Politecnico di Milano, Swiss National Supercomputing Centre

September 1st, 2018

Implementation and evaluation of Discontinuous Galerkin methods using Galerkin4GridTools

Project for the course: Advanced Programming for Scientific Computing, Mathematical Engineering MSc., Politecnico di Milano

Author: Niccolò Discacciati (Politecnico di Milano)

Mentors: William B. Sawyer (ETH/CSCS), Christopher Bignamini (ETH/CSCS), Luca Bonaventura (Politecnico di Milano)

1 Introduction

Over the last decades, in research areas linked to partial differential equations (PDEs), the interest for hyperbolic problems has largely grown [1]. Indeed, they find applications in several fields, such as molecular dynamics, atmospheric phenomena or magnetohydrodynamics. It is well known that smooth initial data can lead to discontinuous solutions, so that the presence of shocks or contact waves may create additional complexities.
In the development of the associated numerical algorithms, multiple aspects have to be considered [3, 2]. First, one would like to deal with schemes which could be applied to complex geometries, like an airfoil. This flexibility is becoming more and more important in concrete applications, like computational fluid dynamics. Second, high order methods are needed, especially when multi-dimensional problems, non-smooth solutions or long temporal intervals are present. Third, peculiarities of hyperbolic equations are, for instance, *anisotropy* (i.e. the presence of preferential spatial directions), and conservation of physical quantities, like mass, momentum or energy. Such properties have to be somehow taken into account by the numerical scheme. Finally, algorithms exploiting spatial locality guarantee a good parallelization potential. When dealing with multidimensional problems having large number of unknowns, an efficiently parallelized code is able to reduce the computational time, without losing in terms of accuracy of the numerical solution. In some cases, like weather forecasting, parallelization becomes compulsory in order to get concrete results in a reasonable time.
From one side, the classical schemes like Finite Differences, Finite Volumes or Finite Elements fail in at least one of the above-mentioned aspects. On the other hand, the Discontinuous Galerkin (DG) solvers have all the desired properties.
In this project we develop a C++ version of a DG solver for conservation laws. Even though the code supports a generic flux function, the focus is given to two main problems, namely the linear advection equation and the Shallow Water Equations (SWEs). The code is completed by adding the support for a variable discretization degree in the mesh, usually called *adaptivity*, while the ultimate goal of the project would be the extension to a spherical geometry. The code supports only *static adaptivity*, meaning that the discretization degrees, which are by definition space-dependent, do not change in time. A *dynamic adaptivity*, i.e. a variable-in-time discretization degree, goes beyond the goal of this project. However, a simple, but not fully validated, algorithm is provided. For our (future) purposes, adaptivity is mainly needed in order to deal with the poles of a sphere, which are singular points of the sphere itself. Indeed, close to the Equator the discretization degree could be reasonably high, leading to a good (and smooth) approximation of the exact solution. However, a lower degree is needed close to the poles, in order to avoid problems arising from the Courant–Friedrichs–Lewy (CFL) condition. Roughly speaking, at high latitudes the grid spacing reduces, so that a low degree is needed in order not to be forced to pick a very low time step. More generally, a variable discretization degree represents a key strategy in order to save computational resources. Indeed, the degree varies based on certain solution features, and low degrees might be sufficient when the solution does not exhibit fine structures. The main application of the code is a numerical analysis of atmospheric dynamics on the Earth. Indeed, in such a context, the SWEs are viewed as a preliminary benchmark case to investigate such dynamics.
The code is mainly built using two libraries:

- GridTools (GT), 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, a C++ Trilinos library. It provides the numerical support for finite element discretizations. Epetra, another Trilinos library, is partly employed.

The Galerkin4GridTools (G4GT) framework provides the link between these libraries, adding a higher-level layer to GridTools and the support to solve PDEs with finite elements. From the user's point of view, this is undoubtedly an advantage, since he does not necessarily have to deal with the innermost part of GT in order to implement FE schemes.
The rest of this report is structured as follows. In Section 2 we provide the details about 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. Some concluding remarks are made in Section 6.

2 Numerical discretization

2.1 The problem

Let $\Omega = [a,b] \times [c,d] \subset \mathbb{R}^D$ $(D = 2)$ be a rectangular domain described by the spatial coordinates $\boldsymbol{x} = (x,y)$ and $T > 0$ be a fixed time instant. Consider a generic conservation law, expressed as

$$\frac{\partial \boldsymbol{u}}{\partial t} + \nabla \cdot \boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{s}(\boldsymbol{u}),$$

where $\boldsymbol{u} : (\Omega \times [0,T]) \to \mathbb{R}^n$ is the vector of conserved variables, $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^{n \times D}$ is the so-called *flux function* and $\boldsymbol{s} : (\Omega \times [0,T]) \to \mathbb{R}^n$ is a generally non-zero source term. We emphasize the fact that $\boldsymbol{f} = \boldsymbol{f}(\boldsymbol{u})$, with a possibly nonlinear dependence on the conserved variable. For the sake of simplicity, we omit the dependence of $\boldsymbol{f}$ on space and time. The problem is completed with *periodic* boundary conditions. Therefore, we impose that the solution satisfies

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

at a continuous level. A periodic initial condition is also given, denoted by $\boldsymbol{u}_0 = \boldsymbol{u}_0(\boldsymbol{x})$.
Here, the superscript n denotes the number of equations we are solving or, equivalently, the number of conserved variables. Unless stated otherwise, we turn our attention to the scalar case $n = 1$, leaving the extension to systems of conservation laws to the dedicated Subsection.

2.2 Spatial discretization

As mentioned in Section 1, the spatial discretization is performed using a Discontinuous Galerkin scheme [3, 4].
Let us start by splitting the domain into K elements, denoted by D^k, for any $k = 1 \dots K$. In view of the GridTools implementation, we consider a structured grid composed of rectangular elements. In other words, D^k is simply a rectangle, say $[x_l^k, x_r^k] \times [y_b^k, y_t^k]$. For each mesh element, let V_h^k be the finite-dimensional space of multivariate polynomials up to a given degree $r = r(k)$ in each dimension. More precisely,

$$V_h^k = \left\{ v : v = \sum_{i,j=1}^{r} \alpha_{ij} x^i y^j, \quad x \in [x_l^k, x_r^k], y \in [y_b^k, y_t^k] \right\}. \tag{1}$$

Consequently, the finite-dimensional space where 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. In particular,

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

where $u_h^k \in V_h^k$ is the local finite-dimensional solution. By definition, the DG method is purely local, so that most of the reasoning can be carried out within a given mesh element D^k.
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 $\phi_h^k \in V_h^k$. In other words, the residual belongs to the orthogonal complement of the local discrete space (1).
After an integration by parts, the weak DG formulation is

$$\int_{D^k} \frac{\partial u_h^k}{\partial t} \phi_h^k + \int_{\partial D^k} \boldsymbol{f}^*(u_h) \cdot \boldsymbol{n}_k \, \phi_h^k - \int_{D^k} \boldsymbol{f}(u_h^k) \cdot \nabla \phi_h^k = \int_{D^k} s(u_h^k) \phi_h^k. \tag{2}$$

