS \tag{1.11}$$

Proof. Straightforward algebraic manipulations using identity:

$$(a \times b) \cdot c = (b \times c) \cdot a \tag{1.12}$$

(1) set $LHS := (A_1 \cdot \eta_1)\varphi_1 + (A_2 \cdot \eta_2)\varphi_2$, then:

$$
\begin{aligned}
2RHS :=& \quad (A_1 + A_2) \cdot (\varphi_1\eta_1 + \varphi_2\eta_2) && + \\
+& \quad (A_1 \cdot \eta_1 + A_2 \cdot \eta_2)(\varphi_1 + \varphi_2) && = \\
=& \quad (A_1 \cdot \eta_1)\varphi_1 + (A_1 \cdot \eta_2)\varphi_2 && + \\
+& \quad (A_2 \cdot \eta_1)\varphi_1 + (A_2 \cdot \eta_2)\varphi_2 && + \\
+& \quad (A_1 \cdot \eta_1)\varphi_1 + \underbrace{(A_1 \cdot \eta_1)\varphi_2}_{=-(A_1\cdot\eta_2)\varphi_2} && + \\
+& \quad \underbrace{(A_2 \cdot \eta_2)\varphi_1}_{=-(A_2\cdot\eta_1)\varphi_1} + (A_2 \cdot \eta_2)\varphi_2 && = 2LHS;
\end{aligned}
$$

(2) set $LHS := (A_1 \times \eta_1) \cdot B_1 + (A_2 \times \eta_2) \cdot B_2$, then:

$$
\begin{aligned}
2RHS :=& \quad (A_1 \times \eta_1 + A_2 \times \eta_2) \cdot (B_1 + B_2) && - \\
-& \quad (A_1 + A_2) \cdot (B_1 \times \eta_1 + B_2 \times \eta_2) && = \\
=& \quad (A_1 \times \eta_1) \cdot B_1 + (A_2 \times \eta_2) \cdot B_1 && + \\
+& \quad (A_1 \times \eta_1) \cdot B_2 + (A_1 \times \eta_1) \cdot B_1 && - \\
-& \quad (B_1 \times \eta_1) \cdot A_1 - (B_2 \times \eta_2) \cdot A_1 && - \\
-& \quad (B_1 \times \eta_1) \cdot A_2 - (B_1 \times \eta_1) \cdot A_1 && = \\
\overset{((1.12))}{=}& \quad 2LHS && + \\
+& \quad (A_1 \times \eta_1) \cdot B_2 + (A_2 \times \eta_2) \cdot B_1 && - \\
-& \quad \underbrace{(\eta_1 \times A_2)}_{=A_2\times\eta_2} \cdot B_1 - \underbrace{(\eta_2 \times A_1)}_{=(A_1\times\eta_1)} \cdot B_2 && = 2LHS;
\end{aligned}
$$

Finally, the desired formula for both cases is achieved by integrating and summing up the identities above over internal facets $f \in \mathcal{F}^o(\mathcal{M})$ and incorporating integrals over exterior facets $f \in \mathcal{F}^\partial(\mathcal{M})$. □

Combining equations (1.8) - (1.9), applying Proposition 1.2.1 we obtain:

$$
\begin{aligned}
&\sum_{K\in\mathcal{M}} \int_K cA \cdot A'_N - (A \cdot v) \operatorname{div} A'_N + \operatorname{curl}(v \times A'_N) \cdot A dx + \\
&+ \sum_{f\in\mathcal{F}(\mathcal{M})} \int_f \{A \cdot v\}[[A'_N]]_\eta dS + \sum_{f\in\mathcal{F}^o(\mathcal{M})} \int_f [[A \cdot v]]_\eta \cdot \{A'_N\} dS + \\
&+ \sum_{f\in\mathcal{F}^o(\mathcal{M})} \int_f \{A\} \cdot [[v \times A'_N]]_\times dS - \sum_{f\in\mathcal{F}(\mathcal{M})} \int_f [[A]]_\times \cdot \{v \times A'_N\} dS = 0
\end{aligned}
\tag{1.13}
$$

Since $A, v \in C(\Omega, \mathbb{R}^3)$ and thus $A \cdot v \in C(\Omega)$, the corresponding jumps vanish on the internal facets $f \in \mathcal{F}^o(\mathcal{M})$:

$$[[A]]_\times = 0, \qquad [[A \cdot v]]_\eta = 0, \tag{1.14}$$

Hence variational formulation simplifies to:

$$\sum_{K\in\mathcal{M}} \int_K cA \cdot A'_N - (A \cdot v) \operatorname{div} A'_N + \operatorname{curl}(v \times A'_N) \cdot A dx + \\ + \sum_{f\in\mathcal{F}(\mathcal{M})} \int_f \{A \cdot v\}[[A'_N]]_\eta dS + \\ + \sum_{f\in\mathcal{F}^o(\mathcal{M})} \int_f \{A\} \cdot [[v \times A'_N]]_\times dS - \sum_{f\in\mathcal{F}^\partial(\mathcal{M})} \int_f [[A]]_\times \cdot \{v \times A'_N\} dS = 0 \tag{1.15}$$

Remark: the idea to make use of (1.14) comes from [BMS04].

2 Discretization

2.1 Discrete variational formulation

We would like to discretize the formal variational formulation restricting A to a subspace $(\mathcal{P}_p(\mathcal{M}))^3$:

seek for $A_N \in (\mathcal{P}_p(\mathcal{M}))^3$ such that $\forall A'_N \in (\mathcal{P}_p(\mathcal{M}))^3$:

$$\sum_{K\in\mathcal{M}} \int_K cA_N \cdot A'_N - (A_N \cdot v) \operatorname{div} A'_N + \operatorname{curl}(v \times A'_N) \cdot A_N dx + \\ + \sum_{f\in\mathcal{F}(\mathcal{M})} \int_f \{A_N \cdot v\}[[A'_N]]_\eta dS + \\ + \sum_{f\in\mathcal{F}^o(\mathcal{M})} \int_f \{A_N\} \cdot [[v \times A'_N]]_\times dS - \sum_{f\in\mathcal{F}^\partial(\mathcal{M})} \int_f [[A_N]]_\times \cdot \{v \times A'_N\} dS = 0 \tag{2.1}$$

Unfortunately, we have $\operatorname{curl}(v \times A'_N)$ in the expression which is not in general available as a finite element for arbitrary v. Applying integration by parts once again, we obtain:

$$\int_K \operatorname{curl}(v \times A'_N) \cdot A_N dx = \int_K (v \times A'_N) \cdot \operatorname{curl} A_N dx + \int_{\partial K} (A_N \times \eta) \cdot (v \times A'_N) dS \tag{2.2}$$

Applying Proposition 1.2.1 to the sum of boundary integrals in (2.2), it becomes:

$$-\sum_{f\in\mathcal{F}^o(\mathcal{M})}\int_f \{A_N\}\cdot[[v\times A'_N]]_\times dS+\sum_{f\in\mathcal{F}(\mathcal{M})}\int_f [[A_N]]_\times\cdot\{v\times A'_N\}dS \quad (2.3)$$

Notice, that when plugged into the discrete variational formulation, the following terms cancel out:

$$\sum_{f\in\mathcal{F}^o(\mathcal{M})}\int_f \{A_N\}\cdot[[v\times A'_N]]_\times dS \quad \text{and} \quad \sum_{f\in\mathcal{F}^\partial(\mathcal{M})}\int_f [[A_N]]_\times\cdot\{v\times A'_N\}dS$$

Therefore, the final expression for discrete variational formulation with already incorporated boundary conditions from (1.1) is of the following form:

$$\sum_{K\in\mathcal{M}}\int_K cA_N\cdot A'_N-(A_N\cdot v)\operatorname{div}A'_N+(v\times A'_N)\cdot\operatorname{curl}A_N dx+$$
$$+\sum_{f\in\mathcal{F}(\mathcal{M})\backslash\mathcal{F}^\partial_-(\mathcal{M})}\int_f\{A_N\cdot v\}[[A'_N]]_\eta dS+\sum_{f\in\mathcal{F}^o(\mathcal{M})}\int_f[[A_N]]_\times\cdot\{v\times A'_N\}dS=$$
$$=-\sum_{f\in\mathcal{F}^\partial_-(\mathcal{M})}\int_f g(A'_N\cdot\eta)dS \quad (2.4)$$

Lemma 2.1.1. *For a vector field B and a vector η in $\mathbb{R}^3$, the following holds:*

$$(\eta\times(\eta\times B))=-(\eta\cdot\eta)B+(\eta\cdot B)\eta; \quad (2.5)$$

additionally assuming that n is a unit vector, we have the identity:

$$B=(\eta\cdot B)\eta-(\eta\times(\eta\times B)). \quad (2.6)$$

Applying Lemma 2.1.1 to A_N, A'_N, v and η as in context of formulation (2.4), we obtain the relation:

$$(v\cdot\eta)(A_N\cdot A'_N)=(v\cdot A_N)(A'_N\cdot\eta)-(A_N\times\eta)(v\times A'_N) \quad (2.7)$$

