# Implementation and evaluation of Discontinuous Galerkin methods using Galerkin4GridTools

Author: Niccolò Discacciati*

Mentors: William B. Sawyer, Luca Bonaventura, Christopher Bignamini

February 16th, 2019

## 1 Introduction

Over the last decades, the interest in hyperbolic partial differential equations (PDEs) and their numerical solutions has grown dramatically [23]. Possible applications include molecular dynamics, simulation of atmospheric phenomena or magnetohydrodynamics. It is well known that hyperbolic problems constitute a challenging class of PDEs, mainly due to the need to preserve physical quantities and the emergence of discontinuities even for smooth initial conditions [19]. In the development of the associated numerical algorithms, multiple aspects have to be considered [15, 14]. Firstly, schemes able to handle complex geometries are preferred in practical application. This flexibility allows an increasing accuracy in the simulations, while keeping the computational cost controlled. Secondly, high-order methods guarantee good approximation properties even using relatively coarse meshes. If long temporal intervals and multi-dimensional problems are considered, or a strict limit on the required accuracy is imposed, discretizations with high-order schemes become necessary. Thirdly, physical properties of the continuous equation have to be preserved at a discrete level. They include anisotropy, i.e., the presence of preferential spatial directions, and conservation of physical quantities, as mass, momentum and energy. Finally, algorithms exploiting spatial locality guarantee a good parallelization potential in a distributed memory context. In multi-dimensional contexts, a large number of degrees of freedom is generally present. Hence, an efficiently parallelized code is able to reduce the computational time, without losing the accuracy of the numerical solution. In some cases, like weather forecasting, parallelization is mandatory in order to get concrete results in a reasonable time.
The classical schemes, like Finite Differences (FD), Finite Volumes (FV) or Finite Elements (FE) fail in at least one of the above-mentioned aspects. On the other hand, Discontinuous Galerkin (DG) solvers can be interpreted as a class of methods gathering the good properties of the above-mentioned schemes. The high-order accuracy is guaranteed by high-degree element-wise polynomials, while conservation is numerically enforced at the element boundaries.
In this project, we develop a C++ version of a DG solver for conservation laws. Even though the code allows the selection of a generic flux function, we primarily focus on the linear advection equation and the Shallow Water Equations (SWEs). The support for a Cartesian geometry is complete, and several tests have been run to validate the code. Switching to spherical coordinates creates multiple additional complexities, from both a theoretical and a computational point of view. For this reason, only the linear advection equation is considered in the code implementation. The code is completed by adding the support for a variable discretization degree in the mesh,

*Email address: niccolo.discacciati@epfl.ch

usually called *adaptivity*. Only *static adaptivity* has been validated, with the distribution of the polynomial degrees not changing in time. A *dynamic adaptivity*, characterized by a time-varying degree, goes beyond the goal of this project and it is only partially supported in the code. The main advantages of variable degrees fall in the reduced computational cost, while keeping a comparable accuracy. In a spherical geometry, adaptivity improves the stability bounds given by the Courant–Friedrichs–Lewy (CFL) condition.
The code is mainly built using:

- GridTools (GT) [13], an efficient C++ library developed at CSCS. It makes extensive usage of template meta-programming and it has optimizations for both Central Processing Unit (CPU) and Graphics Processing Unit (GPU) architectures.
- Intrepid [4], a C++ Trilinos library. It provides the numerical support for finite element discretizations. Epetra [5], another Trilinos library, is partly employed.

The Galerkin4GridTools (G4GT) framework provides the link between these libraries, adding a higher-level, user-friendly layer to GridTools. Since GT is mostly designed for FD schemes, G4GT has the further advantage to introduce the support for FE discretization using GT-based codes.
The rest of this report is structured as follows. In Section 2 we provide the details on the mathematical formulation of conservation laws and their numerical discretization. Then, in Section 3 we describe the coding strategy, highlighting its main features. In Sections 4 and 5 we show the numerical results, commenting on the code performances on CPUs. A few concluding remarks are made in Section 6.

# 2 Numerical discretization

## 2.1 The problem

Let $\Omega \subset \mathbb{R}^D$ $(D = 2)$ be a bounded domain described by the spatial coordinate $\boldsymbol{x} = (x, y)$ and $T > 0$ be a fixed time instant. In view of the code implementation, we restrict to rectangular domains, denoted by $\Omega = [a, b] \times [c, d]$. Consider a generic conservation law, expressed as

$$\frac{\partial \boldsymbol{u}}{\partial t} + \nabla \cdot \boldsymbol{f} = \boldsymbol{s}, \quad \forall (\boldsymbol{x}, t) \in \Omega \times [0, T], \tag{1}$$

where $\boldsymbol{u} : \Omega \times [0, T] \to \mathbb{R}^n$ is a vector of $n$ conserved variables, $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^{n \times D}$ is the flux function and $\boldsymbol{s} : \Omega \times [0, T] \to \mathbb{R}^n$ is a source term. We emphasize the fact that in (1) the convective flux $\boldsymbol{f}$ depends on $\boldsymbol{u}$, $\boldsymbol{x}$ and $t$. However, we often write $\boldsymbol{f} = \boldsymbol{f}(\boldsymbol{u})$, omitting the dependence of $\boldsymbol{f}$ on space and time. A similar reasoning holds for the source term $\boldsymbol{s}$. Problem (1) is completed with suitable boundary and initial conditions. We restrict our attention on *doubly periodic* boundary conditions, defined as

$$\boldsymbol{u}(x = b, y, t) = \boldsymbol{u}(x = a, y, t), \quad \boldsymbol{u}(x, y = d, t) = \boldsymbol{u}(x, y = c, t), \quad \forall t \in [0, T]$$

with a periodic initial condition, denoted by $\boldsymbol{u}_0 = \boldsymbol{u}_0(\boldsymbol{x})$.
We first restrict our attention to a Cartesian geometry, serving as a simple framework to describe the spatial discretization. The extension to the spherical geometry is provided in Section 2.3.

## 2.2 Spatial discretization: Cartesian geometry

As mentioned in Section 1, the spatial discretization is performed using a Discontinuous Galerkin scheme [15, 9]. We briefly describe it in the scalar case $n = 1$. We start by splitting the rectangular domain into $K$ elements, denoted by $D^k$, for any $k = 1 \dots K$. We restrict our attention on conforming structured meshes composed of rectangular elements

$$D^k = [x_l^k, x_r^k] \times [y_b^k, y_t^k] \quad k = 1, \dots, K.$$

In each element, let $V_h^k$ be the finite-dimensional space of multivariate polynomials up to a given degree $r = r(k)$ in each spatial dimension

$$V_h^k = \left\{ v : v = \sum_{i,j=0}^{r} \alpha_{ij} x^i y^j, \quad x \in [x_l^k, x_r^k], y \in [y_b^k, y_t^k] \right\}.$$

Consequently, the finite-dimensional space in which we seek the solution is the space of broken polynomials defined as

$$V_h = \left\{ v \in L^2(\Omega) : v|_{D^k} \in V_h^k \right\}$$

In the spirit of a standard Discontinuous Galerkin discretization, the numerical solution can be viewed as the direct sum of local approximations

$$u_h = \bigoplus_{k=1}^{K} u_h^k, \tag{2}$$

where $u_h^k \in V_h^k$. Due to (2), we can restrict our attention on a single mesh element, dropping the superscript $k$ if necessary. Defining the local residual as

$$\mathcal{R}_h^k = \frac{\partial u_h^k}{\partial t} + \nabla \cdot \boldsymbol{f}(u_h^k) - s(u_h^k),$$

we impose it to vanish locally in a Galerkin sense, namely:

$$\int_{D^k} \mathcal{R}_h^k \phi_h^k = 0$$

for any suitably defined test function $\phi_h^k \in V_h^k$. After an integration by parts, the weak DG formulation is

$$\int_{D^k} \frac{\partial u_h^k}{\partial t} \phi_h^k + \int_{\partial D^k} \boldsymbol{f}^*(u_h) \cdot \boldsymbol{n}_k \; \phi_h^k - \int_{D^k} \boldsymbol{f}(u_h^k) \cdot \nabla \phi_h^k = \int_{D^k} s(u_h^k) \phi_h^k. \tag{3}$$

In (3), the physical flux at the element boundary is replaced by a numerical approximation, denoted by $\boldsymbol{f}^*$ [15]. This guarantees that the flux is single-valued at each edge, enforcing flux conservation across any edge. Note that all terms in (3) are local to the $k$-th element, except the numerical flux, which depends on the neighboring elements. This is the reason why we write $\boldsymbol{f}^*(u_h)$ instead of $\boldsymbol{f}^*(u_h^k)$. The choice of $\boldsymbol{f}^*$ plays a key role in terms of consistency, accuracy and stability of the numerical solver. Consider a generic edge $e$ and let $k, \tilde{k}$ the indexes of the elements sharing $e$, and assume that $\tilde{k}$ is on the right of $k$. A popular choice [15] consists of setting

$$\boldsymbol{f}^*(u_h) = \boldsymbol{f}^*(u_h^k, u_h^{\tilde{k}}) = \frac{\boldsymbol{f}(u_h^k) + \boldsymbol{f}(u_h^{\tilde{k}})}{2} - \frac{\alpha}{2}(u_h^{\tilde{k}} - u_h^k)\boldsymbol{n}_k, \tag{4}$$

where $\alpha \geq 0$ is a large enough stabilization parameter and $\boldsymbol{n}_k$ is the normal unit vector, pointing outwards $D^k$. Note that the right-hand-side of (4) consists of a centered flux plus a penalty

term on the jump of the solution, and guarantees both high-order accuracy and stability of the scheme. The parameter $\alpha$ is generally defined as [8]

$$\alpha = \max |\boldsymbol{f}'(u) \cdot \boldsymbol{n}|, \tag{5}$$

with the inner product acting between the two spatial components of $\boldsymbol{f}$ and the normal vector. The right-hand-side of (5) can be physically interpreted as the maximum wave speed. The maximum is either taken globally over all the elements (*Lax-Friedrichs* flux) or locally across the $e$-th edge (*Rusanov* flux). Both options are available in the code, but due to performance reasons we present the results using the former. Even though this leads to higher dissipation, the qualitative behavior of the resulting solution is not altered.
Recalling (3), we need to choose a suitable basis function for $V_h^k$, denoted as

$$\{\phi_j\}_{j=1}^{(r+1)^D}. \tag{6}$$

Therefore, the following local expansion for the solution holds:

$$u_h^k = \sum_{j=1}^{(r+1)^D} u_j^k \phi_j. \tag{7}$$

An important aspect, common to most of FE schemes, is the definition of the *reference element*, denoted by $I$, with spatial coordinates $\boldsymbol{s} = (s, t)$. In this work, we select

$$I = [-1, 1] \times [-1, 1].$$

A significant part of the complexity is shifted to $I$, making most of the computations easier and faster. After assembling most of the FE tools in $I$, a function

$$\Psi = \Psi(\boldsymbol{r}) : I \to D^k$$

maps back to the physical element. For instance, once the basis is defined on $I$, we simply set

$$\phi_j(\boldsymbol{x}) = \phi_j(\Psi^{-1}(\boldsymbol{x}))$$

with a little abuse of notation.
Two alternative choices for the functions $\phi_j$ are common in the literature. The first one relies on a *nodal* basis, based on the identification of a set of distinct points [11]. Despite its simplicity, in this work we adopt the second strategy, which aims to construct a *modal* basis. In particular, we consider the multivariate Legendre polynomials. One possible definition, in the one-dimensional case, is the following recurrence relation

$$(n+1)P_{n+1}(s) = (2n+1)sP_n(s) - nP_{n-1}(s), \quad n \geq 1,$$

which is started by setting

$$P_0(s) = 1, \quad P_1(s) = s.$$

Extension to the two-dimensional framework relies on a tensor product argument, resulting in

$$\phi_j(s, t) = P_l(s)P_m(t),$$

where the index $j \geq 1$ is linked to $(l, m)$ as

$$j = l(r+1) + m + 1.$$

The choice of a modal basis is rather popular in (explicit) DG schemes and presents several advantages. The orthogonality condition

$$\int_I P_j(s,t)P_{j'}(s,t) = \frac{2}{2m+1}\frac{2}{2l+1}\delta_{jj'}, \quad j \leftrightarrow (l,m),\ j' \leftrightarrow (l',m')$$

is satisfied, making the mass matrix diagonal. Consequently, the energy of the solution depends only on the squares of the coefficients $u_j$ of (7), up to normalization factors. Criteria for static and dynamic adaptivity are also easy to implement, due to the *hierarchical* structure of the basis. Increasing the discretization degree results in adding more basis functions, without altering the previous ones. Hence,a variation in the degree translates to a different number of coefficient and basis functions.
Due to (6), (3) is equivalent to

$$\int_{D^k} \frac{\partial u_h^k}{\partial t}\phi_i + \int_{\partial D^k} \boldsymbol{f}^*(u_h)\cdot\boldsymbol{n}\ \phi_i - \int_{D^k} \boldsymbol{f}(u_h^k)\cdot\nabla\phi_i = \int_{D^k} s(u_h^k)\phi_i, \quad i = 1,\dots,(r+1)^D,$$

which can be cast as

$$\left[\boldsymbol{M}^k\frac{d\boldsymbol{u}^k}{dt}\right]_i + \int_{\partial D^k} \boldsymbol{f}^*(\boldsymbol{u})\cdot\boldsymbol{n}\ \phi_i - \int_{D^k} \boldsymbol{f}(\boldsymbol{u}^k)\cdot\nabla\phi_i = \int_{D^k} s(\boldsymbol{u}^k)\phi_i, \quad i = 1,\dots,(r+1)^D,$$

where $\boldsymbol{M}^k$ is the local mass matrix

$$\boldsymbol{M}_{ij}^k = \int_{D^k} \phi_j\phi_i$$

and

$$\boldsymbol{u}^k = [u_1^k \dots u_{(r+1)^D}^k]^T$$

is the vector which gathers the modal coefficients of the (local) solution. They are also named *(local) degrees of freedom*, since they are both necessary and sufficient to uniquely determine $u_h^k$.
Gathering all the element-wise contributions, we can formally write the problem as

$$\boldsymbol{M}\frac{d\boldsymbol{u}}{dt} = \boldsymbol{A}(\boldsymbol{u}), \tag{8}$$

where $\boldsymbol{M}$ is the global mass matrix, $\boldsymbol{A}$ is a finite-dimensional operator and $\boldsymbol{u}$ is the unknown vector of modal coefficients of all the elements. There is no need to specify the structure of the operator $\boldsymbol{A} = \boldsymbol{A}(\boldsymbol{u})$, since it is never used for a practical implementation. For linear problems, it can be represented through a (sparse) matrix, having a pattern similar to the one presented in [15] for a second order Poisson problem.
Extending this method to systems of conservation laws is straightforward. Essentially, the previous reasoning can be applied treating each equation separately. Indeed, (1) can be split in its components as

$$\frac{\partial u_i}{\partial t} + \nabla\cdot\boldsymbol{f}_i = s_i, \quad i = 1,\dots,n$$