Here, the physical flux at the element boundary is replaced by a suitably defined numerical flux, denoted by $\boldsymbol{f}^*$. As in Finite Volume schemes, this is mandatory in order to avoid the lack of uniqueness of the flux itself at the element boundaries. In other words, we need to guarantee the *consistency* property of the flux itself. Note that all the terms are local to the k-th element, except for the numerical flux, which depends on the neighboring elements. This is the reason why we write $\boldsymbol{f}^*(u_h)$ instead of $\boldsymbol{f}^*(u_h^k)$. Usually, the choice of $\boldsymbol{f}^*$ plays a key role in terms of accuracy, stability and robustness 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. Then,

$$\boldsymbol{f}^*(u_h) = \boldsymbol{f}^*(u_h^k, u_h^{\tilde{k}}) = \frac{\boldsymbol{f}(u_h^k) + \boldsymbol{f}(u_h^{\tilde{k}})}{2} - \frac{\alpha}{2}(u_h^{\tilde{k}} - u_h^k)\boldsymbol{n}_k, \tag{3}$$

where α is a large enough stabilization parameter, usually defined as $\alpha \geq \max |\boldsymbol{f}'(u)|$. The maximum is either taken globally over Ω or locally within the k-th element or, even better, within the e-th edge. The former leads to the so-called *Lax-Friedrichs* flux, while the latter is referred as *Local Lax-Friedrichs* or *Rusanov* flux. In this work we adopt the latter choice, since it turns out to be less dissipative compared to the former. In both scenarios, we can interpret the right-hand-side of equation (3) as a sum of two contributions. The first is an average of the physical flux function between the neighbors, which is also known as *cenetered* flux. The second depends on the difference of the solutions, whose role is to penalize the jump of the solution itself. More generally, the former guarantees optimal accuracy, while the latter is responsible for stability issues.
The next step aims to choose a suitable basis for the discrete space V_h^k. First, observe that its dimension is simply $(r+1)^D$. We can now denote the basis as

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

Therefore, the following local expansion for the solution holds:

$$u_h^k = \sum_{j=1}^{(r+1)^D} u_j^k \phi_j. \tag{4}$$

We also recall that, once a basis is defined, the weak DG formulation (2) holds with ϕ_h^k equal to each of the basis functions. In principle, the test and the basis functions can be different, as well as the space they span, but, at least in standard DG schemes, they are chosen to be equal [5]. Another important aspect, common to most of the Finite Element schemes, is the definition of the *reference element*, denoted by I, whose spatial coordinates are referred as $\boldsymbol{s} = (s,t)$ in the case $D = 2$. Throughout this work, we select I equal to the square $[-1,1] \times [-1,1]$. Essentially, a significant part of the complexity is shifted to I, making most of the computations easier and faster. Then, a mapping Ψ from the reference to each of the physical elements is identified. Due to this map, in order to build the main Finite Element tools we can focus on I only. 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 ϕ_j are common in the literature. The first one relies on a *nodal* basis, based on the identification of a set of distinct points. The basis is then defined using the Lagrange interpolating polynomials. 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. In order to properly define them, we start with their one-dimensional version. Different equivalent definitions are known, one of them being 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 defining

$$\begin{aligned} P_0(s) &= 1, \\ P_1(s) &= s. \end{aligned}$$

At this point, it is worth recalling that we focus on rectangular elements only. Therefore an extension of the definition of Legendre polynomials to a two-dimensional context is simply carried out performing a tensor product. Thus,

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

where j is linked to (l,m) as

$$j = l(r+1) + m. \tag{6}$$

The relation (6) is mainly required to reference the basis functions using a single index. Other ordering strategies can be considered, too.
The choice of a modal basis is rather popular in (explicit) DG schemes. Indeed, different advantages are present. First, an orthogonality condition is satisfied, making the mass matrix diagonal. Focusing again on the one-dimensional scenario, the Legendre polynomials satisfy

$$\int_{[-1,1]} P_i P_j ds = \frac{2}{2i+1}\delta_{ij},$$

and a similar relation holds for the multidimensional scenario due to arguments related to tensor products and separability of the integrals. Second, the energy of the solution depends only on the squares of the coefficients u_j, up to normalization factors. Third, criteria for adaptivity are easier to implement. Indeed, modal bases are also known as *hierarchical* bases. The functions spanning $V_h^k(r)$ constitute a subset of the set of the basis functions spanning $V_h^k(r+N)$ for $N \geq 1$. In simpler words, increasing the discretization degree results in adding more basis functions, without altering the previous ones. Clearly, this argument does not hold in a nodal framework.
Finally, the weak DG form becomes

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

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 $\boldsymbol{u}^k = [u_1^k \dots u_{(r+1)^D}^k]^T$ 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 local contributions, we can formally write the problem as

$$\boldsymbol{M}\frac{d\boldsymbol{u}}{dt} = \boldsymbol{A}(\boldsymbol{u}), \tag{7}$$

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. The matrix $\boldsymbol{M}$ has a block diagonal structure, where each of the blocks corresponds to a local mass matrix for a given element. Since a modal basis is used, $\boldsymbol{M}$ is actually a diagonal matrix. 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. We simply observe that, at least for linear problems, it can be represented through a matrix, i.e. $\boldsymbol{A}(\boldsymbol{u}) = \tilde{\boldsymbol{A}}\boldsymbol{u}$. In this scenario, $\tilde{\boldsymbol{A}}$ is a sparse matrix with nonzero diagonal blocks and extra-diagonal terms arising from the boundary integrals. An example of the matrix pattern is provided in [3] for a (second order) Poisson problem.

2.3 Temporal discretization

The temporal discretization is performed using an explicit scheme. The main advantage of such algorithms is that there is no need to implement solvers to find the solution of (possibly nonlinear) systems. However, in order to be stable they require an upper bound on the time step, usually known as CFL condition in the context of hyperbolic problems. It depends on the spatial discretization through the maximum element size h and the polynomial degree r. Since in the library we employ there is no available solver for nonlinear systems, explicit time-integration schemes are almost mandatory. We can write (7) in the standard form of an ordinary differential equation as

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

and then apply the time-advancing scheme. A popular family of time integrators is given by the Runge-Kutta schemes [6]. Let Δt be a suitably defined time step and let u^n, u^{n+1} be the numerical solutions at the current and next iteration respectively. Then, u^{n+1} is approximated as

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

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

where s denotes the number of internal substeps. Here, $a_{ij},\ b_i,\ c_j$ are suitably defined coefficients, which characterize the properties, such as accuracy and stability, of the scheme. In this work we employ three different integrators:

- 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

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

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

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

  As expected this choice results in a third order scheme.