Integrating identity (2.7) over $\mathcal{F}^\partial_-(\mathcal{M})$ and adding to variational formulation

(2.4), it becomes:

$$\begin{aligned}
&\sum_{K\in\mathcal{M}}\int_K cA_N\cdot A'_N-(A_N\cdot v)\operatorname{div}A'_N+(v\times A'_N)\cdot\operatorname{curl}A_N dx+\\
&+\sum_{f\in\mathcal{F}(\mathcal{M})}\int_f\{A_N\cdot v\}[[A'_N]]_\eta dS+\sum_{f\in\mathcal{F}^o(\mathcal{M})}\int_f[[A_N]]_\times\cdot\{v\times A'_N\}dS-\\
&\qquad-\sum_{f\in\mathcal{F}^\partial_-(\mathcal{M})}\int_f(v\cdot\eta)(A_N\cdot A'_N)dS=\\
&=-\sum_{f\in\mathcal{F}^\partial_-(\mathcal{M})}\int_f g(A'_N\cdot\eta)dS+\sum_{f\in\mathcal{F}^\partial_-(\mathcal{M})}\int_f h\cdot(v\times A'_N)dS
\end{aligned}\tag{2.8}$$

2.2 Stabilization by upwinding

The resulting numerical solution from (2.8) is only $L^2(\mathcal{M})$-stable. To obtain the stability in a stronger norm, the "*upwinding*" technique is used. More precisely, $\{A_N\cdot v\}$ and $[[A_N]]_\times$ in equation (2.8) are replaced with $\{A_N\}_u\cdot v$ and $2\{A_N\}_u\times\eta$ respectively, where:

Definition 2.2.1. *Setting as in the definition 1.2.1, the* ***upwind*** *value is:*

$$\{\varphi\}_u:=\begin{cases}\varphi_1 & \text{if } \eta_1\cdot v>0,\\ \varphi_2 & \text{if } \eta_1\cdot v<0,\\ \{\varphi\} & \text{if } \eta_1\cdot v=0.\end{cases}$$

The upwind value for a vector-valued function is defined analogously.

Definition 2.2.2. *Weighted average is defined as:*

$$\{\varphi\}_\alpha:=\alpha_+\varphi_++\alpha_-\varphi_-,\tag{2.9}$$

where $\alpha_+,\alpha_-\geq 0,\alpha_++\alpha_-=1$.
The weighted average for a vector-valued function is defined analogously.

Notice, that (also used in [Mar06]):

$$\{\varphi\}_u=\{\varphi\}_\alpha=\{\varphi\}+\frac{[[\alpha]]}{2}[[\varphi]]\tag{2.10}$$

for $\alpha=(\operatorname{sign}(v\cdot\eta)+1)/2$.
Remark: $\{\varphi\}_u=\varphi_++\frac{1}{2}\varphi_+\neq\varphi_+$ on outflow boundary facets!

Recalling the notations from (1.5), the stabilized form of the discrete variational formulation (2.8) becomes:

$$\begin{aligned}\sum_{K\in\mathcal{M}}\int_K cA_N\cdot A'_N-(A_N\cdot v)\operatorname{div}A'_N+(v\times A'_N)\cdot\operatorname{curl}A_N dx+\\+\sum_{f\in\mathcal{F}(\mathcal{M})}\int_f(\{A_N\}_u\cdot v)[[A'_N]]_\eta dS+\sum_{f\in\mathcal{F}^o(\mathcal{M})}\int_f(2\{A_N\}_u\times\eta)\cdot\{v\times A'_N\}dS-\\-\sum_{f\in\mathcal{F}^\partial_-(\mathcal{M})}\int_f(v\cdot\eta)(A_N\cdot A'_N)dS=\\=-\sum_{f\in\mathcal{F}^\partial_-(\mathcal{M})}\int_f g(A'_N\cdot\eta)dS+\sum_{f\in\mathcal{F}^\partial_-(\mathcal{M})}\int_f h\cdot(v\times A'_N)dS\end{aligned}\tag{2.11}$$

Alternatively, stabilization can be done by adding term

$$\sum_{f\in\mathcal{F}^o(\mathcal{M})}\int_f c_f([[A_N]]\cdot[[A'_N]])dS\tag{2.12}$$

to equation (2.8), where the stabilization parameter c_f related to the full upwinding was chosen to be uniformly constant on each facet (refer to [BMS04]):

$$c_f=\frac{1}{2}|v\cdot\eta|\tag{2.13}$$

and provides us with the final variational form directly used in implementation:

$$\begin{aligned}\sum_{K\in\mathcal{M}}\int_K cA_N\cdot A'_N-(A_N\cdot v)\operatorname{div}A'_N+(v\times A'_N)\cdot\operatorname{curl}A_N dx+\\+\sum_{f\in\mathcal{F}(\mathcal{M})}\int_f\{A_N\cdot v\}[[A'_N]]_\eta dS+\sum_{f\in\mathcal{F}^o(\mathcal{M})}\int_f[[A_N]]_\times\cdot\{v\times A'_N\}dS-\\-\sum_{f\in\mathcal{F}^\partial_-(\mathcal{M})}\int_f(v\cdot\eta)(A_N\cdot A'_N)dS+\sum_{f\in\mathcal{F}^o(\mathcal{M})}\int_f\frac{1}{2}|v\cdot\eta|([[A_N]]\cdot[[A'_N]])dS=\\=-\sum_{f\in\mathcal{F}^\partial_-(\mathcal{M})}\int_f g(A'_N\cdot\eta)dS+\sum_{f\in\mathcal{F}^\partial_-(\mathcal{M})}\int_f h\cdot(v\times A'_N)dS\end{aligned}\tag{2.14}$$

3 Implementation

3.1 Test problems

Several test cases were available for computations. All test were conducted on the unit cube mesh $\mathcal{M}$ in 3D (the convex hull of $\{e_1, e_2, e_3\}$). For a specific test, the velocity field v, large enough (ensuring ellipticity) factor c and the desired exact solution A were fixed. Afterwards, values for h, g on $\mathcal{F}_{-}^{\partial}(\mathcal{M})$ and right hand side f on Ω were computed using two different methods:

- numerical approximation of f using `Dolfin` from FEniCS (see 3.2)
- analytic exact expression of f using *Wolfram Mathematica* symbolic software (see Appendix C)

For detailed information on test problems refer to the source code `[tests.py]` in Appendix A.

As shortly mentioned above, test problems were implemented using FEniCS (see 3.2) finite element library. The validity of the algorithm was confirmed; additionally, various error norm convergence rates were obtained and analysed.

Linear solver used: `GMRES` (**G**eneralized **M**inimum **R**esidual) with `ILU` (**I**ncomplete **LU**-factorization) preconditioner from `PETSc` back-end.

3.2 FEniCS

Key features of FEniCS:

- `Python` and `C++` interface (can be combined with `SciPy` and `MatPlotLib`)
- math-like syntax for variational forms (`UFL + FFC`)
- custom linear algebra back-ends:
  `uBLAS, PETSc, SLEPc, Epetra, MTL4, UMFPACK`
- additional mesh manipulation libraries: `CGAL, SCOTCH`
- visualization via `VTK`
- easy implementation of subdomains, boundary parts, variable coefficients
- `MPI` support

For more detailed information, refer to [AL10, Lan09].

4 Results

4.1 Error norms

Denote the error field by:

$$E := A_{\text{approx.}} - A_{\text{exact}} \tag{4.1}$$

Absolute errors were computed using two different norms:

- $(L^2(\mathcal{M}))^3$-norm:

$$\|E\|_{(L^2(\mathcal{M}))^3} := \left(\sum_{K \in \mathcal{M}} \int_K E \cdot E dx \right)^{\frac{1}{2}} \tag{4.2}$$

- $(H(\text{curl}; \mathcal{M}))^3$-semi-norm:

$$|E|_{(H(\text{curl};\mathcal{M}))^3} := \left(\sum_{K \in \mathcal{M}} \int_K \operatorname{curl} E \cdot \operatorname{curl} E dx \right)^{\frac{1}{2}} \tag{4.3}$$

Absolute jump errors were also computed using two different norms:

- $(L^2(\mathcal{F}^o(\mathcal{M})))^3$-norm of the jump $[[E]]$:

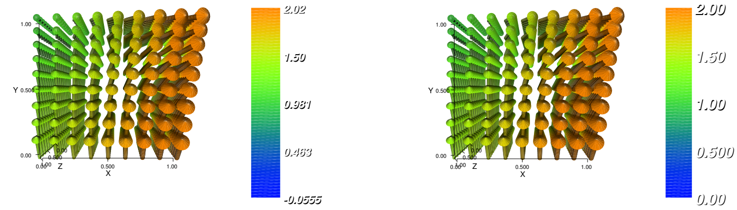
$$\|[[E]]\|_{(L^2(\mathcal{F}^o(\mathcal{M})))^3} := \left(\sum_{f \in \mathcal{F}^o(\mathcal{M})} \int_f [[E]] \cdot [[E]] dS \right)^{\frac{1}{2}} \tag{4.4}$$