We stress the fact that this argument holds only in a Cartesian geometry, since

$$\nabla\cdot\boldsymbol{g} = \begin{pmatrix}\nabla\cdot\boldsymbol{g}_1\\ \vdots\\ \nabla\cdot\boldsymbol{g}_n\end{pmatrix}, \quad \forall\boldsymbol{g} = \begin{pmatrix}\boldsymbol{g}_1\\ \vdots\\ \boldsymbol{g}_n\end{pmatrix} \in \mathbb{R}^{n\times D} \tag{9}$$

With a little abuse of notation, we denote with $\boldsymbol{f}'$ the Jacobian matrix of the convective flux with respect to $\boldsymbol{u}$. Similarly, the maximum local wave speed is given by its maximum absolute eigenvalue, taking into account both the spatial components.

### 2.3 Spatial discretization: spherical geometry

Aiming to simulate atmospheric dynamics on the Earth, the spherical geometry is described by the longitude-latitude coordinates $(\lambda, \theta)$, fixing the radius $R$. Consequently, we have $\Omega = [0, 2\pi] \times [-\frac{\pi}{2}, \frac{\pi}{2}]$. As in Section 2.2, we begin our description with a scalar conservation law. Even though most of the reasoning carried out for a Cartesian geometry is not altered, we emphasize the main changes. Firstly, we recall that the gradient and divergence operators are defined as

$$\nabla g = \frac{1}{R\cos\theta}\frac{\partial g}{\partial \lambda}\mathbf{e}_\lambda + \frac{1}{R}\frac{\partial g}{\partial \theta}\mathbf{e}_\theta, \quad \forall g \in \mathbb{R} \tag{10a}$$

$$\nabla \cdot \boldsymbol{g} = \frac{1}{R\cos\theta}\left(\frac{\partial g_\lambda}{\partial \lambda} + \frac{\partial (g_\theta \cos\theta)}{\partial \theta}\right), \quad \forall \boldsymbol{g} = (g_\lambda, g_\theta) \in \mathbb{R}^2 \tag{10b}$$

When integrating by parts, we recall that the infinitesimal surface element on Cartesian coordinates $dA = dxdy$ is transformed into $dA = R^2\cos\theta d\lambda d\theta$. Using the modal basis, we get

$$\begin{aligned}\int_{D^k} \frac{\partial u_h^k}{\partial t}\phi_i R^2\cos\theta + \int_{\partial D^k} (f_\lambda^*(u_h)\cdot n_\lambda + f_\theta^*(u_h)\cdot\cos\theta n_\theta)\ \phi_i R - \int_{D^k} \boldsymbol{f}(u_h^k)\cdot\nabla\phi_i R^2\cos\theta \\ = \int_{D^k} s(u_h^k)\phi_i R^2\cos\theta,\end{aligned} \tag{11}$$

Note that due to the multiplication with $\cos\theta$, the singularities in (10a) and (10b) cancel out. The mass matrix is now defined as

$$\boldsymbol{M}_{ij}^k = \int_{D^k} \phi_j\phi_i R^2\cos\theta$$

and loses its diagonal structure. For practical reasons, the weak DG form follows from (11) after canceling $R$ in all the terms.
The extension to systems needs particular care. One can easily verify that (9) is no longer valid. Considering a $2\times 2$ tensor $\boldsymbol{g}$ with components

$$\boldsymbol{g} = \begin{pmatrix}\boldsymbol{g_1}\\ \boldsymbol{g_2}\end{pmatrix} = \begin{pmatrix} g_{1\lambda} & g_{1\theta}\\ g_{2\lambda} & g_{2\theta}\end{pmatrix}$$

and the definition of divergence operator in spherical coordinates [6], we find that

$$\nabla\cdot\boldsymbol{g} = \begin{pmatrix}\nabla\cdot\boldsymbol{g}_1\\ \nabla\cdot\boldsymbol{g}_2\end{pmatrix} + \boldsymbol{q} = \begin{pmatrix}\nabla\cdot\boldsymbol{g}_1\\ \nabla\cdot\boldsymbol{g}_2\end{pmatrix} + \begin{pmatrix}-\frac{g_{1\theta}}{R\cot\theta}\\ +\frac{g_{1\lambda}}{R\cot\theta}\end{pmatrix}, \quad \forall \boldsymbol{g}\in\mathbb{R}^{2\times 2}$$

where $\boldsymbol{q}$ is an extra-term due to the non-constant metric factors of spherical coordinates. More generally, any curvilinear coordinate system produces an additional term due to its metric factors. More details about generic coordinate systems can be found in, e.g., [7] and the references therein. Therefore, in order to split (1) in its components, we need to add the extra term. Hence,

$$\frac{\partial u_i}{\partial t} + \nabla\cdot\boldsymbol{f}_i + q_i = s_i, \quad i = 1,\dots,2$$

is then treated in a way similar to (11). The singularities in the denominator of $\boldsymbol{q}$ cancel out when integrating by parts.

### 2.4 Temporal discretization

The temporal discretization is performed using an explicit scheme. In this way, we avoid implementing solvers for nonlinear systems, which can be a nontrivial task in a parallel framework.

However, in order to guarantee stability, an upper bound on the time step is required.
We can write (8) in the standard form of an ordinary differential equation as

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{M}^{-1}\boldsymbol{A}(\boldsymbol{u}) = \boldsymbol{F}(\boldsymbol{u}), \tag{12}$$

observing that the mass matrix need to be inverted only once, at least when no dynamic adaptivity is present. A popular family of time integrators for (12) is given by the Runge-Kutta schemes [2]. Let $\Delta t$ be a suitably defined time step and let $u^n, u^{n+1}$ be the numerical solutions at the current and next step respectively, i.e. at time $n\Delta t$ and $(n+1)\Delta t$. Then, $u^{n+1}$ is approximated as

$$u^{n+1} = u^n + \Delta t \sum_{i=1}^{s} b_i U_i$$

$$U_i = F(t^n + c_i \Delta t, u^n + \Delta t \sum_{j=1}^{s} a_{ij} U_j) \quad \forall i = 1, \dots, s$$

where $s$ denotes the number of internal substeps. In (13), $a_{ij}$, $b_i$, $c_j$ are suitably defined coefficients which characterize the properties of the scheme, such as consistency and stability. In this work we employ four different schemes:

- RK1 (Explicit Euler). The easiest method computes the time derivative using a first-order explicit approximation. Then, $s = 1$ and the coefficients are simply

$$\begin{aligned} a_{11} &= 0, \\ b_1 &= 1, \\ c_1 &= 0. \end{aligned}$$

- RK2. This second order scheme is composed of two steps ($s = 2$) with corresponding coefficients

$$\begin{aligned} a_{11} = 0,\ a_{21} = 1,\ a_{22} = 0, \\ b_1 = 1/2,\ b_2 = 1/2, \\ c_1 = 0,\ c_2 = 1. \end{aligned}$$

- RK3. Similarly, here we choose $s = 3$ and we set

$$\begin{aligned} a_{11} = 0,\ a_{21} = 1,\ a_{22} = 0,\ a_{31} = 1/4,\ a_{32} = 1/4,\ a_{33} = 0, \\ b_1 = 1/6,\ b_2 = 1/6,\ b_3 = 4/6, \\ c_1 = 0,\ c_2 = 1,\ c_3 = 1/2. \end{aligned}$$

  As expected, this choice results in a third order scheme.

- RK4. Finally, the highest order method we use is the fourth order scheme defined by

$$\begin{aligned} a_{11} = 0,\ a_{21} = 1/2,\ a_{22} = 0,\ a_{31} = 0,\ a_{32} = 1/2,\ a_{33} = 0, \\ a_{41} = 0,\ a_{42} = 0,\ a_{43} = 1, a_{44} = 0, \\ b_1 = 1/6,\ b_2 = 2/6,\ b_3 = 2/6,\ b_4 = 1/6 \\ c_1 = 0,\ c_2 = 1/2,\ c_3 = 1/2,\ c_4 = 1. \end{aligned}$$

We recall that for explicit schemes, the coefficients $a_{ij}$ must vanish for $j \geq i$, so that the values for $a_{ij}$ with $j > i$ are never reported, while $a_{ii}$ are always equal to zero. Note that higher order

methods can be easily implemented, provided that the coefficients of the method are assigned. We recall that that the number of stages $s$ can be chosen equal to the order of the method only up to $s = 4$. If this is not the case, the number of substeps has to be larger than the order of the scheme. A stability condition has to be taken into account, i.e., the time step $\Delta t$ has to be small enough. In the context of hyperbolic conservation laws, the this upper bound is known as CFL condition and depends on the spatial discretization and the convective flux as

$$\Delta t \leq C \frac{h}{\max |f'(u)|}, \tag{14}$$

where $h$ is a suitable mesh spacing, and the denominator denotes a global wave speed. The constant $C$ in (14) is also known as *Courant number* and depends on the maximum discretization degree $r$. Usually, it behaves as

$$C \sim r^{-2},$$

making the stability bound more restrictive in case of high-order discretizations. The CFL condition turns out to be prohibitive in a spherical geometry. Close to the poles, the mesh spacing is reduced, due to large latitudes. This reduces even further the stability condition, limiting the applicability of high polynomial degrees. One possible solution relies on the static adaptivity, with the goal to reduce $r$ as the latitudes increase.

### 2.5 Practical issues

We provide a more detailed description on how to practically compute most of the finite-dimensional operators. For simplicity, in this Section we assume the discretization degree $r$ to be constant in space and time.
Most of the complexity lies in the treatment of nonlinearities. One standard option is to project the flux function on the space of polynomials, and an expansion similar to (7) can be brought up. This is rather typical in the context of nodal Discontinuous Galerkin schemes [11, 26]. On the other hand, if a modal basis or curvilinear elements are employed, all the required terms have to be computed using quadrature formulas [21]. An integration error might be still present, but it can be arbitrarily reduced by increasing the number of quadrature points. Obviously, Gaussian quadrature is employed, since it guarantees the highest degree of exactness for a given number of points [17].
As an example, consider a generic function $g = g(x, y)$, which has to be integrated on a mesh element $D^k$. Then, we have

$$\int_{D^k} g(x,y)dxdy = \int_I \hat{g}(s,t) \; |\det J| \; dsdt \simeq \sum_{q=1}^{Q^D} \hat{g}(s_q,t_q) \; |\det J|(s_q,t_q) \; w_q.$$

The first equality holds since we shift the computation to the reference element, defining $\hat{g}$ as $\hat{g}(s,t) = g(\Psi(s,t))$ and $J$ as the jacobian matrix of the mapping $\Psi$. Then, we approximate the integral over $I$ using $Q^D$ quadrature points denoted by $(s_q, t_q)$, with corresponding weights $w_q$. A similar reasoning holds for the integrals on the element boundaries $\partial D^k$.
In order to choose quadrature points and weights, we focus on the one-dimensional scenario, with the extension to higher spatial dimensions provided by tensor product arguments. Ideally, one would like to include the element boundary in the set of quadrature points. In this way, the internal and boundary integrals can be computed using a single set of points. The former would exploit the whole set, while the latter would involve only the boundary nodes. In the theory of numerical integration, this choice is known as Gauss-Legendre-Lobatto (GLL) quadrature rule. However, due to implementation issues, we consider the Gauss-Legendre (GL) strategy, which does not include the extrema of the interval in the set of quadrature nodes. In order to compute

internal integrals, this choice is similar to the GLL one, using the whole set of quadrature points. The GL rule guarantees a higher degree of exactness with respect to the GLL for a fixed number of quadrature points, since no restriction on the values of the points is required. However, the drawback lies in the computation of the boundary integrals. A different set of points has to be considered, since the boundaries are not part of the GL nodes.
Thus, consider the $Q$ one-dimensional Gauss-Legendre points and weights, computed in the reference interval $[-1, 1]$. We denote them by $p_i$ and $w_i$ respectively. To compute the internal integrals, we simply rely on their tensor product

$$(s_q, t_q) = p_i p_j, \quad w_q = w_i w_j,$$

where $q$ is linked to $(i, j)$ as, e.g., $q = i + (Q+1)j$. On the other hand, for the boundary integrals we consider

$$\begin{aligned}
(s_q, t_q) = (p_i, -1) \quad & w_q = w_i, \quad \text{for the bottom boundary,} \\
(s_q, t_q) = (1, p_i) \quad & w_q = w_i, \quad \text{for the right boundary,} \\
(s_q, t_q) = (p_i, 1) \quad & w_q = w_i, \quad \text{for the top boundary,} \\
(s_q, t_q) = (-1, p_i) \quad & w_q = w_i, \quad \text{for the left boundary,}
\end{aligned}$$

where in this case $q$ is simply equal to $i$, since the boundary edges are one-dimensional intervals.