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.
As mentioned earlier in this Subsection, a stability condition has to be taken into account. In other words, the time step Δt has to be small enough to make the time-integration algorithm stable. In this work, we consider

$$\Delta t \leq C \frac{h}{\max |f'(u)|}, \tag{9}$$

where the denominator denotes a global wave speed. In the one-dimensional scalar case, it is simply the maximum absolute value of the derivative of the flux function. In a two-dimensional context, it takes into account both the x and y components of $\boldsymbol{f}(u)$. However, for the sake of simplicity, the same notation is kept for 1D and 2D scenarios. The constant C in (9) is also known as *Courant number* and it depends on the maximum discretization degree r. Usually, it behaves as

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

forcing the time step to be very low for high discretization degrees.
As a side remark, we note that the choice of the characteristic grid spacing h is rather arbitrary. Firstly, observe that we can define a characteristic length for each element, say h^k. Then, h is simply the maximum of the local lengths. Thus, the issue lies in the definition of h^k, rather than h. Practically, the former is usually picked as the maximum or minimum distance among the element vertexes. Since in this work we deal with rectangular elements only, a good choice is provided by $h^k = \max(\Delta x^k, \Delta y^k) = \max(x_r^k - x_l^k, y_t^k - y_b^k)$.

2.4 Practical issues

Now, we provide a more detailed description on how to practically compute the more challenging finite-dimensional operators. For simplicity in the explanation, in this Subsection we assume the discretization degree r to be constant in space and time.
Essentially, most of the complexity lies in how to deal with nonlinearities. One standard option is to project the flux function on the space of polynomials, so that an expansion with respect to the basis functions can still be exploited. This is rather typical in the context of nodal Discontinuous Galerkin schemes, where the coefficients are simply given by the flux function values at the nodes [3]. The main advantage of such approach is that most of the operators can be computed in a pre-processing step. However, since we rely on modal expansion, this is not a viable strategy. Therefore, we compute all the required terms using quadrature formulas at each time iteration. Of course, 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 [7].
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

$$\int_{D^k} g(x,y)dxdy = \int_I \hat{g}(s,t)\ |\det J|\ dsdt \simeq \sum_{q=1}^{Q^D} \hat{g}(s_q,t_q)\ |\det J|(s_q,t_q)\ w_q. \tag{10}$$

The quadrature points and weights are denoted by (s_q,t_q) and w_q respectively, while Q^D represents the number of points we employ. The first equality holds since we shift the computation to the reference element, defining $\hat{g}$ as $\hat{g} = g \circ \Psi$ and J as the jacobian matrix of the mapping Ψ. In principle, since Ψ is affine, its determinant is constant and can be taken out of the last sum in (10). A similar reasoning holds for the integrals on the element boundaries ∂D^k.
The next step consists in the choice of the quadrature points and weights. Again, we can focus on the one-dimensional scenario, since the extension to higher dimensions is provided by computing tensor products of points and weights. 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 quadrature rule. However, due to implementation issues, we consider the Gauss-Legendre strategy, which does not include the extrema of the interval in the set of quadrature nodes. Practically, in order to compute internal integrals, this is equivalent to the previous one. Actually, it is even better, since two more degrees of freedom can be exploited and a higher exactness degree is present. However, the drawback lies in the computation of the boundary integrals. Here, a different set of points has to

be considered, since the boundaries are not part of the Gauss-Legendre 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 $q = i + (Q+1)j$ or in an equivalent fashion. On the other hand, for the boundary integrals we consider

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

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

2.5 Adaptivity

It is worth recalling that so far no strong hypothesis has been made on the discretization degree. It has assumed to be constant in the previous Subsection, but this requirement can be easily relaxed, as we show in this Subsection.
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, such adaptivity is implemented in a straightforward way. Dealing with the internal integrals, no issues arise. Indeed, all the 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 theoretically 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. Note that both have to satisfy $Q_i \geq r^i + 1$ in order to exacly integrate polynomials having degree equal to $2r^i$. 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 the largest 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 [8] for further details. In this work, this is not fully validated, but we believe that this does not constitute a crucial problem. We point out that some computations which are carried out in the pre-processing step might not be valid, in the sense that some operators might vary in the temporal loop. A clear example is given by the local mass matrices, whose size depends on the discretization degrees. We define the total (normalized) energy of the solution as

$$\epsilon_{tot} = \sum_{j=1}^{(r+1)^D} |u_j|^2,$$

which coincides with the spatial L^2-norm provided that normalized Legendre polynomials are employed. Similarly, we define the energy contained in the highest modes as

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

where $\mathcal{R}$ is the set of indexes j such that there exist two integers (l, m) such that $j = l(r+1) + m$ and either l or m is greater or equal to r. In simpler words, we take into account the energy of the modes having an x or y degree at least equal to r. 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 $p+1$.

2.6 Extension to systems of equations

So far we focused on the scalar case, i.e. when the unknown variable u is a scalar quantity. In principle, extending the previous framework to systems is quite straightforward. Roughly speaking, considering separately each equation, the same reasoning holds. Thus, the local solution u_h^k belongs to the space $(V_h^k)^n$, being n the number of equations. We make a simple remark concerning the physical flux, and each of its component in particular. In the scalar case, they are scalar quantities, assembled in D-dimensional vectors. Thus, there is no ambiguity in defining the derivative of each component, so that the notation $f'(u)$ makes perfect sense. Dealing with systems, $\boldsymbol{f}'(\boldsymbol{u})$ turns out to be a matrix, namely the jacobian matrix of the flux function with respect to its argument. Similarly, $\max |f'(u)|$ is given by the maximum absolute eigenvalue of the jacobian matrix, representing a maximum wave speed. In this work, we adopt the same notation for scalar problems and systems, unless explicitly required.

2.7 Test cases

We conclude this Section by recalling the main problems and test cases we focus on. We observe that extending the previous framework to more general problems is very simple. Roughly speaking, only the flux function has to be changed. In this work we consider two main conservation laws.
The first case is a simple linear advection problem. It represents a prototype for scalar problems and it might be used as a simple benchmark case for code validation:

$$\frac{\partial u}{\partial t} + \nabla \cdot (\boldsymbol{\beta} u) = \frac{\partial u}{\partial t} + \frac{\partial(\beta^x u)}{\partial x} + \frac{\partial(\beta^y u)}{\partial y} = 0. \tag{11}$$

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 $\boldsymbol{\beta}$.
The second test represents a benchmark case for atmospheric dynamics and numerical weather prediction. It also represents a good test for systems of conservation laws. Such equations are known as *Shallow Water Equations* (SWEs). They can be obtained starting from Navier-Stokes equations, using suitable scaling arguments [9]. As the name suggests, one simply assumes that the horizontal length scale is much greater than the vertical one. Let η 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 $\boldsymbol{v} = (v_x, v_y)$ be the velocity field associated to the flux. The SWEs constitute the equations governing mass and momentum conservation and can be written as

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

$$\frac{\partial (h\boldsymbol{v})}{\partial t} + \nabla \cdot (h\boldsymbol{v} \otimes \boldsymbol{v}) = -gh\nabla\eta - f\boldsymbol{k} \times h\boldsymbol{v}. \tag{12a}$$

A remark has to be done concerning both terms appearing on the right-hand-side of equation (12a). The second one takes into account the rotation of the reference system, and it is known as *Coriolis* force. Its presence becomes mandatory in numerical weather forecasting, in order to take into account the rotation of the Earth. The first term can be rather dangerous to deal with. Recalling that $h = \eta + b$, we can split it as