- $(L^2(\mathcal{F}^\partial(\mathcal{M})))^3$-norm of the jump $[[E]]$:

$$\|[[E]]\|_{(L^2(\mathcal{F}^\partial(\mathcal{M})))^3} := \left(\sum_{f \in \mathcal{F}^\partial(\mathcal{M})} \int_f [[E]] \cdot [[E]] dS \right)^{\frac{1}{2}} \tag{4.5}$$

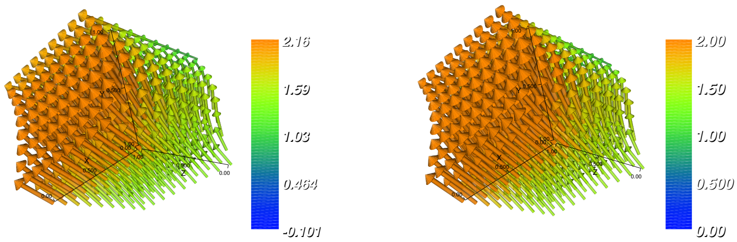
All **error plots** are w.r.t $1/h$, i.e. **w.r.t. number of mesh cells in one space dimension**. Number of degrees of freedom in this case is of order $(1/h)^3$.

4.2 Plots of numerical and exact solutions



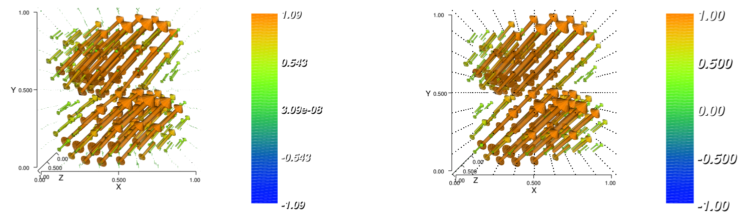
(a) numerical (b) exact

Figure 4.2.1: DG solution for polynomial v and A



(a) numerical (b) exact

Figure 4.2.2: DG solution for non-linear v and A



(a) numerical (b) exact

Figure 4.2.3: DG solution for zero-bc v and A

4.3 Test: non-linear

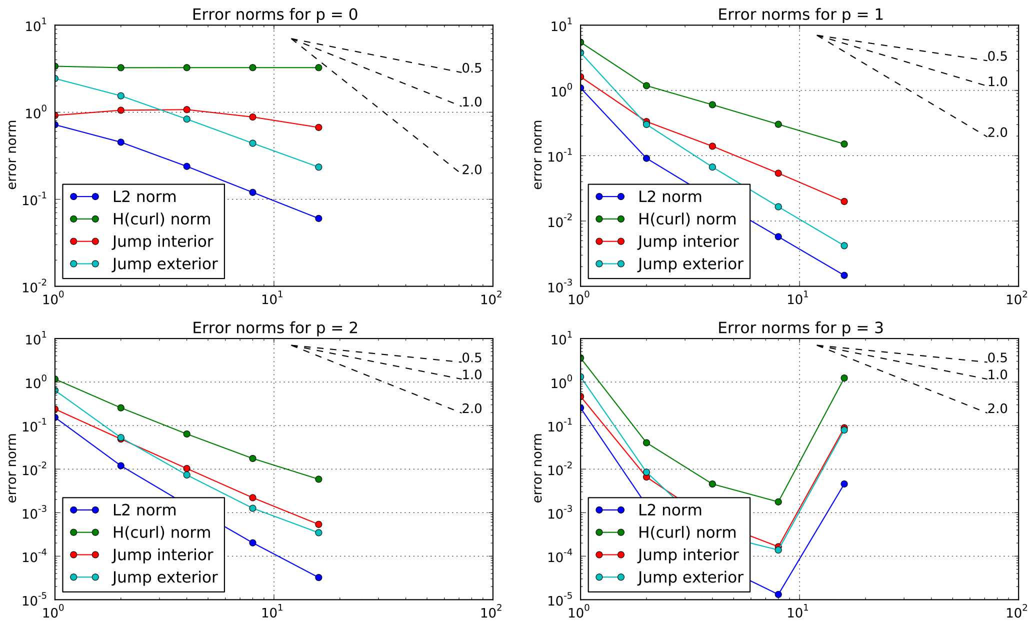


Figure 4.3.1: Error norm convergence for non-linear test

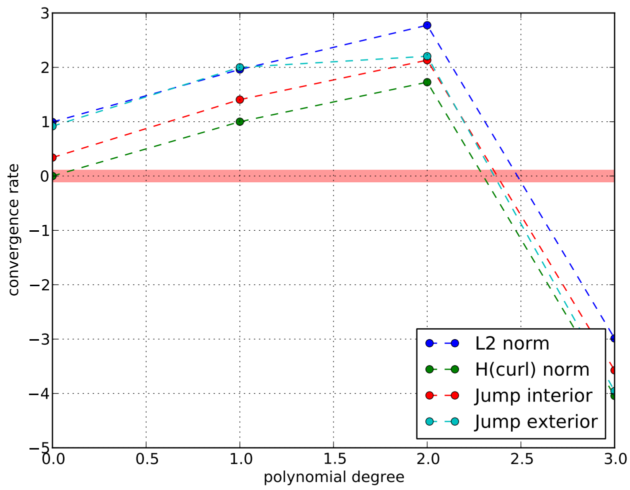


Figure 4.3.2: Convergence rates for non-linear test

4.4 Test: zero boundary condition

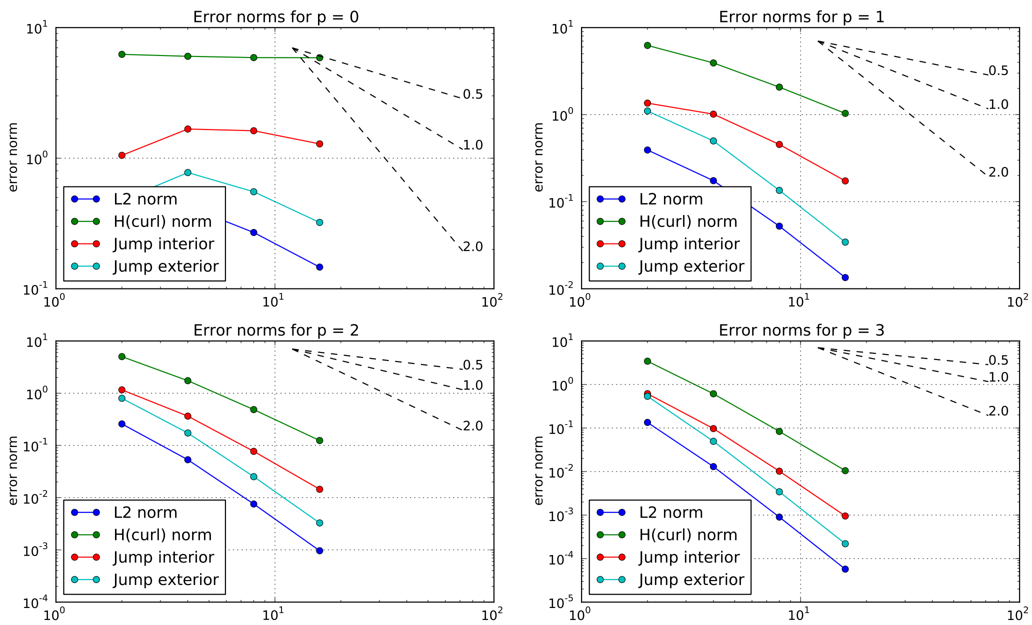


Figure 4.4.1: Error norm convergence for zero-bc test

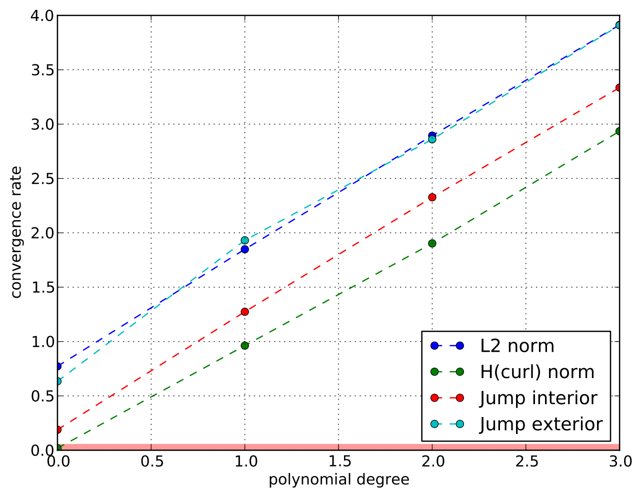


Figure 4.4.2: Convergence rates for zero-bc test

4.5 Asymptotic error behaviour

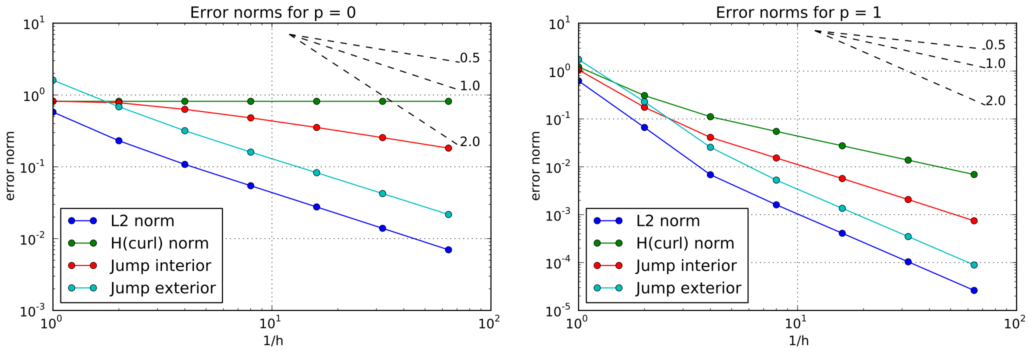


Figure 4.5.1: Error norm convergence for refined polynomial test