### 2.6 Adaptivity

It is worth recalling that so far no strong assumption has been made on the discretization degree. One of the main goals of the project is to allow a variable polynomial order. Due to the high degree of spatial locality of the Discontinuous Galerkin schemes, the adaptivity is implemented in a straightforward way. Dealing with the internal integrals, no issues arise. Indeed, all terms are local within $D^k$, and no communication with neighboring elements is necessary. Thus, all the computations can be carried out independently, with a possibly different $r$ in each element. On the other hand, some issues appear to be present when boundary integrals have to be computed. However, this is not the case, provided that the number of quadrature points is the same on a given edge. Indeed, suppose that two neighboring elements, say $D^k$ and $D^{\tilde{k}}$ have a different discretization degree $r^1 = r(k)$ and $r^2 = r(\tilde{k})$, with a possibly different number of quadrature points $Q_1^D$ and $Q_2^D$. Thus, two different expansions hold:

$$u_h^k = \sum_{j=1}^{(r^1+1)^D} u_j^k \phi_j, \quad u_h^{\tilde{k}} = \sum_{j=1}^{(r^2+1)^D} u_j^k \phi_j.$$

To compute the required integral, it is enough to evaluate the basis functions on the same set of quadrature points, so that the Rusanov flux can be simply computed. It is evident that the number of points has to be (at least equal to) the maximum between $Q_1$ and $Q_2$ in order not to perform any projection.
Thus, for static adaptivity no further complexities arise. We give the description of a simple algorithm for dynamic adaptivity, referring to [22] for further details. This is not yet fully validated, but we believe that this does not constitute a challenging task. The main difference is that some operators which are now assembled in the pre-processing step need to be updated at each timestep. The most evident example is the mass matrix, which depends on the spatial distribution of the polynomial degrees. We define the total energy of the solution as

$$\epsilon_{tot} = \left\| u_h^k \right\|^2_{L^2(D^k)} = \sum_{j=1}^{(r+1)^D} |u_j|^2 \left\| \phi_j \right\|^2,$$

Similarly, we define the energy contained in the highest modes as

$$\epsilon_r = \sum_{j \in \mathcal{R}} |u_j|^2 \, \|\phi_j\|^2 ,$$

where

$$\mathcal{R} = \left\{ j \ : \ \exists (l,m) \in \mathbb{N}^2 \text{ s.t. } j = l(r+1) + m + 1 \text{ with } (l \geq r) \vee (m \geq r) \right\}.$$

In other words, we take into account the energy of the modes having a degree at least equal to $r$ in at least one of the spatial variables. We finally define the quantity

$$w_r = \sqrt{\frac{\epsilon_r}{\epsilon_{tot}}},$$

and we consider an error tolerance *tol*. Then, the algorithm proceeds as follows:

1. Compute $w_r$.
2. If $w_r > tol$, then increase the degree $r$ by one and set the *new* coefficients to zero.
3. If $w_r \leq tol$, then compute $w_p$ ($p = r-1, \dots, 1$) until it exceeds the tolerance *tol*. Set the new degree equal to $p+1$.

### 2.7 Test cases

We explicitly define the main differential problems we consider in this report. We focus on two classes of partial differential equations, acting as prototype cases for scalar equations and systems of conservation laws. However, adding a support for other flux function is straightforward.
The first problem is linear advection equation

$$\frac{\partial u}{\partial t} + \nabla \cdot (\boldsymbol{\beta} u) = 0.$$

Provided that the transport field $\boldsymbol{\beta}$ is divergence-free, the solution is simply given by the initial condition advected in the direction determined by $\boldsymbol{\beta}$. Thus, this is used to validate the code, and represents a first challenging test in spherical geometry.
The second test represents a benchmark case for atmospheric dynamics and numerical weather prediction, as well as a good prototype for systems of conservation laws. Such equations are known as *Shallow Water Equations* (SWEs). They can be obtained starting from Navier-Stokes equations, assuming that the horizontal length scale is much greater than the vertical one [3]. Let $\eta$ be the free surface profile with respect a suitably defined reference level. Similarly, let $b \geq 0$ be the bathymetry profile, which is assumed to be constant in time. It is useful to further define $h = \eta + b$. Let $\boldsymbol{v} = (v_1, v_2)$ be the depth-averaged velocity field associated to the flux. The SWEs represent mass and momentum conservation and can be written as

$$\frac{\partial h}{\partial t} + \nabla \cdot (h\boldsymbol{v}) = 0,$$

$$\frac{\partial (h\boldsymbol{v})}{\partial t} + \nabla \cdot (h\boldsymbol{v} \otimes \boldsymbol{v}) = -gh\nabla\eta - f\boldsymbol{k} \times h\boldsymbol{v}. \tag{15a}$$

The left-hand-side of both equations fall in the standard DG framework and it is treated easily. On the other hand, the right-hand-side of (15a) can be rather challenging. The second term takes into account the rotation of the reference system, and it is known as *Coriolis* force. Since no differential operators are present, its discretization is easily treated as a standard volumetric term. The first term can be rather dangerous to deal with. In order to discretize it, the first option would be to integrate again by parts, obtaining

$$-\int_{D^k} g\eta \ \nabla(h\phi_i) + \int_{\partial D^k} g\eta^* \ h^* \boldsymbol{n} \ \phi_i. \tag{16}$$

By a second integration by parts we obtain the discrete differential operator $\mathcal{D}$ defined as

$$\int_{D^k} gh\ \mathcal{D}\eta\ \phi_i = \int_{D^k} gh\ \nabla\eta\ \phi_i + \int_{\partial D^k} g(\eta^* - \eta)h^*\boldsymbol{n}\ \phi_i. \tag{17}$$

In both (16) and (17), $\eta^*,\ h^*$ are numerical approximations of the free surface and the water height. Usually, an average of the neighboring variables is employed, in a way similar to a centered flux. The formulation (17) is preferred over (16), since we can guarantee the *well-balancing* property of the scheme [8]. A well-balanced scheme is able to preserve the so-called *lake at rest* solution, i.e.,

$$\eta_h = const,\ \boldsymbol{v}_h = \mathbf{0}.$$

In our case, the left-hand-side of (15a) is identically zero, as well as the Coriolis term. Using (16), the right-hand-side becomes zero only if the two integrals cancel out, which is not guaranteed unless exact integration is performed. This is clearly not possible in spherical geometry, due to the metric factors and cosine functions. However, using (17), both terms are zero independently. A few simplifications can be brought up in case of zero bathymetry profile $b$. In this case,

$$gh\nabla\eta = gh\nabla h = g\nabla\left(\frac{h^2}{2}\right).$$

Hence, this term can be included in the physical flux and treated in the standard conservation form. This guarantees the well-balancing property in a Cartesian geometry only. More details on the treatment of the right-hand-side can be found in, e.g., [22, 21, 8].

# 3 The C++ code

## 3.1 Compiling the code

Before providing the details on the Discontinuous Galerkin solver, we give the instructions on how to compile the code. We assume our working environment to be Piz-Daint at CSCS. Furthermore, the `scratch` folder represents our default location.
As a preliminary step, the following modules have to be loaded

```
module load daint-gpu
module load Boost
module load CMake
module load cray-trilinos/12.10.1.1
module load cray-netcdf-hdf5parallel/4.4.1.1.3
```

First, the *GridTools* (GT) library has to be installed. We simply clone the suitable version of GT we exploit as

```
git clone https://github.com/eth-cscs/gridtools.git && cd
   gridtools && git checkout tags/1.07.00
```

In order to avoid a compilation error, the following modification (not included in the released version) has to be carried out

```
sed -i 's/index_type/index_t/g' /gridtools-1.07.00/include/
   gridtools/stencil-composition/structured_grids/iterate_domain.
   hpp
```

This fix appears to be included in more recent releases, as the 1.08.00. After creating a building directory,

```
mkdir build && cd build
```

we simply run the following command

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=<GT_install_path> -
   DCMAKE_BUILD_TYPE=RELEASE -DUSE_GPU:BOOL=OFF -DCUDA_ARCH:
   STRING="sm_35" -DUSE_MPI:BOOL=OFF -DSINGLE_PRECISION:BOOL=OFF
   -DENABLE_CXX11:BOOL=ON -DENABLE_PYTHON:BOOL=OFF -
   DENABLE_PERFORMANCE_METERS:BOOL=ON -DCUDA_TOOLKIT_ROOT_DIR=
   $CRAY_CUDATOOLKIT_DIR -DBOOST_INCLUDEDIR=$EBROOTBOOST/include
   -DDISABLE_TESTING:BOOL=ON ..
```

and we finally install the library as

```
make -j && make install
```

At this point, the library should have been installed in the specified folder. The second step relates to the Galerkin4GridTools (G4GT) library, that we are going to modify. Essentially, most of the steps are similar as above.

```
git clone https://github.com/eth-cscs/galerkin4gt.git && cd
   galerkin4gt && git checkout advection_2d
```

Note that the C++ code described in the following is contained in the branch `advection_2d`. We create another build directory

```
mkdir build && cd build
```

and we run

```
cmake -DCMAKE_INSTALL_PREFIX=<G4GT_install_path> -DGRIDTOOLS_ROOT
   =<GT_install_path> cmake -DCMAKE_BUILD_TYPE=RELEASE -DUSE_GPU=
   OFF -DENABLE_NVCC_OUTPUT:BOOL=OFF -DENABLE_XDMF:BOOL=OFF -
   DENABLE_INTREPID:BOOL=ON -DINTREPID_SOURCE_DIR:PATH=
   $CRAY_TRILINOS_PREFIX_DIR/include/ -DINTREPID_LIB_DIR:PATH=
   $CRAY_TRILINOS_PREFIX_DIR/lib -DCUDA_TOOLKIT_ROOT_DIR=
   $CRAY_CUDATOOLKIT_DIR -DBLAS_HARDCODED_LIB=
   $CRAY_LIBSCI_PREFIX_DIR/lib/libsci_gnu_61.a -
   DLAPACK_HARDCODED_LIB=$CRAY_LIBSCI_PREFIX_DIR/lib/
   libsci_gnu_61.a -DCMAKE_CXX_FLAGS=-DBOOST_NO_CXX11_DECLTYPE -
   DENABLE_TESTS:BOOL=OFF ..
```

At this point, one should be able to compile all the required examples.
The compilation of the single examples is simply carried out modifying the file named

```
../examples/CMakeLists
```

In particular, the fifth row has to be set as

```
list( APPEND EXAMPLES <name_of_cpp_files_without_extension> )
```

and run the `make` command one last time.
From here on, our main folder will be the `galerkin4gt` one, unless stated otherwise.

### 3.2 An overview

Even though the code makes extensive usage of the Galerkin4GridTools library, it is worth adding some comments on the GridTools (GT) library, which constitutes the main tool for efficient computations.
In a nutshell, GT is a *C++ library for applications on regular or block regular grids*. Here, block regular refers to structured grids made of, e.g., rectangles. They constitute the basic tool to implement numerical solvers for partial differential equations. A good prototype is the Finite Differences method, for which the GT library is primarily designed. Their main drawback lies in the fact that high-order schemes are rather difficult to design, since they require the extension of the stencil, losing spatial locality. This issue is overcome by the Discontinuous Galerkin method, where a communication between an element and its *first* neighbors is needed. However, the extensive usage of template meta-programming and optimized tools for CPUs and GPUs make GT a very attractive library for efficient and fast computations. Practically, GT mainly supports Finite Differences codes. The degrees of freedom are simply the solution values at suitably defined points. GT lacks of the support for discretization schemes which allow multiple degrees of freedom inside the mesh elements, such as Finite Elements. Indeed, one would like to arbitrarily increase the discretization degree keeping a high degree of spatial locality. By definition, FE schemes rely on the weak formulation of the equation, and the concept of test and basis function has to be introduced. For a significant variety of solvers, one simply has to deal with polynomials evaluated at certain quadrature points. Since this is a generic task, the implementation is already available in multiple libraries. In this work, we employ *Trilinos*, which provides the support for FE discretization through the *Intrepid* library. Essentially, its usage can be viewed as simple way *not to reinvent the wheel*.
Therefore, G4GT can be interpreted as a library which links the above-mentioned ones. As shown in Section 3.3, Intrepid is needed only in a first phase. The storages returned by the library are then converted in similar ones which can be handled by GT. After this step, only GT is used to implement the solver. G4GT constitutes also a higher level to GT, which can be easier to understand for the final user. Most of the technicalities are hidden, while one has to focus only on the implementation of the methods required by the numerical algorithm. This is handled in a rather user-friendly way. The main G4GT folder consists of several subfolders, among which three of them are worthy of mention:

- `./numerics/` contains the main tools to compute quantities required by a Finite Element discretization. Basically, it is the unique folder where Trilinos libraries are extensively used.
- `./functors/` contains the classes which perform the computations.
- `./examples/` contains the main files for the implemented test cases.

### 3.3 Discontinuous Galerkin solver

Due to the GT and G4GT design, we need to impose a couple of constraints, which might represent a limit in the concrete applicability of the code. However, changing the interface would require a huge code refactoring and it is left as an extension to this project.

- A three-dimensional discretization. Even though the differential problem is purely two-dimensional, we embed it into a three-dimensional framework. The description of the numerics is unchanged, since tensor product arguments are still valid in order to define basis functions and quadrature points. Moreover, setting a two-dimensional initial condition, the same pattern is kept throughout the simulation. The unique drawback of this approach lies in the code performance, since both memory and computational time is spent to deal with

a possibly large number of degrees of freedom in the third dimension, which are practically useless. However, we recall that GT is designed for three-dimensional problems, so that treating only a two-dimensional discretization might also slow down the performance. A single element in the third dimension is considered, with $z$-width always equal to 2.

- Same number of quadrature points. We recall that the goal is the implementation of an adaptive scheme, with variable discretization degree. However, due to the G4GT implementation (and in particular due to the discretization of the affine mapping $\Psi$), it is easier to assume that the number of quadrature points does nor change in space and time. Clearly, we set it to be at least equal to $r_{max} + 1$, in order to perform exact computations for polynomials having degree up to $2r_{max} + 1$. Again, this limitation might affect the performances of the code. Indeed, suppose that $r_{max} = 4$. Then, the number of quadrature points is at least equal to $5^3 = 125$. When computing integrals of linear polynomials ($r = 1$), 8 points suffice to perform exact integrations. We stress the fact that this constitutes a drawback only in case of variable polynomial degrees. On the other hand, our choice does not have an impact on the CFL condition (14), since it is affected by the discretization degree only.

### 3.3.1 User-definable parameters

We first focus on the variables the user can arbitrarily modify. The program for the Cartesian case can be run as