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

In other words, it is made of two terms. The former can be written in the conservation form, and can be added to the flux at the left-hand-side of equation (12a). The latter is interpreted as a source term and vanishes in presence of a flat topography. Thus, in this scenario, the SWEs can be written in the standard conservation form and no further issues arise. On the contrary, for $b \neq 0$, an additional boundary term has to be added in order to guarantee the *well-balancing* property of the scheme [10]. In this case, it is common practice not to perform the above-mentioned splitting and replace the spatial gradient with a discrete differential operator $\mathcal{D}$ defined as

$$\int_{D^k} gh\ \mathcal{D}\eta\ \phi_h^k = \int_{D^k} gh\ \nabla\eta\ \phi_h^k + \int_{\partial D^k} g(\eta^* - \eta)h^*\boldsymbol{n}\ \phi_h^k,$$

where η^*, h^* are numerical quantities replacing the physical ones. Usually, an average of the latter along a given edge (centered flux) is employed. However, the numerical results we show are obtained with a flat bottom topography, therefore there is no need to introduce the extra boundary term.

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 unload PrgEnv-cray
module load PrgEnv-gnu
module load cudatoolkit
module load Boost
module load cray-trilinos
module load CMake
module load cray-libsci
```

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/1.05.01
```

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’ ./include/stencil-composition/
   structured_grids/iterate_domain_cxx11.hpp
```

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=ON -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 ..
```

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 Galerkin4GritTools (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> -DCMAKE_BUILD_TYPE=RELEASE -DUSE_GPU=ON -
   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 -DBLAS_DIR:PATH=
   $CRAY_LIBSCI_PREFIX_DIR/lib -DLAPACK_DIR:PATH=
   $CRAY_LIBSCI_PREFIX_DIR/lib -DCUDA_TOOLKIT_ROOT_DIR=
   $CRAY_CUDATOOLKIT_DIR -DBLAS_HARDCODED_LIB=$CRAY_LIBSCI_PREFIX_DIR
   /lib/libsci_gnu_51.so -DLAPACK_HARDCODED_LIB=
   $CRAY_LIBSCI_PREFIX_DIR/lib/libsci_gnu_51.so -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

Essentially, the code makes extensive usage of the Galerkin4GridTools library. However, it is worth spending some words to describe the GridTools library, which consistutes 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 designed. Their main drawback lies in the fact that high order schemes are rather difficult to design, since they would require to extend the stencil. 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 other discretization schemes, in particular 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 the following, 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 rather difficult to understand for the final user. Most of the technicalities are therefore 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 worth to be mentioned:

- `./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 concretely perform the computations.
- `./examples/` contains the main files for the implemented test cases. In particular, we will focus on the matrix-vector multiplication example and the DG solver.

To ease the readability of this work, we structure the rest of this Section as follows. Firstly, we describe a simple example, i.e. a matrix-vector multiplication, in order to define the main concepts embedded in the library. The DG solver is described afterwards.

3.3 Matrix-vector multiplication

The goal of this example is to show how to compute a matrix-vector multiplication $\boldsymbol{Ab} = \boldsymbol{out}$. The matrix $\boldsymbol{A}$ is assumed to be block-diagonal, where each block has n_{rows} rows and n_{cols} columns. It can be interpreted as a simplified version of the computation of an internal integral of the DG solver. Indeed, no communication between neighboring elements is present.
We start by defining the problem sizes

```
constexpr uint_t d1(5);
constexpr uint_t d2(5);
constexpr uint_t d3(5);
constexpr uint_t n_rows(10);
constexpr uint_t n_cols(20);
```

where d_1, d_2 and d_3 are the number of mesh elements in each of the three spatial dimensions. We then instantiate the matrix and vectors as follows:

```
//matrix A
using matrix_A_storage_info_t=gdl::storage_info_t< __COUNTER__, 5 >;
using matrix_A_type=gdl::storage_t< matrix_A_storage_info_t >;
matrix_A_storage_info_t m_A_(d1,d2,d3,n_rows,n_cols);
matrix_A_type m_A(m_A_, 0.e0, "m_A");

// vector b
using vector_b_storage_info_t=gdl::storage_info_t< __COUNTER__, 4 >;
using vector_b_type=gdl::storage_t< vector_b_storage_info_t >;
vector_b_storage_info_t v_b_(d1,d2,d3,n_cols);
vector_b_type v_b(v_b_, 0.e0, "v_b");

// vector out
using vector_out_storage_info_t=gdl::storage_info_t< __COUNTER__, 4
   >;
using vector_out_type=gdl::storage_t< vector_out_storage_info_t >;
vector_out_storage_info_t v_out_(d1,d2,d3,n_rows);
vector_out_type v_out(v_out_, 0.e0, "v_out");
```

For our purposes, it is enough to focus on the constructors for `m_A_`, `v_b_` and `v_out_`. The first three indexes refer to the number of mesh elements, while the additional ones define the size of the local operators. For example, for a given mesh element, the matrix $\boldsymbol{A}$ has size 10×20. The other definitions are rather standard, and are simply needed to efficiently store the data. In particular, the storage information type is set by an index named `__COUNTER__` and a second one specifying the dimension of the tensor. In G4GT, there is no need to focus on how data are practically stored. We just recall that, for codes designed on CPUs, the matrix will be unwrapped in a single vector, which is accessed row-wise. Among the three spatial directions, the innermost one is the horizontal axis (d_1), i.e. data belonging to two consecutive elements in the horizontal direction are stored contiguously, after a proper unwrap of the local degrees of freedom. More details can be found in `./numerics/basis_functions.hpp`. After a simple initialization of the storages, the computational domain is set as

```
auto grid=gridtools::grid<axis>({0, 0, 0, d1-1, d1},
{0, 0, 0, d2-1, d2});
grid.value_list[0] = 0;
grid.value_list[1] = d3-1;
```

Let us focus on the horizontal axis, defined by `{0, 0, 0, d1-1, d1}`. The first two indexes refer to the left and right offset. Setting it to zero means that the whole computation is local. The third and fourth ones denote the first and the last mesh element that are considered, before the offset is taken into account. Finally, the fifth index is needed for storage purposes and it is always set equal to d_1. A similar reasoning holds for the vertical dimension, while the z axis is treated differently. Again, the user does not have to take care of how data are treated.
Now, we need to define the set of variables that will be involved in the various steps of the stencil. Basically, this corresponds to declaring the list of storages that will be provided as input/output parameters of our computations.

```
typedef gdl::gt::arg<0,matrix_A_type> p_m_A;
typedef gdl::gt::arg<1,vector_b_type> p_v_b;
typedef gdl::gt::arg<2,vector_out_type> p_v_out;
typedef boost::mpl::vector<p_m_A, p_v_b, p_v_out> accessor_list;
::gridtools::aggregator_type<accessor_list> domain((p_m_A() = m_A),(
   p_v_b() = v_b),(p_v_out() = v_out));
```