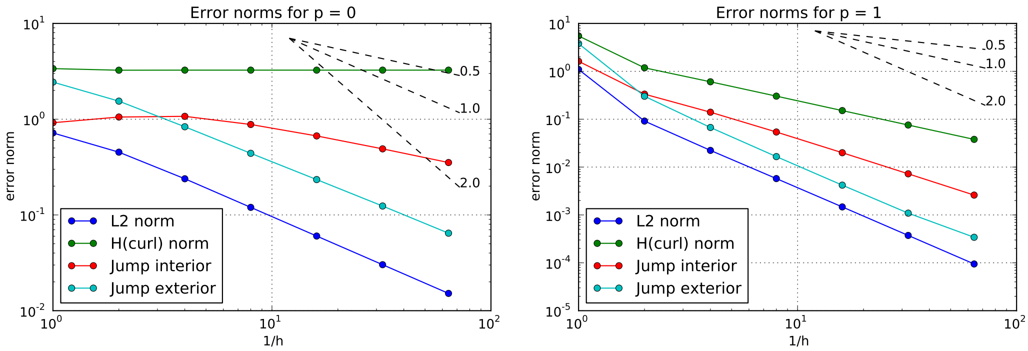


Figure 4.5.2: Error norm convergence for refined non-linear test

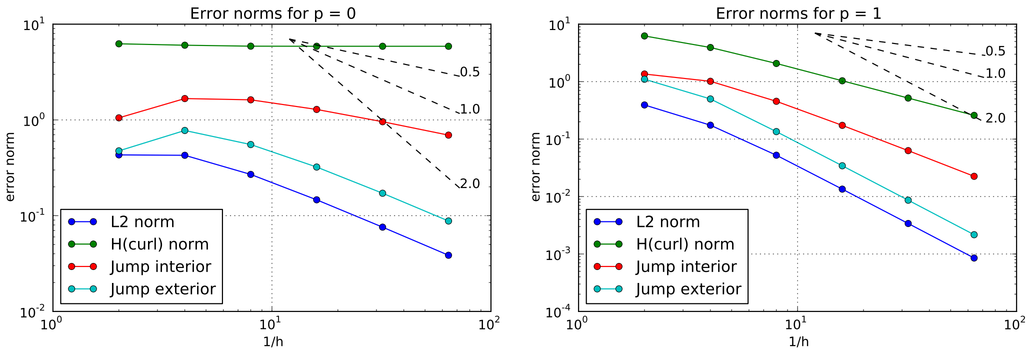


Figure 4.5.3: Error norm convergence for refined zero-bc test

4.6 Error behaviour for p = 0

Asymptotic convergence for higher polynomial degrees is hard to monitor due to curse of dimensionality and limited computing resources. Hence, we additionally elaborate a bit more on the case $p = 0$.

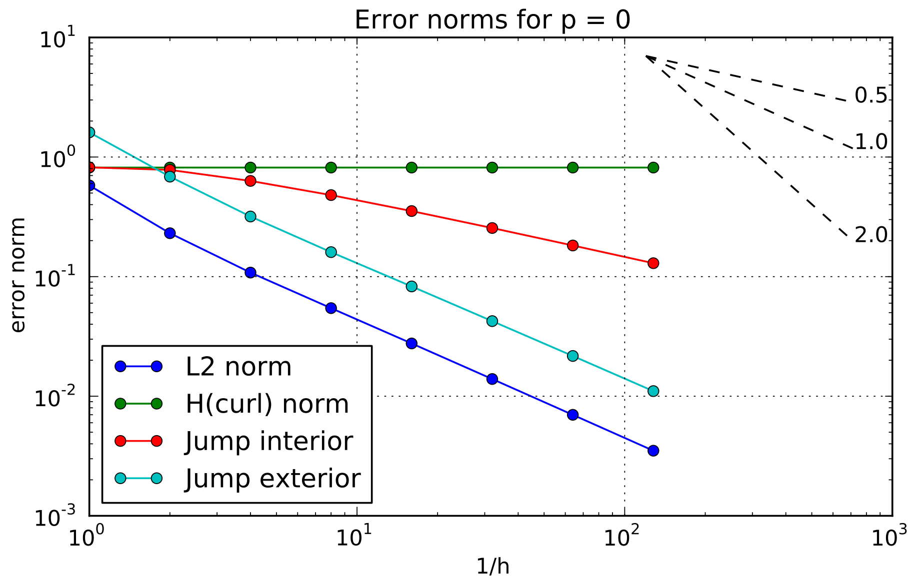


Figure 4.6.1: Error norm convergence for maximally refined polynomial test

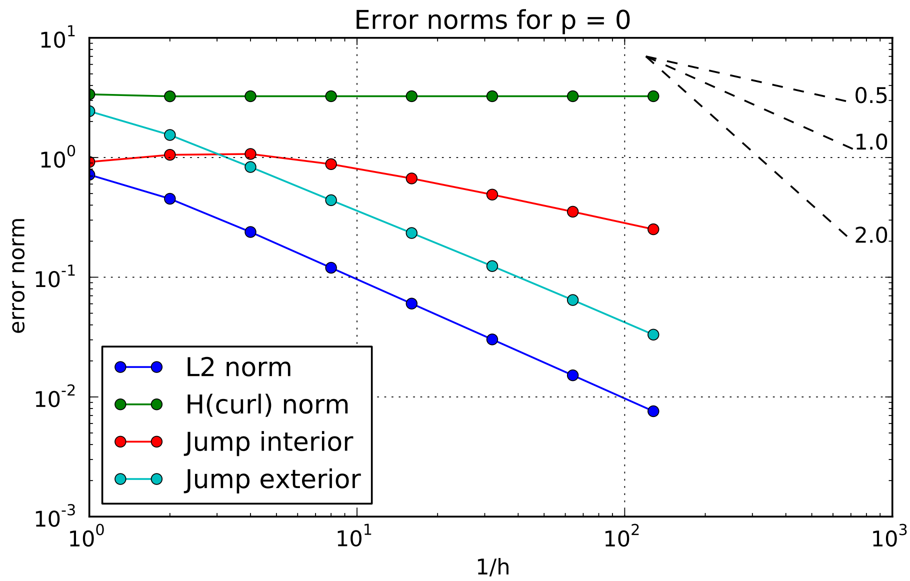


Figure 4.6.2: Error norm convergence for maximally refined non-linear test

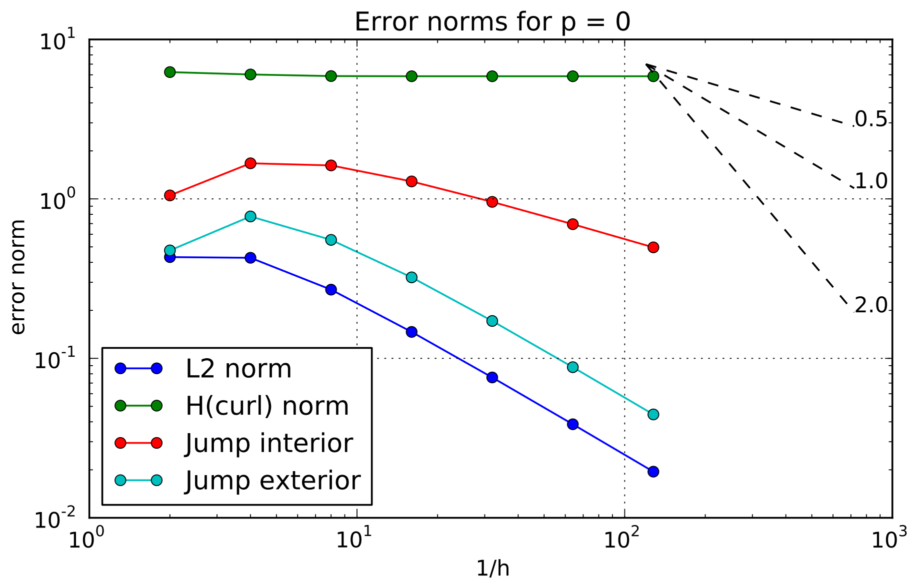


Figure 4.6.3: Error norm convergence for maximally refined zero-bc test

5 Conclusions

Order of convergence increases with polynomial degree p and empirical relation for every error norm is:

$(L^2(\mathcal{M}))^3$**-norm**	$2p$
$(H(\text{curl};\mathcal{M}))^3$**-semi-norm**	p
$(L^2(\mathcal{F}^o(\mathcal{M})))^3$**-norm**	$p+1/2$
$(L^2(\mathcal{F}^\partial(\mathcal{M})))^3$**-norm**	$2p$

Remark: The convergence rates above are w.r.t $1/h$, i.e. **w.r.t. number of mesh cells in one space dimension**. Dividing them by 3 gives convergence rates w.r.t. number of degrees of freedom.