```
SWE_RK <a> <b> <c> <d> <e> <f> <d1> <d2> <d3> <n_it> <dT>
```

where the first six values determine the three-dimensional computational domain $[a, b] \times [c, d] \times [e, f]$. The following three set the number of mesh elements in the three spatial directions. The last two values represent the number of time steps and the timestep $\Delta t$ respectively. One should set $e = -1$, $f = 1$ and $d_3 = 1$ in order not to waste more resources than necessary. Similarly, the code relying on spherical geometry can be run as

```
SWE_sphere <d1> <d2> <d3> <n_it> <dT>
```

with obvious meaning of the parameters.
Second, a set of parameters have to be provided

```
static constexpr ushort_t order_max = 3;
static constexpr ushort_t order_mapping = 1;
static constexpr ushort_t order_poly = 2*(order_max);
static constexpr uint_t order_RK=3;
static constexpr ushort_t type = 1;
```

The maximum polynomial order in the scheme is set in the variable `order_max`. Its main goal is to define the size of the storages. On the other hand, `order_mapping` refers to the degree needed to discretize the geometric mapping $\Psi$ from the reference to the physical elements. Dealing with structured meshes, such a mapping is affine and a degree equal to one is enough for our goals. This argument remains true even in the spherical case, since a longitude-latitude grid is considered. The polynomial order `order_poly` determines the degree of the polynomials that can be integrated exactly in a single spatial dimension. Usually, it is set to $O = 2r_{max}$. Consequently, the number of (one-dimensional) quadrature points is set as $Q =$ `<int>` $(O+2)/2$, which turns out to be equal to $r_{max} + 1$ in most of the cases. The variable `order_RK` governs the time integration scheme to be adopted, ranging from 1 to 4. Finally, the linear advection problem or the Shallow Water Equations are set choosing `type=0` or `type=1` respectively.

### 3.3.2 Finite-element discretization

Now, we describe how to compute the tools for finite element discretization. This is done using the Intrepid library, and the values are saved in storages which can be handled by G4GT. Note that all these values do not vary across the mesh, so that it is enough to save them once, making the storages accessible (read-only) by all the elements.

```
discretize_multidim<order_max, order_poly> discr_;
discr_.compute(Intrepid::OPERATOR_VALUE);
discr_.compute(Intrepid::OPERATOR_GRAD);

boundary_discretize_multidim<discretize_multidim<order_max,
   order_poly>,6> bd_discr_(0,1,2,3,4,5);
bd_discr_.compute(Intrepid::OPERATOR_VALUE);
```

The key role is played by the class `discretize_multidim`, whose template parameters are the maximum polynomial order and the degree of exactness of the quadrature rule. The term *multidim* refers to the fact that the computations have to be carried out for all the discretization degrees ranging from 1 to $r_{max}$ for adaptivity reasons. However, since the computation must be done at compile time, a standard *for* loop is not a viable option. This is handled using *template recursion*, by defining a template class `compute_helper` and its static method `compute_impl`. A base case is specified to break the loop. The actual computation is carried out using the class `intrepid::discretization`, which relies on the Trilinos library to extract the values. The tensor product is performed by hand, since there is no Intrepid method to get three-dimensional evaluations.

### 3.3.3 Assembler

Then, the `assembler` class is constructed. Essentially, it takes into account the mesh structure, providing methods to discretize the geometric mapping $\Psi$. However, its structure is simpler, since there is no need to compute values for different discretization degrees. Moreover, the number of quadrature points in space is kept constant.

```
using geo_map=reference_element<order_mapping, Lagrange, Hexa>;
using cub=cubature<order_poly, Hexa>;
using geo_t = intrepid::geometry<geo_map, cub>;
geo_t geo_;

using as = assembly_dg<geo_t>;
as assembler(d1,d2,d3);
assembler.compute(Intrepid::OPERATOR_GRAD);
assembler.compute(Intrepid::OPERATOR_VALUE);
```

### 3.3.4 Initialization

The next step is the initialization of the storages required by the computation. This is treated in the standard GT format, and it is omitted here. Moreover, both the initial condition and the flux function have to be specified. To keep it simple, we rely on *lambda functions*. We also use the `gt::array` class, since we want to have a general structure which can be used for both scalar equations and systems of conservation laws.

```
//Initial condition
gt::array<std::function<float_type (float_type ,float_type ,
   float_type)> , nb_eq> u_init;
u_init[0]=[](float_type x, float_type y, float_type z){return
   0.;};
...

//Flux function
gt::array<std::function<float_type (float_t,float_t,float_t,
   float_t,float_t,float_t)> , nb_eq*2> flux_function;
if (type==1) { //SWEs
flux_function[0]=[](float_type x, float_type y, float_type z,
   float_type u1, float_type u2, float_type u3){return u2;};
...
}
```

In this step, the initialization of the degree distribution has to be performed. The degree in each element is specified and stored in a dedicated object.

```
for (uint_t i=0; i<d1; i++)
        for (uint_t j=0; j<d2; j++)
                for (uint_t k=0; k<d3; k++)
                        //uniform initialization
                        order_loc_v(i,j,k)=order_max;
```

### 3.3.5 The Modal to Nodal class

Beyond the finite element discretization, a key role is also played by the handling of the conversion from modal to nodal values. This is required in at least two steps. Firstly, the initial condition is specified by assigning nodal values at a suitable set of disjoint points. However, the discretization is performed using a modal basis and a conversion has to be carried out. Secondly, at the final simulation time the obtained modal coefficients have to be converted to nodal values for post-processing and visualization purposes.
The following change of basis argument holds. Let us focus on a single mesh element, dropping the superscript $k$ for simplicity. We have

$$u(\boldsymbol{x}_i) = \sum_{j=1}^{(r+1)^D} u_j \phi_j(\boldsymbol{x}_i) = \boldsymbol{V}\boldsymbol{u},$$

where the first equality holds thanks to the expansion (7) and the second holds defining the Vandermonde matrix $\boldsymbol{V}$ as

$$\boldsymbol{V}_{ij} = \phi_j(\boldsymbol{x}_i). \tag{18}$$

The number of columns of $\boldsymbol{V}$ clearly equals the number of basis functions. On the other hand, the number of rows can, in principle, be arbitrary. One simply has to select a certain set of disjoint points. Therefore, in general $\boldsymbol{V}$ is a rectangular matrix. However, to perform the conversion from nodal to modal, $\boldsymbol{V}$ must be invertible. Thus, the cardinality of the set of the nodal points must be equal to the basis cardinality of the given element. The choice we adopt is to consider a tensor product of uniformly spaced points within the one-dimensional reference interval. On the other hand, to perform the conversion from modal to nodal a good choice is to keep the same set of points, independently of the discretization degree. This would result in a

tensorial grid which can be treated easily to post-process the results. Otherwise, an interpolation would be necessary.
We discuss now the key part of the code, which relies on a template class `M2N_multidim`. The method `set_multi_matrix` is required to compute the Vandermonde matrix (18) and, in principle, could be merged in the constructor of the class. Again, it relies on template recursion and to a class `M2N` to compute all the required matrices. The `N2M_multi_convert` method performs the conversion from nodal values, stored in `Storage1`, to modal values, stored in `Storage2`, supposing that the local degree is specified in `Storage3`. The equivalent version performs the opposite conversion as `M2N_multi_convert`. It is not shown here, since it is not currently used. The last method we report is indeed used to carry out the modal to nodal conversion, saving the nodal values on a uniform grid, independently on the discretization degree. Its structure is equivalent to the previous ones.

```
M2N_multidim<order_max,(order_poly+2)/2> mod2nod;
mod2nod.set_multi_matrix();

mod2nod.N2M_multi_convert(u_0_nodal_v,u_v,order_loc_v,d1,d2,d3,
    nb_eq,false);
//Defined as
//template <typename Storage1, typename Storage2, typename
    Storage3>
//void N2M_multi_convert( Storage1 const & in, Storage2 & out,
    Storage3 const & orders, uint_t const d1, uint_t const d2,
    uint_t const d3, uint_t const nb_eq=1, bool print=false){...}

mod2nod.M2N_multi_convert_enriched(u_v,u_nodal_v,order_loc_v,d1,
    d2,d3,nb_eq,false);
```

A couple of further comments have to be made. Firstly, this class is never used in the temporal loop. Secondly, matrix inversions have to be carried out. Their size is usually rather small and the computation of the inverse has to be done only once. Thus, we rely on the *Epetra* library from Trilinos. It is used in the class `M2N`, whose goal is to carry out conversions for a given degree. The implementation idea is similar to the finite element discretization described above. The key methods are the ones performing the matrix-vector multiplication using the Vandermonde matrix or its inverse and the `invert()` one. The latter relies on Epetra solvers to invert matrices.

### 3.3.6 The constant-in-time operators

We are ready to begin with the description of the main part DG solver. For simplicity, we consider the Cartesian geometry framework, with a natural extension to the spherical case. The main difference consists in multiplications with $\cos\theta$ when required.
First, we assemble finite-dimensional operators, which are constant in time. Here, GT syntax is used. The domain is made of all mesh elements, since such quantities have to be computed for each rectangle.
Concerning the boundary part, the determinant of the boundary Jacobian matrix has to be computed in each quadrature point. Since the mesh is structured, the determinant is constant and equal to the ratio between the edge lengths of the physical and the reference element. The normal outwards unit vectors are also computed for each face of the elements.The treatment of the internal part is rather similar. First the determinant of the Jacobian matrix is computed in all the quadrature points, and it is equal to the ratio between the volumes of the physical and

the reference element. The mass matrix and its inverse are also computed, since we assume a constant-in-time degree distribution. In case of a Cartesian geometry, $\boldsymbol{M}$ and $\boldsymbol{M}^{-1}$ are both diagonal. On the other hand, on the sphere only a block-diagonal structure is present. Thus, we rely on Epetra to invert each block, in order to avoid coding a solver for linear systems.

```
 auto compute_assembly=gt::make_computation< BACKEND >( coords,
    ..., gt::make_multistage ( execute<forward>()

//Boundary part
, gt::make_stage<functors::update_bd_jac<as::boundary_t::
   boundary_t, Hexa> >(p_grid_points(), p_bd_dphi_geo(), p_bd_jac
   ())
, gt::make_stage<functors::measure<as::boundary_t::boundary_t, 1>
    >(p_bd_jac(), p_bd_measure())
, gt::make_stage<functors::compute_face_normals<as::boundary_t::
   boundary_t> >(p_bd_jac(), p_ref_normals(), p_normals())

//Internal part
, gt::make_stage<functors::update_jac<geo_t> >(p_grid_points(),
   p_dphi_geo(), p_jac())
, gt::make_stage<functors::det<geo_t> >(p_jac(), p_jac_det())
, gt::make_stage<functors::mass_extended >(p_jac_det(), p_weights
   (), p_phi(), p_psi(), p_mass(), p_order())
, gt::make_stage<functors::inv_diag_extended > (p_mass(),
   p_mass_inv(), p_order())
, gt::make_stage<functors::inv<geo_t> >(p_jac(), p_jac_det(),
   p_jac_inv())

));
```

### 3.3.7 Temporal loop: common part

From here on, all steps are repeated at each timestep. Therefore, their implementation has to be efficient in terms of memory and floating point operations.
The first part is again common to all mesh elements and it is made, in turn, of two main blocks. The first one is the initialization process, setting to zero the temporary storages. The second one aims at computing the nodal values of the solution and the flux function, which will be used to compute integrals.