Essentially, we define the placeholders `p_m_a`, `p_v_b` and `p_v_out` and a list `accessor_list`, which gathers the placeholders themselves. This type is then provided as template parameter for the instantiation of an `aggregator_type` variable `domain`. Clearly, the order of the placeholders given as template parameters to the list and the one given in the contructor of the `domain` variable must match.
We are now ready to setup our calculation, defined as

```
auto compute=::gridtools::make_computation<gdl::BACKEND>(domain,
grid,
gridtools::make_multistage(gridtools::enumtype::execute<gridtools::
   enumtype::forward>(),
gridtools::make_stage<gdl::functors::matvec>(p_m_A(),p_v_b(),p_v_out
   ())
));
```

The template argument `BACKEND` plays a key role in terms of performances. This is set in the file `./numerics/basis_functions.hpp` and throughout this work the default values are proven to be the most efficient ones. For CPU computations, the architecture is set as the one of the host system, while a blocking strategy is also employed. Beyond the domain information, the key arguments that have to be provided are the `functors` which are responsible of the concrete computation. Since this example is rather simple, we have just one functor. This is given as template parameter to the `make_stage` function, which receives as input arguments the placeholders representing the matrix, the input and the output vector respectively.
The functor is defined as follows

```
struct matvec {

using in1=gt::accessor<0, enumtype::in, gt::extent<> , 5> const;
using in2=gt::accessor<1, enumtype::in, gt::extent<> , 4> const;
using out=gt::accessor<2, enumtype::inout, gt::extent<> , 4> ;
using arg_list=boost::mpl::vector< in1, in2, out > ;

template <typename Evaluation>
GT_FUNCTION
static void Do(Evaluation & eval, x_interval) {
uint_t const cardinality_i=eval.template get_storage_dim<3>(in1());
uint_t const cardinality_j=eval.template get_storage_dim<4>(in1());
for(short_t I=0; I<cardinality_i; I++){
        float_type val=0.;
        for(short_t J=0; J<cardinality_j; J++){
                val += eval(in1(0,0,0,I,J)*in2(0,0,0,J));
        }
        eval(out(0,0,0,I))+=val;
        }
}

};
```

First, the `accessor` types are defined. In our examples we simply have regular accessors, which can be read or modified within the functor. Each of them is defined through four parameters. The first one is simply an index, which ranges from 0 to $N-1$, being N the number of input placeholders. The second one dictates the type of access to the data associated to the accessors. Clearly, the output vector is the only one which has to be modified, thus a read-write access is mandatory. Both the matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$ have to be read only, and they can be declared constant. The third one is the extent, which is set to zero throughout this work. Finally, the last index denotes the dimension of the storage. The most important role is played by the `Do` method, which receives two inputs. Practically, only the first one is significant in the G4GT framework, and it is usually called `eval`. The return statement of the Do method is `void`, and it is usually a *static* method, so it can be called even without creating an object of class `matvec`. The code is rather easy to understand. We just point out that the storages are accessed as one may expect. The first three indexes refer to the spatial element and their value is relative to the current element. In other words, setting them to zero refers to a local computation. The additional indexes are needed to access the local values, once the element is fixed. In our case, we are simply performing a local matrix-vector multiplication as

$$out_i^{loc} = \sum_{j=1}^{n_{cols}} A_{ij}^{loc} b_j^{loc}$$

We remark that once the correct indexes are provided, the concrete value is extracted by the `eval` method, which receives as input the indexed storage.

3.4 Discontinuous Galerkin solver

The previous example did not rely on the *Intrepid* library, since no finite element discretization had to be carried out. Here, the degree of complexity is much higher. Thus, it is worth dividing the description of the code in smaller parts.
Due to the GT and G4GT design, we need to force two hypotheses, 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 scenario. Despite the fact that our advection problem is simply two-dimensional, we embed it into a three-dimensional framework. In other words, rectangles are replaced by cubes and quadrature rules change accordingly. Clearly, a tensor-product argument still holds. To post-process the data, the third dimension is ignored again. The drawback of this approach lies in the code performance, since both memory and computational time is spent to deal with several values in the third dimension, which are practically useless, since they turn out to be equal. However, we recall that GT is designed for three-dimensional problems, so that forcing a two-dimensional example might also slow down the performances. More simply, spending a lot of resources on code refactoring might not worth it. Clearly, we consider a single element in the third dimension, setting $d_3 = 1$ and the 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 Ψ), 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, consider a scenario where degrees from 1 to 4 are considered. Then, the number of quadrature points is at least equal to $5^3 = 125$. When computing integrals of linear polynomials we waste a lot of resources, since 8 points suffice to perform exact integrations. We note that our choice does not have an impact on the CFL condition (9), since it is affected by the discretization degree only.

3.4.1 User-definable parameters

We first focus on the variables the user can arbitrarily modify. First, observe that the program 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. The following three set the number of mesh elements in the three spatial directions. The two last values represent the number of time iterations and the time step Δt respectively. Note that the latter is set a priori and it has to be small enough to satisfy the stability condition for the whole temporal interval. Another viable option would consists in picking an adaptive time step, by estimating the wave speed at each iteration. No significant differences should arise between the two approaches, provided that the scheme is stable. We recall that one should set $e = -1$, $f = 1$ and $d_3 = 1$ in order not to waste more resources than necessary.
Second, a set of parameters have to be provided. Since they are required at compile time, they are declared as constant expressions.

```
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 Ψ from the reference to the physical elements. Dealing with structured meshes, such mapping is affine and a degree equal to one is enough for our goals. 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 $Q = (O + 2)/2$, 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 3. Finally, the linear advection problem or the shallow water equations are set choosing `type=0` or `type=1` respectively. Clearly, other test cases can be implemented.

3.4.2 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 key method of such class is the `compute` one. As the name suggests, it practically computes the values or the gradient of the basis functions in the required set of quadrature points. Then, a similar reasoning holds for the boundary discretization, where only the values of the basis functions are required by the scheme.
A closer look at the implementation of the previous classes has to be done. Indeed, 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.

```
//Compute method
void compute(Intrepid::EOperator const& operator_){
switch (operator_){
case Intrepid::OPERATOR_VALUE : {
        m_phi_at_cub_points_s=std::unique_ptr<
           basis_function_storage_t>(new basis_function_storage_t());
        compute_helper<order_max,order_max,basis_function_storage_t,
           cub>::compute_impl(Intrepid::OPERATOR_VALUE,
           m_phi_at_cub_points_s);
        break;
        }
...
}
}

//The compute_helper class
template <int_t order_loc, uint_t order_max, typename storage,
   typename cub>
struct compute_helper{
static void compute_impl (Intrepid::EOperator const & operator_, std
   ::unique_ptr<storage> & storage_p)

        using fe=reference_element<order_loc, enumtype::Legendre,
           enumtype::Hexa>;
        using discr_t = intrepid::discretization<fe, cub>;
        discr_t fe_;

        switch (operator_){
```

```
        case(Intrepid::OPERATOR_VALUE) : {
                fe_.compute(Intrepid::OPERATOR_VALUE);
                auto val_local_storage = fe_.val();
        }
        }
        ...
        compute_helper<order_loc-1,order_max,storage,cub>::
           compute_impl(operator_, storage_p);
}
};

//The base case
template <uint_t order_max, typename storage, typename cub>
struct compute_helper<-1,order_max,storage,cub>{
static void compute_impl (Intrepid::EOperator const & operator_, std
   ::unique_ptr<storage> & storage_p) {return;}
};
```

The actual computation is carried out using the class `intrepid::discretization`, which relies on the Trilinos library to extract the values. Its full implementation is not reported here, since we want to focus on its main features only. The tensor product is performed by hand, since there is no Intrepid method to get three-dimensional evaluations.