6 Further research

- Scalability of the code. Most of the CPU time is spent on:
  - assembling matrices: finite elements can be easily distributed over nodes
  - solving linear system: configure `PETSc` solver as a back-end and enable its parallel capabilities
- Zero order term. Getting rid of the cA term in PDE (1.1).
- Diffusion term. Introducing additional $\epsilon\,\text{curl}\,\text{curl}\,A$ term in PDE (1.1).
- Consistency. Come up with a test having load vector $f = 0$ (not so easy for non-linear velocity fields v) and monitor error behaviour, since $f = 0$ shows up in magnetic convection equations.

References

[AL10] G. N. Wells A. Logg. Fenics documentation, July 2010. Available from: `http://www.fenics.org/wiki/Documentation`.

[BMS04] F. Brezzi, L. D. Marini, and E. Sli. Discontinuous galerkin methods for first-order hyperbolic problems. *Math. Models Methods Appl. Sci*, 14, 2004.

[DAM01] Bernardo Cockburn Douglas Arnold, Franco Brezzi and Donatella Marini. Discontinuous galerkin methods for elliptic problems. -, 2001.

[Hit03] Nigel Hitchin. Differentiable manifolds (lectures notes), 2003. Available from: people.maths.ox.ac.uk/∼hitchin/hitchinnotes.

[HS08] Ralf Hiptmair and Christoph Schwab. *Numerical Methods for Elliptic and Parabolic Boundary Value Problems.* SAM, ETH Zurich, 2008. Lecture slides. Available from: www.sam.math.ethz.ch/∼hiptmair.

[Lan09] Hans Petter Langtangen. Fenics tutorial, November 2009. Available from: www.fenics.org.

[Mar06] L. Donatela Marini. Discontinuous galerkin methods for advection-diusion-reaction problems. In *17th International Conference on Domain Decomposition Methods*, 2006.

Mathematics Department
Swiss Federal Institute of Technology
Ramistrasse 101, CH-8092 Zurich, Switzerland
`jonas.sukys@sam.math.ethz.ch`

Appendices

A FEniCS/Python source codes

Source code is available for download at:

`www.sukys.lt/search/label/publications`

Listing 1: [config.py] main configuration:

```
# === MAIN CONFIGURATION ===

# === PARAMETERS [ feel free to play around ] ===

test           = 'polynomial'    # default test to use: polynomial, non-linear, zero-bc
P              = 1               # polynomial degree to be used
levels         = 3               # number of refinement levels

pylab_on       = True            # plots errors and convergence rates (using pylab)
show_fig       = True            # shows all pylab plots in xsession
save_fig       = False           # saves all pylab plots into files

viper_on       = False           # plots the exact and approximated fields
save_viper     = False           # saves all viper plots into files | set to 'False' to prevent HUGE files in 'viper' directory

errors_skip    = 0               # number of first error norms to ignore when plotting
errors_slopes  = [0.5, 1.0, 2.0] # slopes to be added to plots as a referece for convergence rates

rates_wrt      = 'hmax'          # compute convergence rates w.r.t.: 'hmax' OR 'ndofs'
rates_method   = 'average'       # compute convergence rates from:  'last' 2 errors OR 'average' of all errors
rates_skip     = 0               # number of first error norms to ignore when computing AVERAGE convergence rates for plots

compute_all    = True            # computes solution for polynomial degress p = 0, 1, 2, ..., P and plots the order of convergence for every error
debug_on       = False           # includes debugging branches ( plots inflow boundary parts, etc. )

level_offset   = 0               # first refinement level
stability      = 1               # stability parameter

# === CONSTANTS [ no need to modify in general ] ===

FE_TYPE        = 'DG'            # finite elements to use: 'DG' - Discontinuous Galerkin; 'CG' - Continuous Galerkin
RHS_BACKEND    = 'Mathematica'   # compute load vector for the right hand side using: 'Dolfin' OR 'Mathematica'

QUAD_FACTOR    = 1               # parameters for quadrature order (for error norm computation)
QUAD_OFFSET    = 2               # parameters for quadrature order (for error norm computation)

FIGURE_FORMAT  = 'pdf'           # default format for saving figures to files
ERRORS_FORMAT  = '%1.0e'         # format string for errors
RATES_FORMAT   = '%+1.2f'        # format string for convergence rates
SUFFIX_FORMAT  = '_test=%s_P=%d_levels=%d' # output file suffix format | usage: SUFFIX_FORMAT % (test, P, levels)

VIPER_PATH     = 'viper/'        # default path for saving viper VTK files
PYLAB_PATH     = 'pylab/'        # default path for saving pylab plots
TEXT_PATH      = 'text/'         # default path for saving text output
DUMP_PATH      = 'dump/'         # default path for saving dumped objects

# === CONSISTENCY CHECKS [ do NOT modify] ===

if FE_TYPE == 'CG':
  SUFFIX_FORMAT += '_CG'         # to avoid conflicts with 'DG' FE_TYPE
  MIN_P          = 1
else:
  MIN_P          = 0             # minimal polynomial degree for FE space
```

Listing 2: `[initialization.py]`: initialization routines:

```
# === INITIALIZATION ROUTINES ===

from config import *
from auxiliary import initializeList, getRefinements
from errors import TYPE_SIZE
import sys

try:
  test    = sys.argv[1]              # test to use
  P       = int(sys.argv[2])         # polynomial degree
  levels  = int(sys.argv[3])         # number of refinements
except:
  print '\n=== SETTING PARAMETERS FROM: [config.py] ===\n' + \
    'Alternative usage: main.py [test_name] [pol_degree] [number_of_refinements] [env]\n'

try:
  env = sys.argv[4]                  # environment: 'cluster' OR 'xsession'
except:
  env = 'xsession'

if env == 'cluster':
  print '\n=== CLUSTER ENVIRONMENT: DISABLING VIPER & PYLAB ===\n'
  pylab_on      = False
  viper_on      = False

if RHS_BACKEND == 'Dolfin':
  print '=== LOAD VECTOR BACKEND: DOLFIN (FEniCS) ===\n'
else: # RHS_BACKEND == 'Mathematica'
  print '=== LOAD VECTOR BACKEND: Mathematica (Wolfram) ===\n'

SUFFIX       = SUFFIX_FORMAT % (test, P, levels)

# initializing lists
errors       = initializeList(P+1, TYPE_SIZE)
rates        = initializeList(P+1, TYPE_SIZE)
ndofs        = initializeList(P+1, 0)
hmax         = initializeList(P+1, 0)
cpu_time     = initializeList(P+1, 0)

refinements  = getRefinements(level_offset, levels)
```