```
auto common=gt::make_computation< BACKEND >(
domain_iteration, coords, ..., gt::make_multistage ( execute<
   forward>()

//Initialization
, gt::make_stage <functors::RK123_initialize> (it::p_u_old(), it
   ::p_U0(), it::p_U1(), it::p_U2(), it::p_U3(), it::p_u_temp(),
   it::p_order(), it::p_currentRK() )
, gt::make_stage<functors::assign<5,zero<int> > > (it::p_result()
   )
, gt::make_stage<functors::assign<5,zero<int> > > (it::p_result2
   ())
```

```
//Compute values at quadrature points
, gt::make_stage<functors::compute_u_int_extended> (it::p_u_temp
   (),it::p_phi(), it::p_u_t_phi_int(), it::p_order())
, gt::make_stage<functors::compute_u_bd_extended> (it::p_u_temp()
   ,it::p_bd_phi(), it::p_u_t_phi_bd(), it::p_order())
, gt::make_stage<functors::compute_flux_function_int<nb_eq> > (it
   ::p_u_t_phi_int(),it::p_flux_function(),it::p_fun_int())
, gt::make_stage<functors::compute_flux_function_bd<nb_eq> > (it
   ::p_u_t_phi_bd(),it::p_flux_function(), it::p_fun_bd())

));
```

### 3.3.8 Temporal loop: boundary fluxes

This second part is clearly the most critical one, since communication between neighboring elements is present. In principle, no significant differences in the treatment of internal or boundary mesh elements are present, at least from a theoretical point of view. Indeed, the definition of the integrals over the element boundary remains the same. However, dealing with periodic boundary conditions, the elements that have to be accessed change consistently. For internal elements, only the first neighbors have to be accessed. Communication is done with the top, bottom, left and right element only. On the other hand, for boundary elements, we have to access the element at the opposite side of the domain, since periodic boundary conditions are applied. Therefore, a different treatment has to be considered for internal and boundary elements, changing the computational grid they have access to. An alternative approach, with the goal of treating internal and boundary elements equally, would consist in adding a ghost element at the left and right boundary. This is the standard way in which periodic conditions are treated in GT. This is undoubtedly an advantage, since it would simplify the structure of the code. However, this requires compiling the code with MPI, which have never been tested with G4GT. Moreover, our implementation guarantees a greater control on the single functors, since communication is coded explicitly.
As an example, consider the computation of left-right fluxes. Here, *internal* refers to all elements not having a boundary edge lying on the left or right boundary. In other words, defining the grid, the $x$ range goes from 1 to $d_1 - 2$ (instead of 0 to $d_1 - 1$).

```
auto coords_lr = gt::make_grid({1, 1, 1, d1 - 2, d1},
{0, 0, 0, d2 - 1, d2},
d3);

auto fluxes_lr=gt::make_computation< BACKEND >(
coords_lr, ..., gt::make_multistage ( execute<forward>()

, gt::make_stage< functors::Rusanov_lr > (it::p_u_t_phi_bd(), it
   ::p_fun_bd(),it::p_alpha(), it::p_normals(),  it::p_Rus())

));
```

The `Rusanov_lr` functor computes the flux itself, by accessing the neighboring elements. It is interesting to observe that the access to the neighboring elements is done as

```
storage(opposite_i,0,0,qp,face_opposite))
```

using an offset equal to plus or minus one. Concerning the left boundary, in the definition of the computational grid the third and the fourth indexes have to be equal to zero, since only the elements to the left have to be considered. Moreover, to compute the flux, a right offset equal to $d_1 - 1$ has to be considered, leading to a nonzero second index. The functor `bc_left` is essentially equivalent to the `Rusanov_lr` one, except for the access of the *neighboring* element. Again, it is done as

```
storage(opposite_i,0,0,qp,face_opposite))
```

but the variable `opposite_i` is instead defined as

```
short_t opposite_i = (short_t)(face_==1)?1:(face_==3)?d1-1:0;
```

A similar reasoning holds for the right boundary.

```
auto coords_bd_left = gt::make_grid({0, d1-1, 0, 0, d1},
{0, 0, 0, d2 - 1, d2},
d3);

auto compute_bc_left=gt::make_computation< BACKEND >(
coords_bd_left, ..., gt::make_multistage ( execute<forward>()

, gt::make_stage< functors::bc_left<noflux> > (it::p_u_t_phi_bd()
   , it::p_fun_bd(),it::p_alpha(), it::p_normals(),  it::p_Rus(),
    it::p_d1())

));
```

### 3.3.9 Temporal loop: main computation

In this last block, all previous terms are assembled together, computing the internal and boundary integrals, and advancing in time. The computational grid consists of all mesh elements.

```
auto RKstep=gt::make_computation< BACKEND >(
domain_iteration, coords, ..., gt::make_multistage ( execute<
   forward>()

//Assemble the right-hand-side (i.e. inv(Mass)*A(u))
, gt::make_stage<functors::compute_integral_int_extended>(it::
   p_fun_int(), it::p_dphi(), it::p_jac_det(), it::p_weights(),
   it::p_jac_inv(), it::p_result(), it::p_order())
, gt::make_stage<functors::compute_integral_bd_extended>(it::
   p_Rus(), it::p_bd_phi(), it::p_bd_measure(), it::p_bd_weights
   (), it::p_result(), it::p_order())
, gt::make_stage<functors::compute_coriolis<nb_eq,type> > (it::
   p_coriolis_fun(),it::p_qp(), it::p_u_t_phi_int(), it::p_phi(),
    it::p_jac_det(), it::p_weights(),it::p_result(), it::p_order
   (), it::p_time())
, gt::make_stage <functors::matvec_extended>( it::p_mass_inv(),
   it::p_result(), it::p_result2(), it::p_order() )

//Advance to the next step
```

```
, gt::make_stage <functors::RK123_update<order_RK> > (it::p_u_old
   (), it::p_RHS1(), it::p_RHS2(), it::p_RHS3(), it::p_RHS4(), it
   ::p_U0(), it::p_U1(), it::p_U2(), it::p_U3(), it::p_result2(),
    it::p_dT(),  it::p_u_new(), it::p_order(), it::p_currentRK()
   )