```
//How to get the values
fe::hex_basis().getValues(phi_at_cub_points_i, cub_points_i, Intrepid
   ::OPERATOR_VALUE);

//getValues
evaluate_polynomial< Dim, Order+1 >::apply(storage_, val_per_line_i,
   on_boundary, val_per_line_first_last);

//practical implementation
template <typename Storage>
static void apply(Storage& storage_, Intrepid::FieldContainer<gt::
   float_type> const& val_per_line_, int_t on_boundary_, Intrepid::
   FieldContainer<gt::float_type> const& val_per_line_first_last={}){
switch (on_boundary_){
case 0 : {
uint_t m=0;
for(int_t i=0; i<(Order); ++i)
        for(int_t j=0; j<(Order); ++j)
                for(int_t k=0; k<(Order); ++k) {
                        uint_t q=0;
                        for(int_t qz=0; qz<val_per_line_.dimension(1)
                           ; ++qz)
                        for(int_t qy=0; qy<val_per_line_.dimension(1)
                           ; ++qy)
                        for(int_t qx=0; qx<val_per_line_.dimension(1)
                           ; ++qx){
                        storage_(m,q) = val_per_line_(i, qx)  *
                           al_per_line_(j, qy) * val_per_line_(k, qz)
                           ;
                        ++q;
```

```
                                        }
                          ++m;
}
...
}
```

3.4.3 Assembler

A similar reasoning holds for the `assembler` class. Essentially, it takes into account the mesh structure, providing methods to discretize the geometric mapping Ψ. 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. We do not provide all the details, since it is not the goal of the project and most of the definitions and methods were already provided.

```
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);
```

3.4.4 Initialization

The next step is the initialization of the storages required by the computation. Being similar to the simple matrix-vector multiplication example, we omit it. Instead, we focus on the problem definition, which can be again modified by the final user, if needed. In particular, both the initial condition and the flux function have to be specified. To keep it simple, we rely on *lambda functions*. Moreover, we 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;};
...
}
```

3.4.5 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. First, the initial condition is

specified through 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. Second, 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 (4) and the second holds defining the Vandermonde matrix $\boldsymbol{V}$ as

$$\boldsymbol{V}_{ij} = \phi_j(\boldsymbol{x}_i). \tag{13}$$

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, non-invertible 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 on 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.
The key part of the code is reported hereunder, using the definition of a template class `M2N_multidim`. The method `set_multi_matrix` is required to compute the Vandermonde matrix (13) 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 reported here, since it is never used in an adaptive context. 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();
//Defined as
//void set_multi_matrix() {
//multi_matrix_helper<order_max,num1Dpoints>::set_multi_matrix(
   multi_m2n, multi_n2m, multi_m2n_enr); }

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);
//Defined as
//template <typename Storage1, typename Storage2, typename Storage3>
//void M2N_multi_convert_enriched( 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){...}
```

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.

```
template <uint_t Order>
struct M2N{

private:
//Members
Intrepid::Basis_HGRAD_LINE_Cn_FEM_JACOBI<gt::float_type, Intrepid::
   FieldContainer<gt::float_type> > line_polynomial;
Intrepid::FieldContainer<gt::float_type> points_1d;
...
Epetra_SerialDenseMatrix m2n;
Epetra_SerialDenseMatrix n2m;

public:

//Methods to convert N2M and M2N
template <typename Storage1, typename Storage2>
void M2N_convert( Storage1 const & in, Storage2 & out, uint_t const
   d1, uint_t const d2, uint_t const d3) {...}
template <typename Storage1, typename Storage2>
void N2M_convert( Storage1 const & in, Storage2 & out, uint_t const
   d1, uint_t const d2, uint_t const d3) {...}
template <uint_t num1Dpoints>
Epetra_SerialDenseMatrix get_M2N_matrix_enriched(){...}

//Invert an Epetra matrix
uint_t invert() {
Epetra_SerialDenseSolver Solver;
n2m=m2n;
uint_t flag=Solver.SetMatrix(n2m);
uint_t flag2=Solver.Invert();
return flag+flag2;
}

};
```

3.4.6 The constant-in-time operators

We are ready to begin with the description of the DG solver. Indeed, all the key elements have been computed. So far, all the classes and methods were not meant to be optimized neither in terms of memory nor in terms of computational performances. Indeed, they are never used in the temporal loop, where GT plays the key role. We compute now some tools, i.e. operators, which are constant in time. Here, GT syntax is used. The domain is made of all the 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. In the following steps, it is required to compute the boundary integrals. Being the mesh structured, it turns out to be constant and equal to the ratio between the edge lengths of the physical and the reference element. By definition, the latter is equal to two. The normal outwards unit vectors are computed for each face of the elements. On the other hand, the internal part is rather similar to the boundary ones. First the determinant of the jacobian matrix is computed in all the quadrature points, and it turns out to be equal to the ratio between the areas, i.e. volumes, of the physical and the reference element. The mass matrix is also computed, since in this work only static adaptivity is fully implemented. Thus, we can assume that the mass matrix can be computed a priori and does not vary in time. A similar reasoning holds for the computation of its inverse. By orthogonality, there is no need to implement solvers for linear systems, since the mass matrix is diagonal and the inversion procedure is trivial.

```
auto compute_assembly=gt::make_computation< BACKEND >(domain, 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())

));
```

For the sake of completeness, we report the full implementation of the `mass_extended` functor. Here, the word *extended* refers to the fact that adaptivity is supported by the method, in contrast to the `mass` functor. It is interesting to observe that the class deals with both regular and global accessors. The latter are related to the finite element discretization. Although basis and test functions are equal, GT requires two different global accessors, named `phi_t` and `psi_t`.