Listing 3: `[main.py]` main source code:

```
"""
For the detailed explanation refer to paper:
"Discontinuous Galerkin discretization of magnetic convection"
available at: [pdf] http://sukys.lt/search/label/publications
              [doi] !!! add link here !!!
"""

__author__    = "Jonas Sukys (sukys.jonas@gmail.com)"
__date__      = "2010-02-19 -- 2010-08-01"
__copyright__ = "Copyright (C) 2010 Jonas Sukys, D-MATH, ETH Zurich"
__license__   = "GNU LGPL Version 2.1"

# === IMPORTS ===
from dolfin import *
import sys
import time

# === LOCAL IMPORTS ===
from config import *
from tests import *
from errors import *
from debug import *
from auxiliary import *
from dumpload import *

# === INITIALIZATION ===
from initialization import *

# === BEGIN ITERATION over polynomial degrees
for p in range(MIN_P, P+1):

  # timer for CPU effort consumption
  cpu_time_start = time.clock()

  # compute only for the polynomial degree p = P
  if not compute_all: p = P
```

```
# quadrature accuracy order
quad_degree = QUAD_FACTOR * p + QUAD_OFFSET

# === BEGIN ITERATION over refinement levels
for M in refinements:

  # === PROBLEM DATA ===

  # mesh and function space
  mesh = UnitCube(M, M, M)
  hmax[p].append(mesh.hmax())

  V  = VectorFunctionSpace(mesh, FE_TYPE, p)
  ndofs[p].append(V.dim())

  # velocity field
  v = getVelocity(test, quad_degree)

  # ellipticity fix
  c = getZeroOrderTerm(test)

  # exterior normal
  n = FacetNormal(mesh)

  # inflow boundary
  class Boundary_Inflow(SubDomain):
    def __init__(self, normal, velocity):
      self.n = normal
      self.v = velocity
      SubDomain.__init__(self)

    def inside(self, x, on_boundary):
      return on_boundary and dot3D(normal3D(x), self.v(x)) < DOLFIN_EPS

  # === PROBLEM PARAMETERS ===

  # vector field on inflow boundary
  A_0 = getSolution(test, quad_degree)

  # marking boundary parts
  boundary_parts = MeshFunction('uint', mesh, mesh.topology().dim()-1)
  boundary_parts.set_all(1)
  inflow = Boundary_Inflow(n, v)
  inflow.mark(boundary_parts, 0)

  # === VARIATIONAL FORMULATION ===
  A_ = TestFunction(V)
  A = TrialFunction(V)

  def upwind(u, v, n):
    return avg(u) + jump((sgn(dot(v, n)) + 1)/2)/2 * jump(u)

  def jump_cross(u, n):
    return cross(u('+'), n('+')) + cross(u('-'), n('-'))

  c_f = stability * 0.5 * abs(dot(v('+'), n('+')))

  a = c*dot(A,A_)*dx - dot(A,v)*div(A_)*dx + dot(cross(v,A_), curl(A))*dx \
    + avg(dot(A,v))*jump(A_,n)*dS + dot(A,v)*dot(A_,n)*ds(0) + dot(A,v)*dot(A_,n)*ds
        (1) \
    + dot( jump_cross(A, n), avg(cross(v, A_)) )*dS \
    - dot(v, n)*dot(A, A_)*ds(0) \
    + c_f*dot(jump(A), jump(A_))*dS

  # boundary conditions
  g = dot(A_0,v)
  h = cross(A_0,n)

  # source term
  if RHS_BACKEND == 'Dolfin':
    f = getRHS_Dolfin(A_0, A_, v)
  else: # RHS_BACKEND == 'Mathematica'
    f = dot( getRHS_Mathematica(test, quad_degree), A_ )

  L = c*dot(A_0,A_)*dx + f*dx - g*dot(A_,n)*ds(0) + dot(h,cross(v, A_))*ds(0)

  problem = VariationalProblem(a, L, exterior_facet_domains=boundary_parts)

  # configuring solver for the linear system
  problem.parameters['linear_solver'] = 'iterative'
  itsolver = problem.parameters['krylov_solver']
  itsolver['absolute_tolerance']  = 1e-15 # default: 1e-15
```

```
    itsolver['relative_tolerance']  = 1e-6  # default: 1e-6
    itsolver['divergence_limit']    = 10000.0 # default: 10000.0
    itsolver['gmres_restart']       = 30  # default: 30
    itsolver['monitor_convergence'] = True  # default: False
    itsolver['report']              = True  # default: True

    # compute solution
    A_discrete = problem.solve()

    # === A POSTERIORI ERROR ANALYSIS ===

    computeErrors(errors, p, A_discrete, A_0, mesh, quad_degree, FE_TYPE)

  # END ITERATION over refinement levels

  # computing convergence rates w.r.t. mesh size h
  computeConvergenceRates(rates, errors, p, hmax, rates_wrt)

  # compute CPU effort
  cpu_time[p] = time.clock() - cpu_time_start

  if not compute_all: break

# END ITERATION over polynomial degrees

# === TEXT OUTPUT to stdout and fout ===

output('ERRORS', errors, compute_all, cpu_time, TEXT_PATH, SUFFIX, ERRORS_FORMAT)
output('RATES',  rates,  compute_all, cpu_time, TEXT_PATH, SUFFIX, RATES_FORMAT)

# === DUMPING RESULTS to files ===

dump(errors, DUMP_PATH + 'ERRORS' + SUFFIX)
dump(rates,  DUMP_PATH + 'RATES'  + SUFFIX)

# === PYLAB PLOTTING & SAVING ===

if pylab_on:

  plotErrors(refinements, errors, compute_all, errors_skip, errors_slopes, 'Error norms'
      , show_fig, save_fig, PYLAB_PATH, SUFFIX, FIGURE_FORMAT)
  if compute_all:
    plotConvergenceRates(rates, rates_method, rates_skip, '', show_fig, save_fig,
        PYLAB_PATH, SUFFIX, FIGURE_FORMAT)

# === VIPER PLOTTING ===

if viper_on:

  plot(v, title='Velocity field', mesh=mesh, axes=True)
  if debug_on: plotInflowSurface(n, v, mesh)
  plot(A_0, title='Exact solution', mesh=mesh, axes=True)
  try:
    plot(A_discrete, title='Discrete solution', axes=True)
  except:
    print '\nERROR :: Discrete solution was not computed!\n'
  interactive()

# === VIPER SAVING ===

if save_viper:

  file = File(VIPER_PATH + 'velocity' + SUFFIX + '.pvd')
  file << interpolate(v, V)

  file = File(VIPER_PATH + 'exact'    + SUFFIX + '.pvd')
  file << interpolate(A_0, V)

  file = File(VIPER_PATH + 'discrete' + SUFFIX + '.pvd')
  file << A_discrete
```

Listing 4: `[auxiliary.py]`: auxiliary routines:

```
# === AUXILIARY ROUTINES ===

def dot3D(u,v):
  return u[0]*v[0] + u[1]*v[1] + u[2]*v[2]

# for unit cube only!
def normal3D(x):

  from dolfin import DOLFIN_EPS
```

```
  n = x
  for i in range(0, 2):
    if abs(x[0]) < DOLFIN_EPS:
      return (-1,0,0)
    elif abs(x[1]) < DOLFIN_EPS:
      return (0,-1,0)
    elif abs(x[2]) < DOLFIN_EPS:
      return (0,0,-1)
    elif abs(x[0] - 1) < DOLFIN_EPS:
      return (1,0,0)
    elif abs(x[1] - 1) < DOLFIN_EPS:
      return (0,1,0)
    elif abs(x[2] - 1) < DOLFIN_EPS:
      return (0,0,1)

def sgn(x):
  return 0 if x == 0 else x/abs(x)

# returns list of numbers of the form: 2**(offset + level)
def getRefinements(offset, levels):
  M_all = range(offset, offset + levels)

  for i in range(len(M_all)):
    M_all[i] = 1 << M_all[i]

  return M_all

def initializeList(CASE_SIZE, TYPE_SIZE):
  err = []
  for i in range(CASE_SIZE):
    err.append([])
    for j in range(TYPE_SIZE):
      err[i].append([])

  return err
```