));
```

#### 3.3.10 Output

Two main output files are produced.

- `grid.dat`. It contains the grid information for visualization purposes. Uniformly spaced points are considered, including the extrema of each rectangle.
- `sol_N.dat`, where $N \in \{0, \dots, n_{it}\}$. It contains the solution values at the nodal points saved in the previous file. In all cases, the initial condition is saved with index 0, while the user can arbitrarily set the output frequency, which is set equal to $n_{it}$ by default.

## 4 Numerical results

We now present a few numerical results to show the potential of DG schemes.

### 4.1 A smooth problem

Let us start with the Cartesian geometry. The first test we propose is mainly needed to validate the implementation, as well as the accuracy and the stability properties of the scheme. Let us consider the linear advection problem with a constant transport field $\beta = (1, 1)$. Then, we choose $\Omega = [0, 1]^2$ as the two-dimensional computational domain, while the initial condition is

$$u_0(x, y) = \sin(2\pi x) \sin(2\pi y).$$

To validate the scheme, we consider the $L^2(\Omega)$-norm of the error vector evaluated at the final simulation time. Thus, let

$$\epsilon = \|u_h(\cdot, T) - u(\cdot, T)\|_{L^2(\Omega)}$$

Since the exact solution $u$ is the initial condition advected by the transport field, $\epsilon$ can be easily computed. Moreover, due to the spatial locality of DG schemes, it can be computed independently in each element, summing the contributions thanks to the linearity property of the integrals. Theoretical results for hyperbolic problems [15] suggest that the following optimal estimate holds

$$\epsilon \leq C_1 h^{m+1} (1 + C_2 T) \sim h^{m+1}, \tag{19}$$

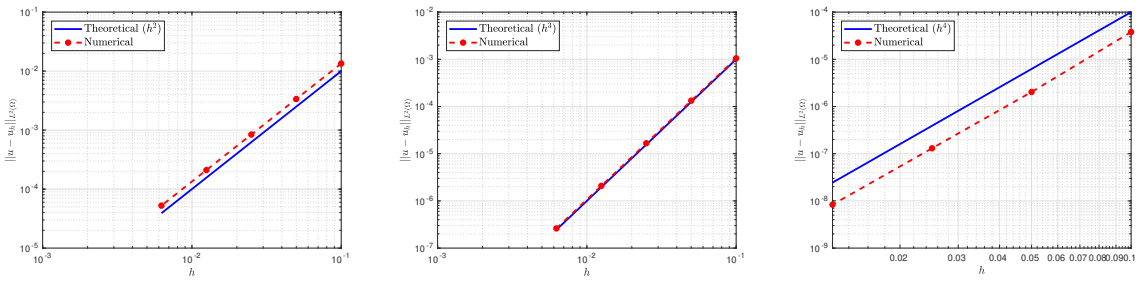
where $C_1, C_2$ are functions of $r$, provided that $u$ is sufficiently smooth and a Lax-Friedrichs/Rusanov flux is used. To estimate the convergence rate itself, we compute the discretization error using two different meshes with characteristic sizes $h_1,\ h_2$ (or equivalently with number of elements equal to $K_1,\ K_2$). Then, the estimated order, denoted by $p$, is computed as

$$p \simeq \frac{\log(\epsilon_1) - \log(\epsilon_2)}{\log(h_1) - \log(h_2)} = \frac{\log(\epsilon_1) - \log(\epsilon_2)}{\log(K_2) - \log(K_1)}.$$

In Table 1, we report the results obtained at time $T = 0.25$ using a Courant number equal to $C = 0.005$ and the RK4 integration scheme for polynomial orders ranging from 1 to 3. The corresponding graphical representation is shown in Figure 1. We obtain the desired convergence

| $K$ | $r = 1$ | | $r = 2$ | | $r = 3$ | |
|---|---|---|---|---|---|---|
| | $\epsilon$ | $p$ | $\epsilon$ | $p$ | $\epsilon$ | $p$ |
| $10^2$ | 1.3436e−2 | - | 1.0505e−3 | - | 3.7809e−5 | - |
| $20^2$ | 3.3692e−3 | 2.00 | 1.3297e−4 | 2.98 | 2.0306e−6 | 4.22 |
| $40^2$ | 8.4050e−4 | 2.00 | 1.6664e−5 | 3.00 | 1.3024e−7 | 3.96 |
| $80^2$ | 2.0993e−4 | 2.00 | 2.0844e−6 | 3.00 | 8.3458e−9 | 3.96 |
| $160^2$ | 5.2469e−5 | 2.00 | 2.6115e−7 | 3.00 | | |

Table 1: $L^2$ errors $\epsilon$ and estimated rate of convergence $p$ with the linear advection problem.




(a) $r = 1$ (b) $r = 2$ (c) $r = 3$

Figure 1: Graphical representation of the results reported in Table 1.

rate for all orders. This remains true even for coarse meshes, even though the estimate (19) holds at an asymptotic level. Note that the time-integration algorithm guarantees a fourth order accuracy, not affecting the benefits of an increasing polynomial order. However, due to (14), in case $r \geq 4$ is used, a sub-optimal convergence is likely to be present.

### 4.2 Gravity waves

We present now a couple of test cases for the SWEs. First, we analyze the propagation of pure gravity waves, with a Coriolis parameter $f$ set to zero. Take the steady state equilibrium solution of the SWEs provided by a constant fluid height and a zero velocity field. Consider a small perturbation of this state

$$h = h_0 + \epsilon h', \quad v_x = 0 + \epsilon v_x', \quad v_y = 0 + \epsilon v_y', \tag{20}$$

where $\epsilon \ll 1$ is a small parameter. Injecting (20) into (15) and manipulating the system, we obtain the following two-dimensional wave equation [20]

$$\frac{\partial^2 h'}{\partial t^2} - g h_0 \, \Delta h' = 0,$$

which describes the propagation of a wave characterized by a speed

$$c = \sqrt{g h_0}.$$

It is evident that such waves are driven by gravity only.
This argument motivates the following test case. Consider a square domain $\Omega = [0, L]^2$ with

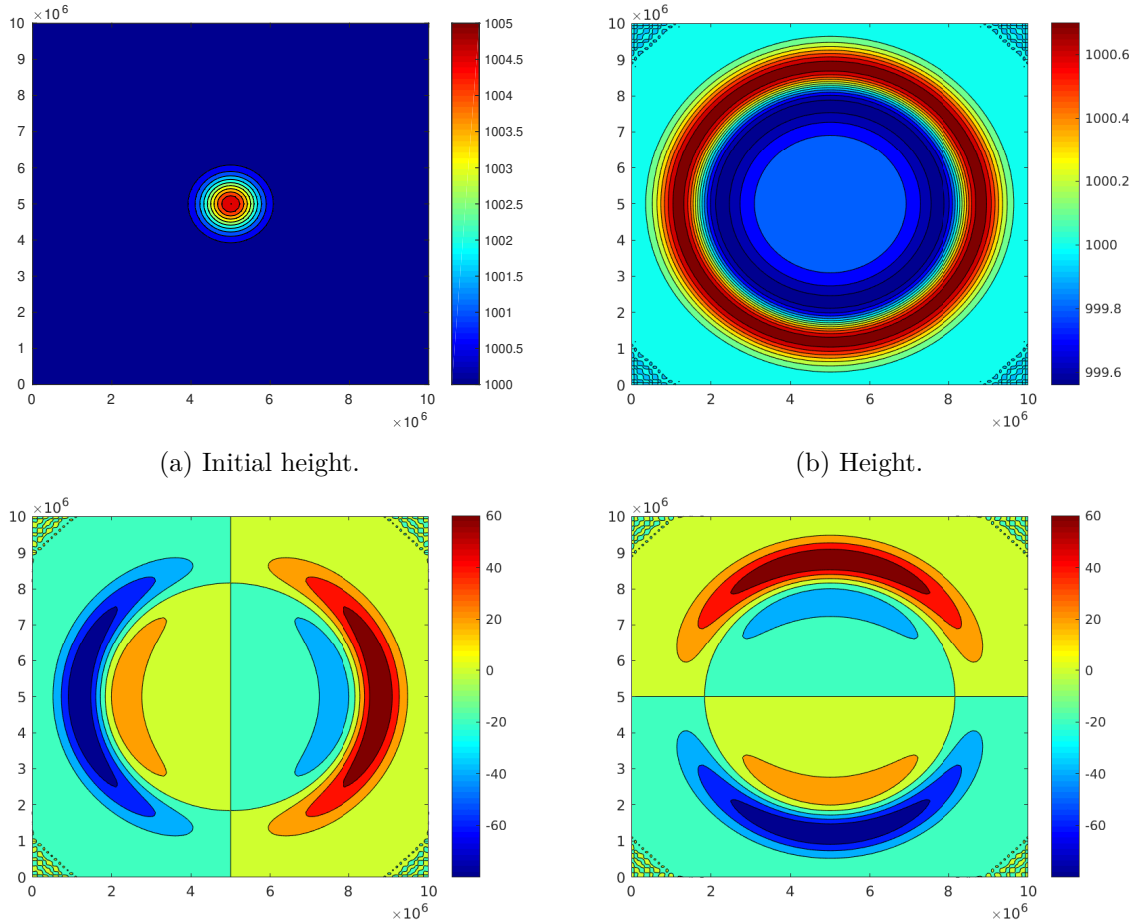
$L = 10^7$ m. The initial velocities and momenta are set to zero, while the height $h$ is equal to

$$h = h_0 + h_1 \exp\left(\frac{(x - L/2)^2 + (y - L/2)^2}{2\sigma^2}\right),$$

where $h_0 = 1000$ m, $h_1 = 5$ m and $\sigma = L/20$ m. This test case can be included in the previous framework by setting

$$\epsilon = \frac{h_1}{h_0} = 5 \cdot 10^{-3} \ll 1.$$

The final simulation time is $T = 36000$ s, so that the wave crest travels for a distance equal to $\Delta r = cT = \sqrt{gh_0}\ T = 3.5656 \cdot 10^6$ m in the radial direction. The simulation has been run using $d_1 \times d_2 = 50 \times 50$ elements, a discretization degree $r = 3$ and the RK4 scheme with a time step $\Delta t = 100$ s. Qualitatively, no differences are present by varying $r$, except for the fact that the accuracy improves, at least by a visual perspective. The results are reported in Figure 2. We stress on the fact that the boundary conditions play a marginal role, since the final time is



(a) Initial height.

(b) Height.

(c) Horizontal velocity.

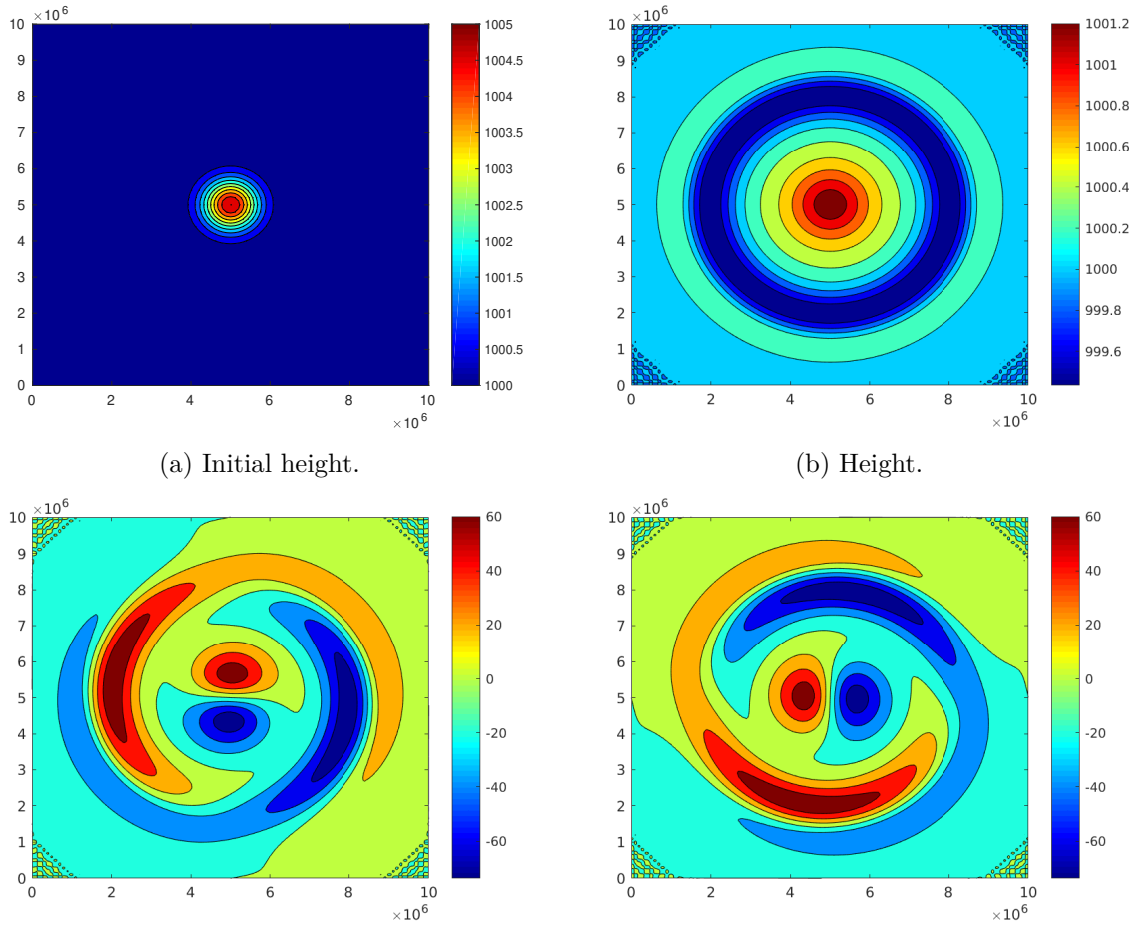
(d) Vertical velocity.

Figure 2: Numerical results for the propagation of pure gravity waves with $d_1 = d_2 = 50$, $r = 3$, RK4, $\Delta t = 100$.

small enough to guarantee that the waves do not exit from the computational domain. Some wiggles are present close to the boundaries, which are visual artifacts and do not affect the global behavior of the solution. Increasing the number of elements and/or the polynomial degree, this visual effect is reduced.

### 4.3 Geostrophic adjustment

A more challenging test case involves the presence of a nonzero Coriolis force. We want to show that the scheme is able to reproduce the process named *geostrophic adjustment*. In the previous scenario, all the energy was initially concentrated close to the center of the domain. Then, due to gravity, it entirely propagates away from the domain and it is contained in the crests of the wave. On the contrary, with a nonzero Coriolis term, gravitational and rotational forces combine. The result is that only part of the energy propagates due to gravity, and the solution peak remains in the center.
The computational domain, as well as the initial condition, is the same as the gravity wave propagation test case. The total number of elements remains unchanged, as well as the discretization degree and the time integration parameter. The Coriolis parameter $f$ is chosen to be constant and equal to $10^{-4}$ s$^{-1}$. This is known as $f$-plane approximation. The results are reported in Figure 3. Again, the low-amplitude wiggles close to the corners are purely artificial




(a) Initial height.

(b) Height.

(c) Horizontal velocity.

(d) Vertical velocity.

Figure 3: Numerical results for the geostrophic adjustment test case with $d_1 = d_2 = 50$, $r = 3$, RK4, $\Delta t = 100$, $f = 10^{-4}$.

and do not affect the qualitative behavior of the solution.

### 4.4 Spherical linear advection

The final test case we report deals with a spherical geometry. We want to guarantee that the numerical scheme is able to preserve a solution which is advected by a suitably defined transport field. Following [25], we select

$$u_0(\lambda,\theta) = \begin{cases} \frac{h_0}{2}\left(1+\cos\left(\pi\frac{r}{a}\right)\right) & \text{if } r < a \\ 0, & \text{otherwise} \end{cases}$$

where $h_0 = 1000$, $a = R/3$, and $r$ is the great circle distance between $(\lambda,\theta)$ and the center, initially taken as $(\lambda_c,\theta_c) = (3\pi/2,\pi/2)$

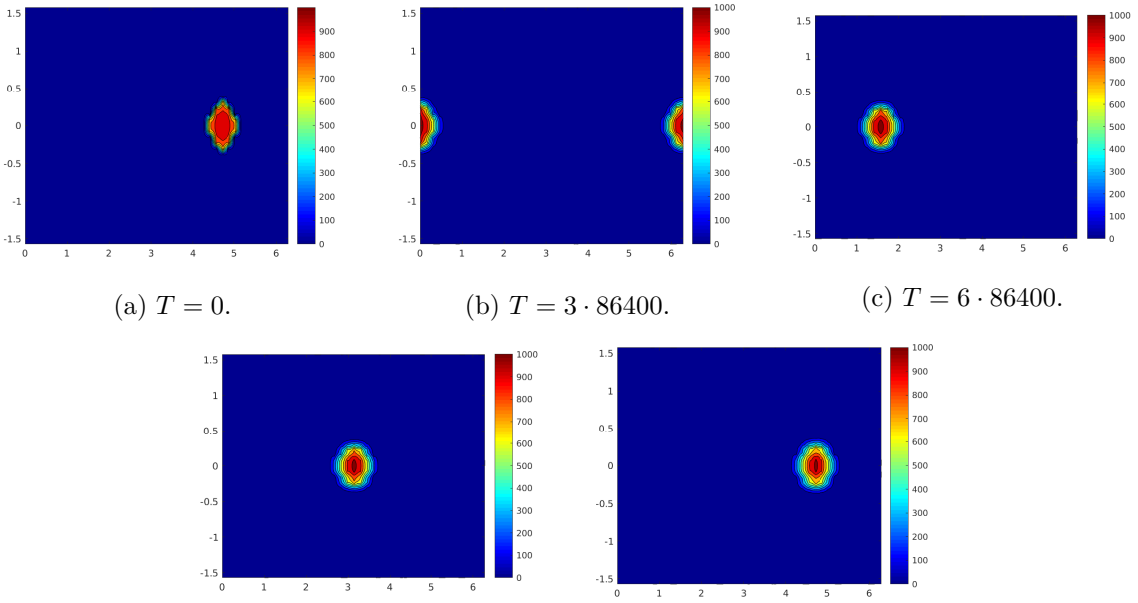
$$r = R\arccos\left(\sin\theta_c\sin\theta + \cos\theta_c\cos\theta\cos(\lambda-\lambda_c)\right).$$

The advecting wind $\boldsymbol{\beta}$ is given by the following divergence-free field

$$\beta^\lambda = u_0(\cos\theta\cos\alpha + \sin\theta\cos\lambda\sin\alpha)$$

$$\beta^\theta = -u_0\sin\lambda\sin\alpha$$

where $u_0 = 2\pi R/(12\cdot 86400)$ is the wind strength and $\alpha \in [0,\pi/2]$ is a parameter controlling the direction of the transport field. Considering now $d1 \times d2 = 40\times 40$ elements, $\Delta t = 50$, RK2 and $r = 2$, we show that the shape of the solution is preserved. The final time is $T = 12\cdot 86400$.
In Figure 4 we note that the results obtained setting $\alpha = 0$, corresponding to a latitudinal wind. The projection on the longitudinal-latitudinal does not have a significant effect on the shape



(a) $T = 0$. (b) $T = 3\cdot 86400$. (c) $T = 6\cdot 86400$.

(d) $T = 9\cdot 86400$. (e) $T = 12\cdot 86400$.

Figure 4: Numerical results for the linear advection on the sphere, $d_1 = d_2 = 40$, $r = 2$, RK2, $\Delta t = 50$, $\alpha = 0$.

of the solution, at least from a qualitative point of view. This is expected, since the initial condition is centered close to the Equator and the transport field has only a nonzero latitudinal component equal to $\beta^\lambda = u_0\cos\theta \approx u_0$. Although the solution is smooth and it qualitatively matches our expectation, the convergence rates turn out to be sub-optimal. Due to the large

scales involved, a better measure of the error is

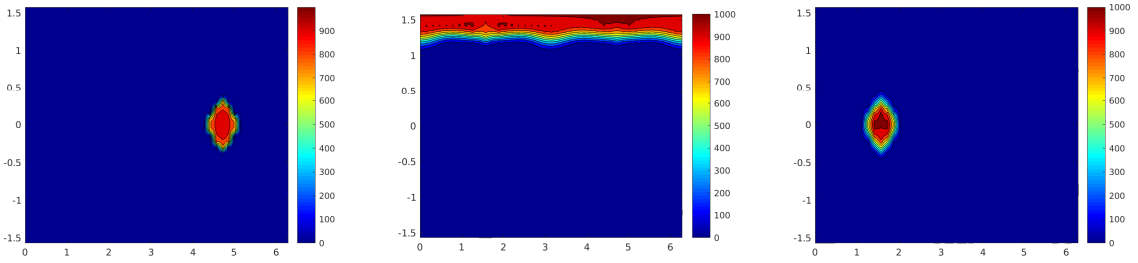
$$\epsilon = \frac{\|u_h(\cdot, T) - u(\cdot, T)\|_{L^2(\Omega)}}{\|u(\cdot, T)\|_{L^2(\Omega)}}.$$

The results, considering again a fourth order scheme, are reported in Table 2. Even though

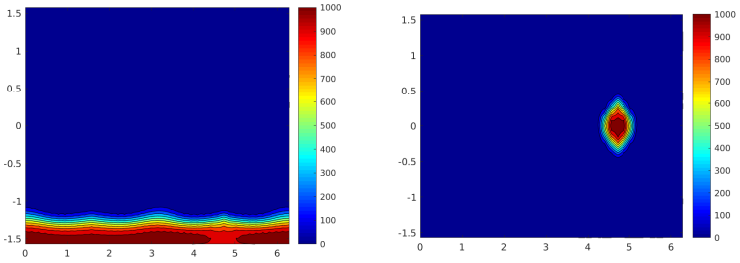
| $K$ | $r = 1$ | | $r = 2$ | | $r = 3$ | |
|---|---|---|---|---|---|---|
| | $\epsilon$ | $p$ | $\epsilon$ | $p$ | $\epsilon$ | $p$ |
| $10^2$ | 3.0864e−1 | - | 4.4843e−1 | - | 2.4989e−1 | - |
| $20^2$ | 4.5592e−1 | -0.56 | 2.1482e−1 | 1.06 | 2.0292e−1 | 0.30 |
| $40^2$ | 2.6224e−1 | 0.80 | 1.7553e−1 | 0.29 | 1.5463e−1 | 0.39 |
| $80^2$ | 1.9187e−1 | 0.45 | | | | |

Table 2: $L^2$ errors $\epsilon$ and estimated rate of convergence $p$ with the linear advection problem on the sphere, $\alpha = 0$.

multiple sources can lead to this loss of optimal convergence, including coding errors, we believe that quadrature errors constitute the main one. Indeed, the metric factors introduce an error with non-negligible magnitude. Having a closer look at the solution, one finds that several oscillations are present far away from the center of the advected cosine bell. Their amplitude is controlled, do not cause numerical instabilities, and do not qualitatively affect the solution profile. However, they can have a large effect on the accuracy, increasing numerical errors. Such oscillations are not reported in Figure 2 and we believe that could be killed introducing slope limiting or artificial dissipation [14, 11].
A similar behavior is found setting $\alpha = \pi/2$, where the solution crosses the poles. The results are reported in Figure 5, and the convergence rates, for $r = 1$ only, are shown in Table 3. Qualitatively, the results match our expectations. Close to the poles, the bell distributes its



(a) $T = 0$. (b) $T = 3 \cdot 86400$. (c) $T = 6 \cdot 86400$.



(d) $T = 9 \cdot 86400$. (e) $T = 12 \cdot 86400$.

Figure 5: Numerical results for the linear advection on the sphere, $d_1 = d_2 = 40$, $r = 2$, RK2, $\Delta t = 50$, $\alpha = \pi/2$.

| $K$ | $r = 1$ | |
|---|---|---|
| | $\epsilon$ | $p$ |
| $10^2$ | 5.8520e−1 | - |
| $20^2$ | 2.8642e−1 | 1.03 |
| $40^2$ | 1.9707e−1 | 0.54 |
| $80^2$ | 1.3441e−1 | 0.55 |

Table 3: $L^2$ errors $\epsilon$ and estimated rate of convergence $p$ with the linear advection problem on the sphere, $\alpha = \pi/2$.

energy at all latitudes, crossing both the singularities without numerical instabilities. It is worth observing that, matching all parameters but $\alpha$, the error magnitude is lower if $\alpha = \pi/2$. Even though this appears counterintuitive, a solution advected in the direction of the poles might help in killing the spurious oscillations, that we identified as one possible source of sub-optimal convergence.

# 5 Performance evaluation

Once the results are validated, a performance evaluation is carried out. We recall that the code makes usages of GT optimizations for a CPU architecture, and most of them are set in the