```
struct mass_extended {
using jac_det = gt::accessor<0, enumtype::in, gt::extent<> , 4>;
using weights_t = gt::global_accessor<1>;
using phi_t = gt::global_accessor<2>;
using psi_t = gt::global_accessor<3>;
using mass_t = gt::accessor<4, enumtype::inout, gt::extent<> , 5> ;
using order_t = gt::accessor<5, enumtype::in, gt::extent<> , 3>;
using arg_list = boost::mpl::vector<jac_det, weights_t, phi_t, psi_t,
    mass_t, order_t> ;

template <typename Evaluation>
GT_FUNCTION
static void Do(Evaluation & eval, x_interval) {

uint_t const num_cub_points=eval.template get_storage_dim<3>(jac_det
   ());
uint_t const local_order=eval(order_t(0,0,0));
uint_t const local_cardinality=(local_order+1)*(local_order+1)*(
   local_order+1);

weights_t weights;
phi_t phi;
psi_t psi;

for(uint_t P_i=0; P_i<local_cardinality; ++P_i) {
        for(uint_t Q_i=0; Q_i<local_cardinality; ++Q_i) {
                for(uint_t q=0; q<num_cub_points; ++q){
                        eval(mass_t(0,0,0,P_i,Q_i))  += eval(phi(
                           local_order,P_i,q,0)*(psi(local_order,Q_i,
                           q,0))*jac_det(0,0,0,q)*weights(q,0,0));
                }
        }
}

}
};
```

3.4.7 Temporal loop: common part

From here on, all the steps are repeated at each time iteration. Therefore, their implementation has to be efficient in terms of memory, in particular cache-friendly, and performance, avoiding useless steps. The first part is again common to all the 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 to compute the nodal values of the solution and the flux function. Indeed, we recall that integrals are computed using quadrature rules, which are based on nodal values.

```
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_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())

));
```

3.4.8 Temporal loop: Rusanov fluxes

This second part is clearly the most critical one, since communication between neighboring elements is present. Periodic boundary conditions are also applied. In principle, no big differences are present among internal or boundary mesh elements. However, consider the following example. Pick an element belonging to the left boundary, i.e. having one of its edges lying on the left boundary of the domain. It has to access to the data contained in the corresponding element belonging to the right boundary, so that the offset would be equal to the number of elements. On the other hand, picking a generic internal element, only data of neighboring elements have to be accessed. Moreover, due to implicit parallelism in GT, each element is treated independently and there is no way to extract information related to the *position* of the current element.
Therefore, a different treatment has to be considered for internal and boundary elements, which practically differ by the computational grid they work with. Let us focus on the computation of the left-right fluxes. Clearly, the treatment of the top-bottom exchanges is equivalent, while there is no need to perform exchanges in the z direction, since it would result in wasting computational resources only. This can be interpreted as a Neumann condition along the third axis.
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, MPI is required and G4GT has never been tested using MPI itself, so that we choose to keep the implementation less generic, having at the same time all the tools under control.
The internal elements are treated as follows. Since horizontal fluxes are considered, in this context the word *internal* refers to all the elements in the y direction, excluding only the ones 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::grid<axis>({1u, 1u, 1u, (uint_t)d1-2u, (uint_t)d1
   },{0u, 0u, 0u, (uint_t)d2-1u, (uint_t)d2});
coords_lr.value_list[0] = 0;
coords_lr.value_list[1] = d3-1;

auto fluxes_lr=gt::make_computation< BACKEND >(
domain_iteration, 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 to 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.

```
struct Rusanov_lr{
```

```
...

template <typename Evaluation>
GT_FUNCTION
static void Do(Evaluation & eval, x_interval) {

uint_t const num_cub_points=eval.template get_storage_dim<3>(u());
uint_t const beta_dim=eval.template get_storage_dim<4>(fun());

//loop over the faces in left and right directions
for (uint_t face_ : {1,3}) {

//Offset
short_t opposite_i = (short_t)(face_==1)?1:(face_==3)?-1:0;

//Opposite face
short_t face_opposite = (short_t)(face_==1)?3:1;

//Stabilization parameter (for simplicity it is constant in space,
   leading to the Lax-Friedrichs flux)
float_type coeff = -eval(alpha());

for (short_t qp=0; qp<num_cub_points; qp++) {

        float_type inner_prod1=0.;
        float_type inner_prod2=0.;

        for (uint_t dim=0; dim<beta_dim; dim++){
                inner_prod1+=eval(fun(0,0,0,qp,dim,face_)*normals
                   (0,0,0,qp,dim,face_));
                inner_prod2+=eval(fun(opposite_i,0,0,qp,dim,
                   face_opposite)*normals(0,0,0,qp,dim,face_));
        }

        eval(out(0,0,0,qp,face_))=(inner_prod1+inner_prod2)/2.-
        eval(coeff*(u(0,0,0,qp,face_)-u(opposite_i,0,0,qp,
           face_opposite))/2.);
}
}

}
};
```

As mentioned before, the boundary elements are treated in a slightly different fashion. The main change lies in the computational grid. Concerning the left boundary, 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::grid<axis>({0u, (uint_t)d1-1u, 0u, 0u, (
   uint_t)d1},{0u, 0u, 0u, (uint_t)d2-1u, (uint_t)d2});
coords_bd_left.value_list[0] = 0;
coords_bd_left.value_list[1] = d3-1;

auto compute_bc_left=gt::make_computation< BACKEND >(
domain_iteration, 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())

));

auto coords_bd_right=gt::grid<axis>({(uint_t)d1-1u, 0u, (uint_t)d1-1u
   , (uint_t)d1-1u, (uint_t)d1},{0u, 0u, 0u, (uint_t)d2-1u, (uint_t)
   d2});
coords_bd_right.value_list[0] = 0;
coords_bd_right.value_list[1] = d3-1;

auto compute_bc_right=gt::make_computation< BACKEND >(
domain_iteration, coords_bd_right, gt::make_multistage (
execute<forward>()

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

));
```

3.4.9 Temporal loop: main computation

In this last block all the previous terms are assembled together, computing the internal and boundary integrals, and advancing in time. No critical issues are present here, since the computational grid consists of all the 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_U0(), it::p_U1(),
   it::p_U2(), it::p_result2(), it::p_dT(),  it::p_u_new(), it::
   p_order(), it::p_currentRK() )