Listing 5: `[tests.py]`: tests for the solver:

```
from dolfin import *

quad_degree = 4

def getVelocity(test, quad_degree):
  if test == 'linear'     : return Expression( ('1.2','1.4','0.7'), degree=quad_degree )
  if test == 'polynomial' : return Expression( ('0.66*(1 - pow(x[1], 2) )', '0.2 + x[2]*
      x[0]', '0.8 - pow(x[0], 2)'), degree=quad_degree )
  if test == 'non-linear' : return Expression( ('0.66*(1 - pow(x[1], 2) )', '0.2 + sin(
      pi*x[0])', '0.8 - pow(x[0], 2)' ), degree=quad_degree )
  if test == 'zero-bc'    : return Expression( ('1.2','1.4','0.7'), degree=quad_degree )

def getZeroOrderTerm(test):
  if test == 'linear'     : return 1.0
  if test == 'polynomial' : return 10.0
  if test == 'non-linear' : return 10.0
  if test == 'zero-bc'    : return 1.0

def getSolution(test, quad_degree):
  if test == 'linear'     : return Expression( ('2*x[0] - 1.5*x[1] + 0.6*x[2]', '1.2*x
      [0] + 2.4*x[1] - 0.6*x[2]', '1.4*x[0] + 0.3*x[1] - 2.5*x[2]'), degree=quad_degree
      )
  if test == 'polynomial' : return Expression( ('x[0]*x[1]', '1 - pow(x[1], 2)', '1 + x
      [2]*x[0]'), degree=quad_degree )
  if test == 'non-linear' : return Expression( ('sin(pi*x[1])', '1 - pow(x[1],2)', '1 +
      pow(x[2],2)'), degree=quad_degree )
  if test == 'zero-bc'    :
    sol = 'sin(pi*x[0])*sin(2*pi*x[1])*sin(3*pi*x[2])'
    return Expression( (sol, sol, sol), degree=quad_degree )

def getRHS_Mathematica(test, quad_degree):

  if test == 'linear'    : return Expression( ('0.72', '4.38', '0.35'), degree=
      quad_degree )

  if test == 'polynomial':
    rhs1 = '0.2*x[0] + 0.66*x[1]*(1 - pow(x[1],2)) - 0.8*x[2] + 2*pow(x[0],2)*x[2] +
        (0.8 - pow(x[0],2))*x[2] + (1 - pow(x[1],2))*x[2] - 2*x[0]*(1 + x[0]*x[2])'
    rhs2 = '-0.66*x[0] - 0.66*x[0]*pow(x[1],2) + 0.66*x[0]*(1 - pow(x[1],2)) - 2*x
        [1]*(0.2 + x[0]*x[2])'
    rhs3 = 'x[0]*(0.8 - pow(x[0],2)) + x[0]*(1 - pow(x[1],2)) + 0.66*x[2] - 0.66*pow(x
        [1],2)*x[2]'
    return Expression( (rhs1 ,rhs2, rhs3), degree=quad_degree )
```

```
  if test == 'non-linear':
    rhs1 = '-2*x[0]*(1 + pow(x[2],2)) + 0.6283185307179586*cos(pi*x[1]) + pi*cos(pi*x
         [1])*sin(pi*x[0])'
    rhs2 = '-2.0734511513692637*cos(pi*x[1]) + 2.0734511513692637*pow(x[1],2)*cos(pi*x
         [1]) + 2.0734511513692637*(1 - pow(x[1],2))*cos(pi*x[1]) - 1.32*x[1]*sin(pi*x
         [1]) - 2*x[1]*(0.2 + sin(\[pi]x[0]))'
    rhs3 = '2*(0.8 - pow(x[0],2))*x[2]'
    return Expression( (rhs1 ,rhs2, rhs3), degree=quad_degree )

  if test == 'zero-bc'   :
    rhs1 = '6.597344572538565*cos(3*pi*x[2])*sin(pi*x[0])*sin(2*pi*x[1]) +
         8.79645943005142*cos(2*pi*x[1])*sin(pi*x[0])*sin(3*pi*x[2]) +
         3.7699111843077517*cos(pi*x[0])*sin(2*pi*x[1])*sin(3*pi*x[2])'
    rhs2 = '6.597344572538565*cos(3*pi*x[2])*sin(pi*x[0])*sin(2*pi*x[1]) +
         8.79645943005142*cos(2*pi*x[1])*sin(pi*x[0])*sin(3*pi*x[2]) +
         3.7699111843077517*cos(pi*x[0])*sin(2*pi*x[1])*sin(3*pi*x[2])'
    rhs3 = '6.597344572538564*cos(3*pi*x[2])*sin(pi*x[0])*sin(2*pi*x[1]) +
         8.79645943005142*cos(2*pi*x[1])*sin(pi*x[0])*sin(3*pi*x[2]) +
         3.7699111843077517*cos(pi*x[0])*sin(2*pi*x[1])*sin(3*pi*x[2])'
    return Expression( (rhs1 ,rhs2, rhs3), degree=quad_degree )

def getRHS_Dolfin(A_0, A_, v):
  return dot(cross(curl(A_0),v),A_) + graddot(A_,A_0,v)

def graddot(A_, A_0, v):
  return (A_[0]*v[0].dx(0)*A_0[0] + A_[0]*v[1].dx(0)*A_0[1] + A_[0]*v[2].dx(0)*A_0[2]+ \
          A_[1]*v[0].dx(1)*A_0[0] + A_[1]*v[1].dx(1)*A_0[1] + A_[1]*v[2].dx(1)*A_0[2]+ \
          A_[2]*v[0].dx(2)*A_0[0] + A_[2]*v[1].dx(2)*A_0[1] + A_[2]*v[2].dx(2)*A_0[2]+ \
          A_[0]*A_0[0].dx(0)*v[0] + A_[0]*A_0[1].dx(0)*v[1] + A_[0]*A_0[2].dx(0)*v[2]+ \
          A_[1]*A_0[0].dx(1)*v[0] + A_[1]*A_0[1].dx(1)*v[1] + A_[1]*A_0[2].dx(1)*v[2]+ \
          A_[2]*A_0[0].dx(2)*v[0] + A_[2]*A_0[1].dx(2)*v[1] + A_[2]*A_0[2].dx(2)*v[2])
```

Listing 6: `[errors.py]`: error computation and plotting:

```
# ===| ROUTINES FOR ERROR COMPUTATION, PRINTING AND PLOTTING |===

from config import FE_TYPE

# initializing error types
if FE_TYPE == 'DG':
  L2, CURL, JUMP, JUMP_EXT, TYPE_SIZE = range(5)
  TYPE_NAMES = ['L2 norm', 'H(curl) norm', 'Jump interior', 'Jump exterior']
else: # FE_TYPE == 'CG':
  L2, CURL, TYPE_SIZE = range(3)
  TYPE_NAMES = ['L2 norm', 'H(curl) norm']

def computeErrors(errors, case, A, A_0, mesh, quad_degree, FE_TYPE):

  from dolfin import VectorFunctionSpace, interpolate, Function, sqrt, assemble, dot,
      curl, jump, dx, dS, ds

  # interpolation in higher degree space for better error approximation
  VE   = VectorFunctionSpace(mesh, FE_TYPE, quad_degree)
  A_0_VE  = interpolate(A_0, VE)
  A_VE    = interpolate(A,   VE)

  # computing error field
  E = Function(VE)
  E.vector()[:] = A_VE.vector()[:] - A_0_VE.vector()[:]

  # computing error field norms

  # L2:
  try:
    errors[case][L2].append( sqrt(assemble( dot(E,E)*dx, mesh = mesh )) )
  except: print 'Skipping L2 error...'

  # CURL:
  try:
    errors[case][CURL].append( sqrt(assemble( dot(curl(E),curl(E))*dx, mesh = mesh )) )
  except: print 'Skipping CURL error...'

  # JUMP:
  try:
    errors[case][JUMP].append( sqrt(assemble( dot(jump(E),jump(E))*dS, mesh = mesh )) )
  except: print 'Skipping JUMP error...'

  # JUMP_EXT:
  try:
    errors[case][JUMP_EXT].append( sqrt(assemble( dot(E,E)*ds, mesh = mesh )) )
  except: print 'Skipping JUMP_EXT error...'
```

```
def plotErrors(refinements, err, compute_all, skip, slopes, title_text, show_fig, save_fig, PYLAB_PATH, SUFFIX, FIGURE_FORMAT):

  from pylab import figure, gcf, gca, subplot, loglog, grid, legend, xlabel, ylabel, title, show

  if compute_all:

    if len(err) == 1:
      cols = 1
    else:
      cols = 2

    rows = (len(err) - 1) / cols + 1

    figure(figsize=(cols*8, rows*4.62))

    for p in range(len(err)):

      subplot(rows, cols, p + 1)

      for err_type in range(TYPE_SIZE):
        loglog(refinements[skip:], err[p][err_type][skip:], '-o')

      for slope in slopes:
        plotSlope(slope, gca())

      grid()
      legend( TYPE_NAMES, loc='lower left')

      ylabel('error norm')
      if rows == 1: xlabel('1/h')
      title(title_text + ' for p = ' + str(p))

  else:

    figure(figsize=(8, 4.62))

    p = len(err) - 1
    for err_type in range(TYPE_SIZE):
      loglog(refinements[skip:], err[p][err_type][skip:], '-o')

    for slope in slopes:
      plotSlope(slope, gca())

    grid()
    legend(TYPE_NAMES, loc='lower left')

    ylabel('error norm')
    xlabel('1/h')
    title(title_text + ' for p = ' + str(p))

  if save_fig: savefig(gcf(), PYLAB_PATH + 'ERRORS' + SUFFIX + '.' + FIGURE_FORMAT, FIGURE_FORMAT)
  if show_fig: show()

# returns the list of convergence rates
def computeConvergenceRates(rates, errors, p, ref, wrt):

  from math import log as ln # (log is a dolfin name too)

  for err_type in range(TYPE_SIZE):
    for i in range(1, len(errors[p][err_type])):
      try:
        if wrt == 'hmax':
          rates[p][err_type].append( ln(float(errors[p][err_type][i-1])/float(errors[p][err_type][i]))/ln(float(ref[p][i-1])/float(ref[p][i])) )
        elif wrt == 'ndofs':
          rates[p][err_type].append( ln(float(errors[p][err_type][i-1])/float(errors[p][err_type][i]))/ln(float(ref[p][i])/float(ref[p][i-1])) )
        else:
          print 'wrt must be either \'meshsize\' or \'ndofs\''
          return
      except:
        rates[p][err_type].append( 0.0 )

def plotConvergenceRates(rates, method, skip, title_text, show_fig, save_fig, PYLAB_PATH, SUFFIX, FIGURE_FORMAT):

  from pylab import axhline, gcf, plot, grid, legend, xlabel, ylabel, title, show

  # line splitting converge and divergence regions
```

```
  axhline(y=0, linewidth=10, color='r', alpha=0.4, label='_nolegend_')

  for err_type in range(TYPE_SIZE):
    avg_rate = []
    for case in range(len(rates)):
      if method == 'average':
  if len(rates[case][err_type]) > skip:
          avg_rate.append( sum(rates[case][err_type][skip:]) / ( len(rates[case][
              err_type]) - skip ) )
        else:
    avg_rate.append( sum(rates[case][err_type]) / len(rates[case][err_type]) )
      elif method == 'last':
        avg_rate.append( rates[case][err_type][-1] )
      else:
        print 'method must be either \'average\' or \'last\''

    plot(range(len(rates)), avg_rate, '--o', label=TYPE_NAMES[err_type])

  grid()
  legend(loc='lower right')
  xlabel('polynomial degree')
  ylabel('convergence rate')
  title(title_text)

  if save_fig: savefig(gcf(), PYLAB_PATH + 'RATES' + SUFFIX + '.' + FIGURE_FORMAT,
      FIGURE_FORMAT)
  if show_fig: show()

# text output to stdout and fout
def output(title, data, compute_all, cpu_time, OUTPUT_PATH, SUFFIX, format_string):

  from datetime import timedelta

  # makes array into a compactly formatted string
  def format (array, format_string):
    line = ''
    for idx in range(len(array)):
      line += format_string % array[idx] + '\t'
    return line.rstrip()

  # writes @param line into stdout & fout
  def record (fout, line):
    print line
    fout.write(line + '\n')

  fout = open(OUTPUT_PATH + title + SUFFIX, 'w')

  for p in range(len(data)):

    # compute only for the value p = P
    if not compute_all: p = len(data) - 1

    record(fout, '\n===| ' + title + ' for p = ' + str(p) + ' |=== [CPU time: ' + str(
        timedelta(seconds=round(cpu_time[p]))) + ']\n')
    for err_type in range(TYPE_SIZE):
      record(fout, str(TYPE_NAMES[err_type]) + ':')
      record(fout, format(data[p][err_type], format_string))
      record(fout, '') # adds an additional line between err_type's

    # compute only for the value p = P
    if not compute_all: break

  fout.close()

# writes pylab figure to file
def savefig (fig, path, FIGURE_FORMAT):
  fig.savefig(path, dpi=None, facecolor='w', edgecolor='w', orientation='portrait',
      papertype=None, format=FIGURE_FORMAT, transparent=False)

# plots slopes
def plotSlope(slope, plt):

  from pylab import xlim, ylim, loglog
  from math import pow, log as ln # (log is a dolfin name too)

  xmin, xmax = xlim()
  ymin, ymax = ylim()

  xmin = xmax / 10 * 1.2
  xmax = 6 * xmin

  ymax = ymax * 0.7
  ymin = pow( 10, ln(ymax, 10) - slope * (ln(xmax, 10) - ln(xmin, 10)) )
```

```
  plt.plot([xmin, xmax], [ymax, ymin], '--k', label='_nolegend_')
  plt.annotate(str(slope), xy=(xmax, ymin),  xycoords='data')
```