```
backend<BACKEND_ARCH, GRIDBACKEND, BACKEND_STRATEGY>
```

type. These definitions trigger several architecture-based optimizations, among which the most important is the *layout maps* of GT storages.
Concretely, the performance evaluation was performed using the Roofline model [24], which is based on two main variables that can be represented in a two-dimensional plot. The first one is the *operational intensity*, defined as the number of floating-point operations (flops) per byte of DRAM traffic, taking into account the bytes that are read from/written in the main memory. The second index is the number of *attainable Gflops per second*, i.e., the concrete performance measure. Because of hardware limits, the attainable flops per second cannot go beyond a fixed threshold, determined by the peak memory bandwidth and the peak floating point performance. Practically, this threshold is determined by running benchmark cases, as the *stream* or the *linpack* benchmark. Thus, for a given operational intensity, a computationally efficient kernel should have performance measures close to the upper limit.
In our analysis, we make the following simplifying assumptions:

- Ignore the cache effects. Every access to a variable is taken into account in the computation of the memory traffic. Even though this is in contrast with the definition provided in [24], a precise estimate of the DRAM traffic requires the use of external tools. An option relies on the *Intel* compiler, which directly computes the volume of data from/to the memory after the cache filter. However, G4GT has never been tested using the Intel compiler, and attempting to use it goes beyond the scope of this work.
- Nominal values are used to determine the Roofline model. The concrete hardware limits are generally below this threshold, thus the maximum attainable performance can be shifted below than the reported values. However, since no benchmark cases are available, we are forced to use nominal values.

In this work we perform tests on a CPU only. The simulations have been run on the compute nodes of Piz-Daint at CSCS, using an Intel Xeon E5-2670 v4 processor (single node). The

roofline is easily plotted knowing the values of the peak memory bandwidth (68 $GB/s$) and the peak floating point performance (41.6 $Gflops/s$), as reported in [1]. Concretely, the programs have been run with the following script

```
#!/bin/bash -l
#SBATCH --job-name=SWE
#SBATCH --time=00:10:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=24
#SBATCH --partition=normal
#SBATCH --constraint=gpu