Listing 7: [debug.py]: debugging routines:

```
# === DEBUGGING ROUTINES ===

from dolfin import *
from auxiliary import *

def plotInflowSurface(n, v, mesh):
  # inflow surface
  class Surface_Inflow(SubDomain):
    def __init__(self, normal, velocity):
      self.n = normal
      self.v = velocity
      SubDomain.__init__(self)

    def inside(self, x, on_boundary):
      return dot3D(normal3D(x), self.v(x)) < 0

  # plotting inflow surface
  surface_mesh = BoundaryMesh(mesh)
  surface_parts = MeshFunction('uint', surface_mesh, surface_mesh.topology().dim())
  surface_parts.set_all(1)
  inflow = Surface_Inflow(n, v)
  inflow.mark(surface_parts, 0)
  plot(surface_parts, title='Inflow surface (in red)', axes=True)
```

Listing 8: [dumpload.py]: file I/O for objects:

```
# dumps obj to file
def dump(obj, path):
  import cPickle
  cPickle.dump(obj, open(path, 'wb'))

# loads obj from file
def load(path):
  import cPickle
  return cPickle.load(open(path, 'rb'))
```

Listing 9: [loadplot.py]: routines for dumped files:

```
# LOADS FROM FILES AND PLOTS DUMPED ERRORS AND CONVERGENCE RATES
#( Dolfin isntallation is NOT needed, MatPlotLib is sufficient )

from config import *
from auxiliary import *
from dumpload import *
from errors import *

import sys

# === QUICK CONFIG [ overrides config.py ] ===

errors_skip   = 0   # number of first error norms to ignore when plotting
rates_skip    = 2   # number of first error norms to ignore when computing average
    convergence rates

# ===

try:
  test     = sys.argv[1]  # test to use
  P        = int(sys.argv[2]) # polynomial degree
  levels   = int(sys.argv[3]) # number of refinements
except:
  test     = 'polynomial'
  P        = 1
  levels   = 7

try:
  show_fig = sys.argv[4]
  save_fig = sys.argv[5]
except:
  show_fig = True
  save_fig = False
```

```
if test == 'zero-bc':
  errors_skip = 1
  rates_skip  = 2

SUFFIX = SUFFIX_FORMAT % (test, P, levels)

errors = load (DUMP_PATH + 'ERRORS' + SUFFIX)
rates  = load (DUMP_PATH + 'RATES'  + SUFFIX)

plotErrors(getRefinements(level_offset, levels), errors, compute_all, errors_skip,
    errors_slopes, 'Error norms', show_fig, save_fig, PYLAB_PATH, SUFFIX, FIGURE_FORMAT)
if compute_all: plotConvergenceRates(rates, 'average', rates_skip, '', show_fig,
    save_fig, PYLAB_PATH, SUFFIX, FIGURE_FORMAT)
```

B Shell scripts

Listing 10: [batch_job.sh]: executes multiple configurations:

```
#/bin/bash
# executes computations for multiple configurations

# P          : 0 1 2 3 4
# max LEVELS : 8 7 6 5 4
# cpu_time   : 2 2 2 2 2 // in hours per one test

ENV='cluster'

DEGREES='0 1 2 3'

TESTS='polynomial non-linear zero-bc'

MAX='8'

for P in $DEGREES
do
  LEVELS=$(($MAX-$P))
  for TEST in $TESTS
  do
    python main.py $TEST $P $LEVELS $ENV;
  done
done
```

Listing 11: [save_figs.sh]: saves multiple pylab figures:

```
#/bin/bashcd '/media/data/Dropbox/ETH Zurich/Lectures/Term Paper/fenics/sukysj_dgdmc'
# saves pylab figures for multiple configurations

DEGREES='0 1 2 3'

TESTS='polynomial non-linear zero-bc'

MAX='8'

SHOW_FIG='False'
SAVE_FIG='True'

for P in $DEGREES
do
  LEVELS=$(($MAX-$P))
  for TEST in $TESTS
  do
    python loadplot.py $TEST $P $LEVELS $SHOW_FIG $SAVE_FIG;
  done
done
```

C Mathematica code

Listing 12: [rhs.nb]: computes f on Ω:

```
<< VectorAnalysis`
```

```
SetCoordinates[Cartesian[x, y, z]];

Linear test :

a1 = 2 x - 1.5 y + 0.6 z;
a2 = 1.2 x + 2.4 y - 0.6 z;
a3 = 1.4 x + 0.3 y - 2.5 z;
A = {a1, a2, a3};

v1 = 1.2;
v2 = 1.4;
v3 = 0.7;
v = {v1, v2, v3};

CForm[Grad[A.v] + Cross[Curl[A], v]]

Polynomial test :

a1 = x y;
a2 = (1 - y^2) ;
a3 = (1 + z x);
A = {a1, a2, a3};

v1 = 0.66*(1 - y^2);
v2 = 0.2 + z x;
v3 = 0.8 - x^2;
v = {v1, v2, v3};

CForm[Grad[A.v] + Cross[Curl[A], v]]

Non - linear test :

a1 = Sin[\[Pi] y];
a2 = (1 - y^2) ;
a3 = (1 + z^2);
A = {a1, a2, a3};

v1 = 0.66*(1 - y^2);
v2 = 0.2 + Sin[\[Pi]x];
v3 = 0.8 - x^2;
v = {v1, v2, v3};

CForm[Grad[A.v] + Cross[Curl[A], v]]

Zero - bc test :

a1 = Sin[\[Pi] x]*Sin[2 \[Pi] y]*Sin[3 \[Pi] z];
a2 = Sin[\[Pi] x]*Sin[2 \[Pi] y]*Sin[3 \[Pi] z];
a3 = Sin[\[Pi] x]*Sin[2 \[Pi] y]*Sin[3 \[Pi] z];
A = {a1, a2, a3};

v1 = 1.2;
v2 = 1.4;
v3 = 0.7;
v = {v1, v2, v3};

CForm[Grad[A.v] + Cross[Curl[A], v]]
```

D Computations architecture

Arch x86-64

Kernel 2.6.30.9-90

CPU 16 x Quad-Core 2.3Ghz

Memory 64 GB

OpenMP not enabled

MPI not enabled

Usage per test for `levels` $= 8 -$ `degree`:

CPU: ~ 2 hours

Memory: ~ 40 GB

Location: cmath-7 node in SAM, D-MATH, ETH Zurich