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

srun <name_of_program> <parameters>
```

Here, we focus only on the time-dependent part of the solver. If we consider a reasonably large number of steps, the pre-processing phase does not represent the dominant part of the code. This pre-processing part is also far away from being fully optimized. Moreover, it is the only step where we make usage of Intrepid and Epetra. Since we have minimal control on Trilinos optimizations, it is rather difficult to fully trigger all of them.

### 5.1 Cartesian geometry

Firstly, we report the performance analysis for a benchmark case using the SWEs. A good compromise between computational cost and significance of the solution features is found in the simulation of the geostrophic adjustment. We refer to the test case presented in Section 4.3 for the physical parameters.
Let us consider $d_1 \times d_2 = 50 \times 50$ elements, $\Delta t = 100$, RK4. The results for varying polynomial degree $r$ are reported in Figure 6. No significant variations in the operational intensities are

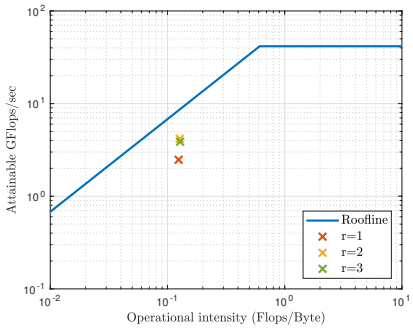



Figure 6: Performance evaluation of the DG solver for SWEs in a Cartesian geometry in case of different polynomial degrees.

present. A possible explanation is based on the assumption of no cache effects. Increasing the polynomial order leads to similar changes in the number of floating point operations and memory traffic. For large orders, the computation of the quantities internal to the element dominates. Hence, convergence to a fixed value will be reached. The attained performance values fall below the theoretical limit. We can find two possible reasons leading to this behavior. Firstly, taking into account the cache hierarchy we would have a lower computational intensity.

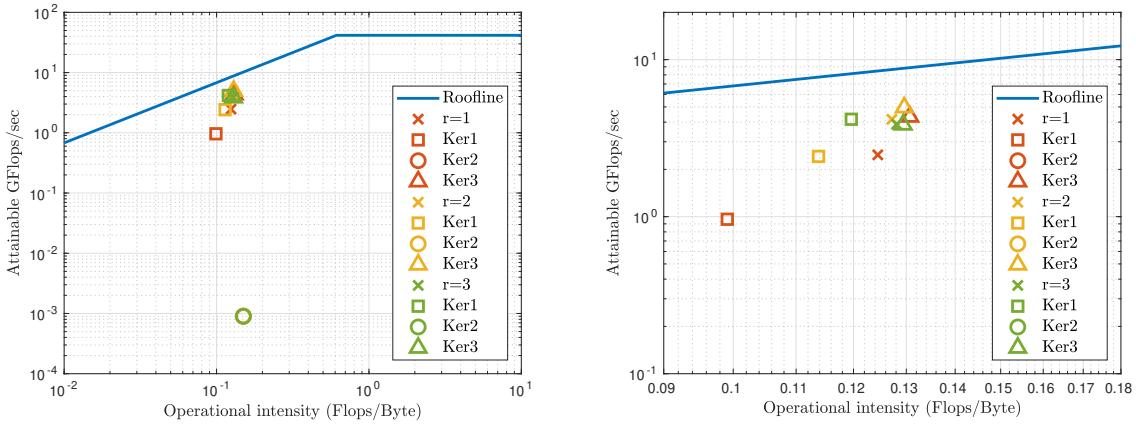
Thus, the same cost will be lie closer to the hardware limit, since a shift towards the left will be present. Secondly, using limits determined by benchmark cases would cause a shift of the upper bound towards the bottom. We also observe that performances obtained with $r = 1$ are lower compared to $r = 2, 3$. Optimizations might not be fully triggered using low orders, causing a loss in performances.
Recalling the way the code is structured, we can recognize three different kernels:

1. Common part (Section 3.3.7). The nodal values for solution and flux function are computed.
2. Boundary fluxes (Section 3.3.8). The boundary fluxes are computed, with communication with neighboring elements. Periodic boundary conditions are also applied.
3. Main computation (Section 3.3.9). The right-hand-side is assembled, and the solution is updated.

Thus, we can analyze the performances of each kernel independently, comparing them with the global program. The results are reported in Figure 7, while Table 4 shows the computational time and the energy consumption. The third kernel has the largest influence on the overall

| $r$ | Global [s] | Kernel 1 [s] | Kernel 2 [s] | Kernel 3 [s] | Energy [kJ] |
|---|---|---|---|---|---|
| 1 | 15.17 | 7.42 | 1.45 | 6.31 | 2.94 |
| 2 | 78.98 | 21.22 | 3.10 | 54.66 | 12.17 |
| 3 | 430.21 | 54.97 | 5.81 | 369.43 | 64.23 |

Table 4: Computational times of the DG solver for SWEs in a Cartesian geometry in case of different polynomial degrees (RK4 scheme).




(a) Global range.

(b) Zoomed range.

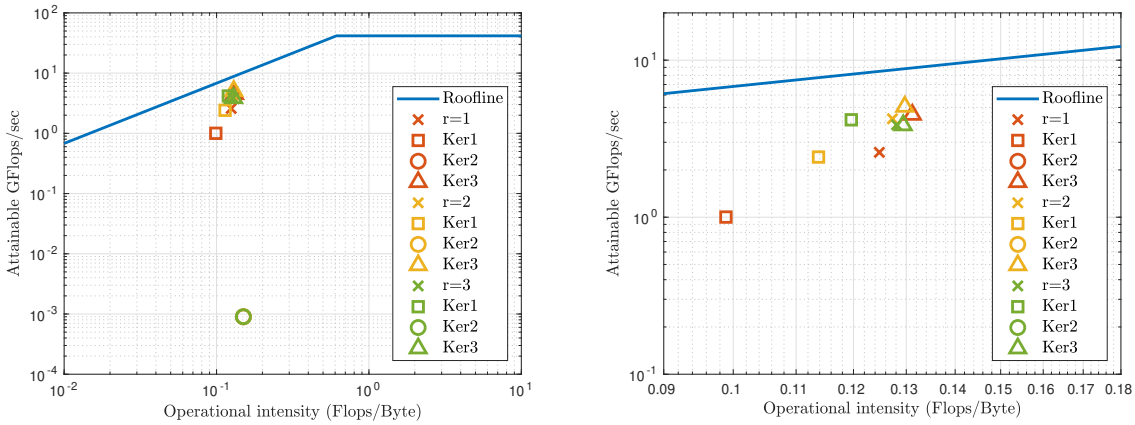
Figure 7: Performance evaluation of the DG solver for SWEs in a Cartesian geometry in case of different polynomial degrees (RK4 scheme). Comparison among the different kernels.

performance, at least asymptotically. This is expected, since most of the computations are performed in it. The assembly of the internal integral is the most intensive part, both in terms of resources and time. On the other hand, the second kernel has always low computational cost. Only boundary elements are considered, with computation of nodal values without any assembly of integral quantities. The performances of kernel 2 in terms of floating point operations per second do not vary with $r$ and are consistently low. Since it is the only phase which involves exchanges between neighboring elements, it is reasonable that cache-misses or non efficient memory accesses are present. A slightly different behavior is observed for $r = 1$. Here, the

dominant kernel is the first one, which affects global performance in a significant way.
It is interesting to compare the results obtained using a fourth order scheme with the ones obtained with varying RK methods. The corresponding order is chosen in such a way that the spatial and temporal orders match. For instance, choosing $r = 2$ we select a third order scheme. We directly switch to the comparison of the different kernels, shown in Figure 8, while Table 5 reports the computational time and the energy consumption. Most of the previous comments

| $r$ | Global [s] | Kernel 1 [s] | Kernel 2 [s] | Kernel 3 [s] | Energy [kJ] |
|---|---|---|---|---|---|
| 1 | 7.27 | 3.56 | 0.66 | 3.05 | 1.49 |
| 2 | 58.38 | 15.93 | 2.25 | 40.20 | 8.71 |
| 3 | 430.21 | 54.97 | 5.81 | 369.43 | 64.23 |

Table 5: Computational times of the DG solver for SWEs in a Cartesian geometry in case of different polynomial degrees (matched temporal order).




(a) Global range.

(b) Zoomed range.

Figure 8: Performance evaluation of the DG solver for SWEs in a Cartesian geometry in case of different polynomial degrees (matched temporal order). Comparison among the different kernels.

remain valid. From a global perspective, the Roofline models appear almost identical. However, the single simulation times change quite significantly, with a clear bias towards the temporal order matching the spatial one. Clearly, the drawback lies in the lower accuracy. However, due to spatial discretization and (14), the accuracy cannot improve much by choosing a high-order time-integration scheme if the spatial degree is kept low.

## 5.2 Spherical geometry

We now focus on the spherical geometry, studying the performances of the linear advection equation. We refer to Section 4.4 for the details.
We select $d_1 \times d_2 = 10 \times 10$ elements, $\Delta t = 100$ s, RK4. By varying the polynomial degree, we get the results shown in Figure 9, while analyzing the single kernels separately we obtain the results reported in Table 6 and Figure 10. The performance drop with respect to the Cartesian case is significant. Even though the $r = 1$ case is still the least efficient one, even with higher orders the results fall below the roofline by almost one order of magnitude. A part of such a drop can be explained again by noting that we are using nominal values and ignoring cache effects. However, we believe that non-friendly memory accesses are present. Essentially, the

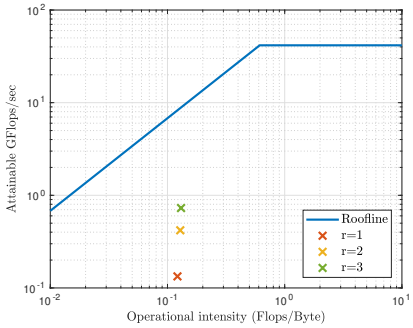
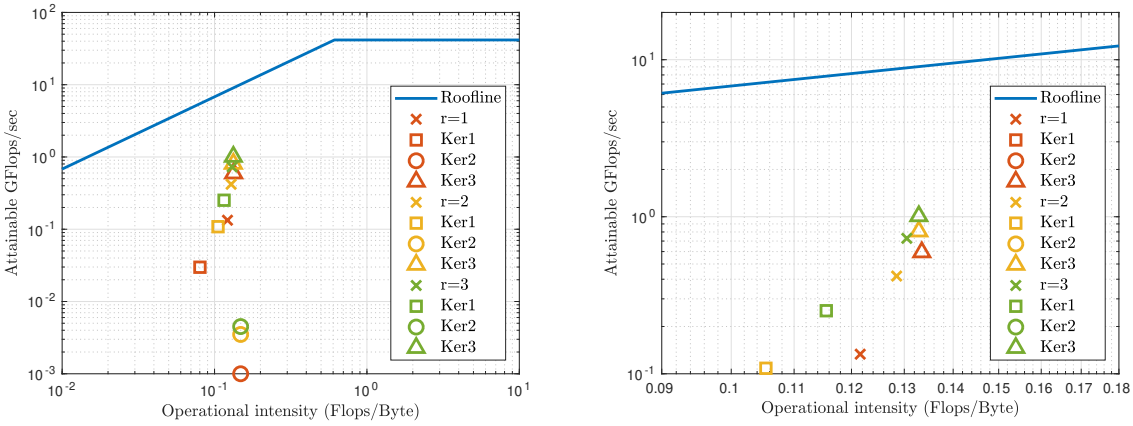



Figure 9: Performance evaluation of the DG solver for SWEs in a Cartesian geometry in case of different polynomial degrees.

| $r$ | Global [s] | Kernel 1 [s] | Kernel 2 [s] | Kernel 3 [s] | Energy [kJ] |
|---|---|---|---|---|---|
| 1 | 112.02 | 80.17 | 12.87 | 18.98 | 20.38 |
| 2 | 321.80 | 173.57 | 8.03 | 140.21 | 44.77 |
| 3 | 945.73 | 342.31 | 11.14 | 592.28 | 109.71 |

Table 6: Computational times of the DG solver for the linear advection equation in a spherical geometry in case of different polynomial degrees (RK4 scheme).




(a) Global range.

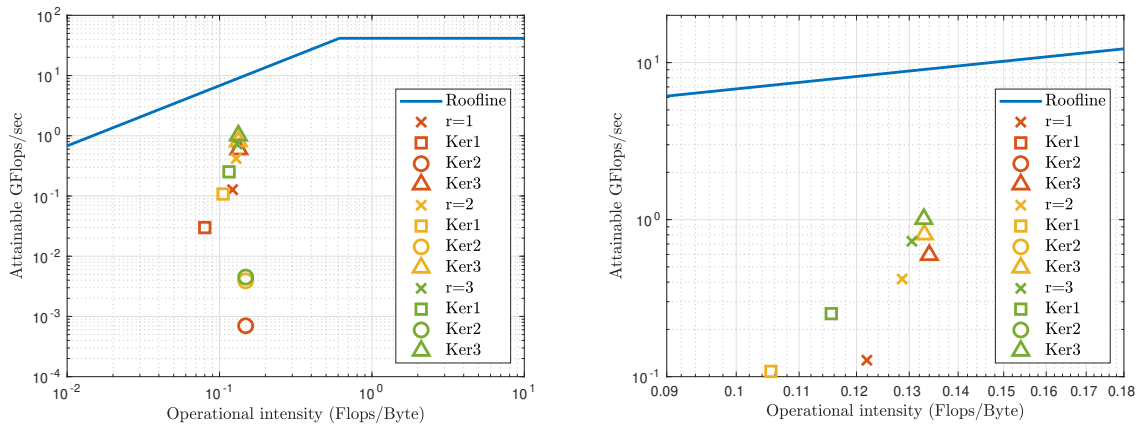
(b) Zoomed range.

Figure 10: Performance evaluation of the DG solver for linear advection equation in a spherical geometry in case of different polynomial degrees (RK4 scheme). Comparison among the different kernels.

main difference between Cartesian and spherical geometry is found in some additional metric factors. It could happen that the memory access and storage of such values is not efficient, causing the lower values. This behavior is confirmed by analyzing the kernels separately. Unlike the Cartesian case, in a spherical geometry the first kernel plays a significant role. Here, metric factors are needed to assemble the flux function, which is spatially dependent. The third kernel is still dominant, but in a less evident way. All performances drop with respect to the Cartesian case, so that we believe that the presence of metric factors causes less efficient programs.
Finally, we report the analysis in case of a variable temporal order in Table 7 and Figure 11. As in the Cartesian case, no significant performance variations are present in comparison with

| $r$ | Global [s] | Kernel 1 [s] | Kernel 2 [s] | Kernel 3 [s] | Energy [kJ] |
|---|---|---|---|---|---|
| 1 | 58.75 | 39.91 | 9.33 | 9.50 | 10.39 |
| 2 | 241.61 | 130.59 | 5.54 | 105.47 | 33.04 |
| 3 | 945.73 | 342.31 | 11.14 | 592.28 | 109.71 |

Table 7: Computational times of the DG solver for the linear advection equation in a spherical geometry in case of different polynomial degrees (matched temporal order).




(a) Global range.

(b) Zoomed range.

Figure 11: Performance evaluation of the DG solver for linear advection equation in a spherical geometry in case of different polynomial degrees (RK4 scheme). Comparison among the different kernels.

the fixed RK order. From a visual perspective, the roofline models look identical. However, the savings in term of computational cost are even more significant. As a general comment, it is worth observing that the elapsed time in the spherical case is much larger than the Cartesian case, mainly due to the larger number of steps. We also believe that the results are not consistent with high-performance computing applications, giving quite low performances. Indeed, it appears counterintuitive that approximately half an hour is required to simulate results on a mesh consisting of $10 \times 10$ elements only. The combination of non-friendly memory accesses, three-dimensional discretization, and possibly incorrect optimizations in the underlying libraries (in particular GT) lead to such a behavior.

# 6 Conclusion

In this project we implemented a Discontinuous Galerkin solver for conservation laws. The main library we used is GridTools, with additions provided by Trilinos libraries Intrepid and Epetra. A support for both scalar problems and the Shallow Water Equations is present for a Cartesian geometry. On the other hand, for a spherical geometry only the linear advection equation is fully tested. Even though most of the effort was spent on the non-adaptive case, the flexibility in changing the degree was added. Focus is given on static adaptivity, assuming that the degree distributions is not varied.
All results are consistent with our expectations. We validated the potential of the scheme using a smooth test case. More challenging examples involve the SWEs, e.g. the propagation of pure gravity waves or the geostrophic adjustment phenomenon. We extended the implementation to the advection on the sphere, studying the motion of a bell advected by a spatially-dependent wind. In all cases, the main solution features are captured by the scheme. We also provided a performance analysis using the Roofline model. The simplifying assumption about the cache hierarchy might represent a limit when commenting on the obtained results. However, we found that the performance is sufficiently close to the hardware limit, at least for a Cartesian geometry. In the spherical case, there is a drop in the performance, possibly due to the frequent access to the metric factors.
Multiple extensions to this project can be implemented, mainly from an implementation point of view. Support for SWEs in the spherical case appears not to be a critical issue. Most of the numerical details have been highlighted in this report, hence it may worth spending resources for their implementation. Generalizing the code to a variable number of quadrature points requires a change in the `assembler` class, which could be modified to handle a different number of points. This can drastically reduce the computational times for adaptive scenarios, especially when the range of the discretization degrees is large. A more general cleaning of the code should also be carried out. Among its advantages, it might help in understanding why the performances appear not to be optimal according to the Roofline model, mainly in the spherical case. It would be extremely helpful to employ performance analysis tools, in order to relax the assumptions on the cache effects. A performance analysis on GPU architectures could represent a milestone for the G4GT library developers. The preliminary results in this direction are extremely encouraging [12].

# References


[1] `https://ark.intel.com/it/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2-60-GHz-8-00-GT-s-Intel-QPI-`. [Online; accessed 16-Nov-2018].

[2] `https://en.wikipedia.org/wiki/List_of_Runge-Kutta_methods`. [Online; accessed 10-Aug-2018].

[3] `http://users.ices.utexas.edu/~arbogast/cam397/dawson_v2.pdf`. [Online; accessed 10-Aug-2018].

[4] `https://trilinos.github.io/intrepid.html`. [Online; accessed 16-Dec-2018].

[5] `https://trilinos.github.io/epetra.html`. [Online; accessed 16-Dec-2018].

[6] `https://users.ices.utexas.edu/~leszek/classes/EM386L/notes.pdf`. [Online; accessed 24-Oct-2018].

[7] `https://en.wikipedia.org/wiki/Curvilinear_coordinates`. [Online; accessed 24-Oct-2018].

[8] B. Bonev et al. "Discontinuous Galerkin scheme for the spherical shallow water equations with applications to tsunami modeling and prediction." In: *J. Comput. Physics* 362 (2018), pp. 425–448.

[9] S. Carcano. "Finite volume methods and Discontinuous Galerkin methods for the numerical modeling of multiphase gas-particle flows." Doctoral Thesis. Politecnico di Milano, 2014.
[10] M. H. Carpenter, a. Kennedy. "Fourth-Order Kutta Schemes." In: *Nasa Technical Memorandum* 109112 (1994), pp. 1–26.
[11] N. Discacciati. "Controlling oscillations in high-order schemes using neural networks." Master's thesis. EPFL and Politecnico di Milano, 2018.
[12] N. Discacciati. *Implementation of Discontinuous Galerkin model problem for atmospheric dynamics on emerging architectures*. Internship report. CSCS, 2017.
[13] O. Fuhrer et al. *GridTools: Towards a library for hardware oblivious implementation of stencil based codes*. `http://www.pasc-ch.org/projects/projects/grid-tools`.
[14] J. S. Hesthaven. *Numerical Methods for Conservation Laws: From Analysis to Algorithms*. SIAM Publishing, 2018, p. 555.
[15] J. S. Hesthaven, T. Warburton. *Nodal discontinuous Galerkin methods*. Vol. 54 TS - C. 2008, p. 500.
[16] A. Quarteroni. *Numerical Models for Differential Problems*. Milano: Springer, 2014.
[17] A. Quarteroni et al. *Matematica Numerica*. Milano: Springer, 2014.
[18] D. Ray, J. S. Hesthaven. "An artificial neural network as a troubled-cell indicator." In: *Journal of Computational Physics* 367 (2018), pp. 166–191.
[19] B. L. Rozhdestvenskii. "Discontinuous solutions of hyperbolic systems of quasilinear equations." In: *Russ. Math. Surv.* 15.53 (1960).
[20] S. Salsa. *Partial Differential Equations in Action*. 2008, p. 568. arXiv: `arXiv:1011.1669v3`.
[21] G. Tumolo, L. Bonaventura. "A semi-implicit, semi-Lagrangian Discontinuous Galerkin framework for adaptive numerical weather prediction." In: *Q.J. Royal Meteorological Society* (2015).
[22] G. Tumolo, L. Bonaventura, M. Restelli. "A semi-implicit, semi-Lagrangian, p-adaptive discontinuous Galerkin method for the shallow water equations." In: *Journal of Computational Physics* (2013).
[23] Z. J. Wang et al. *High-order CFD methods: Current status and perspective*. 2013. arXiv: `fld.1 [DOI: 10.1002]`.
[24] S. Williams, A. Waterman, D. Patterson. "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures." In: *Commun. ACM* (2009).
[25] D. Williamson et al. "A standard test for numerical approximation to the shallow water equations in spherical geometry." In: *Journal of Computational Physics* (1992).
[26] J. Yu, J. S. Hesthaven. "A comparative study of shock capturing models for the discontinuous Galerkin method." In: *EPFL-Article* 231188 (2017), pp. 1–42.