));
```

3.4.10 Output

Finally, we would like to stress on the output of the program. Two main 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 the 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

4.1 A smooth problem

The first test we propose is mainly needed to validate the implementation, as well as the accuracy and stability properties of the scheme. This is a rather standard procedure for numerical solvers for differential problems. 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 a single period of a two-dimensional sine wave, i.e.

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

By following the characteristic method, one easily finds that the exact solution is given by

$$u(x, y, t) = u_0(x - \beta^x t, y - \beta^y t) = \sin(2\pi(x - t))\sin(2\pi(y - t)).$$

To validate the scheme, we introduce a possible way to compute the discretization error. Here, we focus on the spatial $L^2(\Omega)$-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 has an analytical expression, ϵ can be easily computed. Given an element D^k, let $\boldsymbol{u}_h^k,\ \boldsymbol{u}^k$ be the finite dimensional vectors collecting the local degrees of freedom for the numerical and the (projection of the) analytical solution respectively. Therefore,

$$\begin{aligned}\epsilon^2 = \|u_h(\cdot, T) - u(\cdot, T)\|^2_{L^2(\Omega)} &= \sum_{k=1}^{K} \|u_h(\cdot, T) - u(\cdot, T)\|^2_{L^2(D^k)} = \sum_{k=1}^{K} \int_{D^k} \left(u_h^k - u^k\right)^2 \\ &= \sum_{k=1}^{K} \int_{D^k} \sum_{i,j=0}^{N(m)-1} (u_{h,i}^k - u_i^k)\phi_i (u_{h,j}^k - u_j^k)\phi_j = \sum_{k=1}^{K} (\boldsymbol{u}_h^k - \boldsymbol{u}^k)^T \boldsymbol{M}^k (\boldsymbol{u}_h^k - \boldsymbol{u}^k)\end{aligned}$$

where $\boldsymbol{M}^k$ is the local mass matrix. The last equality holds thanks to linearity of the integral operator. Theoretical results for one-dimensional hyperbolic problems [3] suggest that the following optimal estimate holds:

$$\epsilon \le C_1(r) h^{m+1} \left(1 + C_2(r) T\right),$$

provided that u is sufficiently smooth. The same optimal convergence rate is retained even for two-dimensional problems, at least when dealing with structured meshes, as in this work [2]. In order to numerically 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

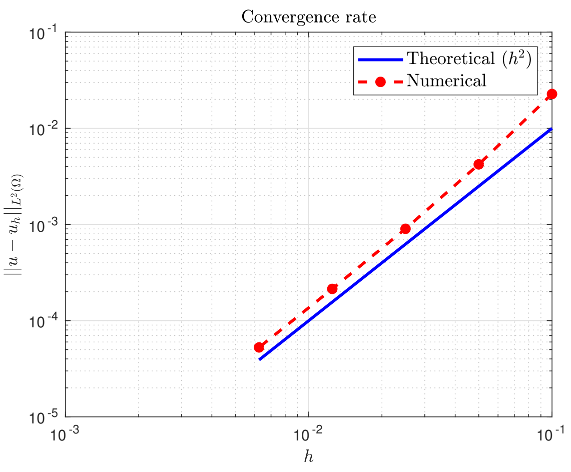
$$p \simeq \frac{\log(\epsilon_1) - \log(\epsilon_2)}{\log(h_1) - \log(h_2)} = \frac{\log(\epsilon_1) - \log(\epsilon_2)}{\log(K_2) - \log(K_1)}.$$

We report the results obtained at time $T = 1$ for different mesh sizes with degrees $r = 1$ and $r = 3$ in Table 1, together with a graphical representation in Figure 1.

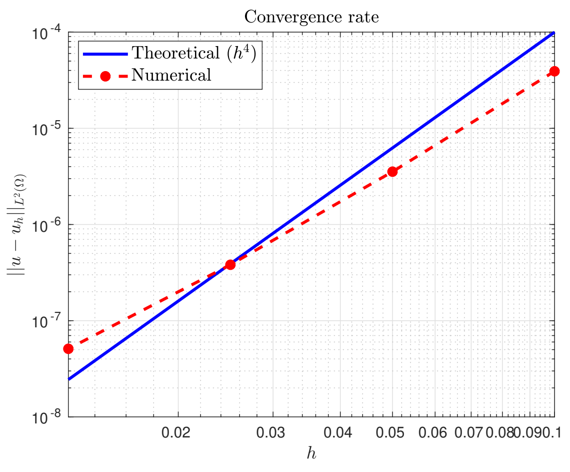
Using linear basis functions, we obtain the optimal convergence rate. The estimated value converges to the optimal one as the mesh is fine enough. On the other hand, a slight deviation from the optimal rate is found for higher orders. This behavior is due to the time integration algorithm. Indeed, due to the CFL condition, one might claim that $\Delta t \sim h$, therefore for spatial orders $r \ge 3$, the global order is bounded by 3, provided that a RK3 scheme is used. To solve this minor issue, one might consider implementing higher order time integration schemes, among which the low-storage RK algorithms [11] constitute a rather popular family. However, we observe that the order of magnitude of the discretization error is in general very low, and the accuracy clearly increases when the basis degree is increased.

K	$r=1$		$r=3$	
	ϵ	p	ϵ	p
10	2.2744e−2	-	3.9185e−5	-
20	4.2232e−3	2.43	3.5420e−6	3.46
40	9.0198e−4	2.23	3.8143e−7	3.21
80	2.1395e−4	2.08	5.1007e−8	2.90
160	5.2724e−5	2.02		

Table 1: L^2 convergence errors and estimated rate with the linear advection problem, $m = 1$ and $m = 4$.



(a) $r = 1$



(b) $r = 3$

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

4.2 Gravity waves

The propagation of pure gravity waves is considered, arising from the shallow water model after a linearization process. For this test case, the Coriolis parameter f is set to zero. Consider the shallow water equations and an equilibrium solution satisfying the equations. A possible option consists of a steady constant fluid height. Assume that this state is perturbed by small quantities, say:

$$h = h_0 + \epsilon h', \quad v_x = 0 + \epsilon v_x', \quad v_y = 0 + \epsilon v_y',$$

where $\epsilon \ll 1$ is a small parameter. Injecting them into the model and neglecting the terms of order greater than 1 in ϵ, we end up with the following linear model:

$$\frac{\partial h'}{\partial t} + \frac{\partial (h_0 v_x')}{\partial x} + \frac{\partial (h_0 v_y')}{\partial y} = 0, \tag{14a}$$

$$\frac{\partial (h_0 v_x')}{\partial t} + g\frac{\partial h_0 h'}{\partial x} = 0, \tag{14b}$$

$$\frac{\partial (h_0 v_y')}{\partial t} + g\frac{\partial h_0 h'}{\partial y} = 0. \tag{14c}$$

Then, differentiating (14a), (14b), (14c) with respect to x, y and t respectively we obtain the following two-dimensional wave equation [12]

$$\frac{\partial^2 h'}{\partial t^2} - gh_0 \, \Delta h' = 0,$$

which describes the propagation of a wave characterized by a speed $c = \sqrt{gh_0}$. The name of this phenomenon originates from the fact that such waves are driven by gravity only.

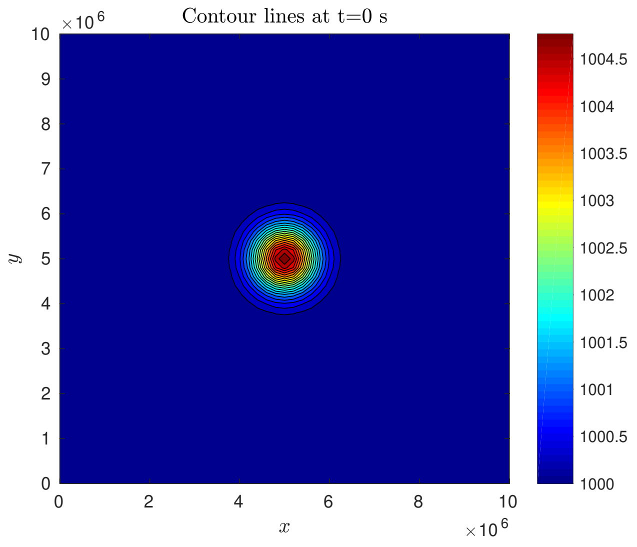
This argument motivates the following test case. Consider a square domain $\Omega = [0, L]^2$ with $L = 10^7$ m. The initial velocities and momenta are set to zero, while the height h is equal to

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

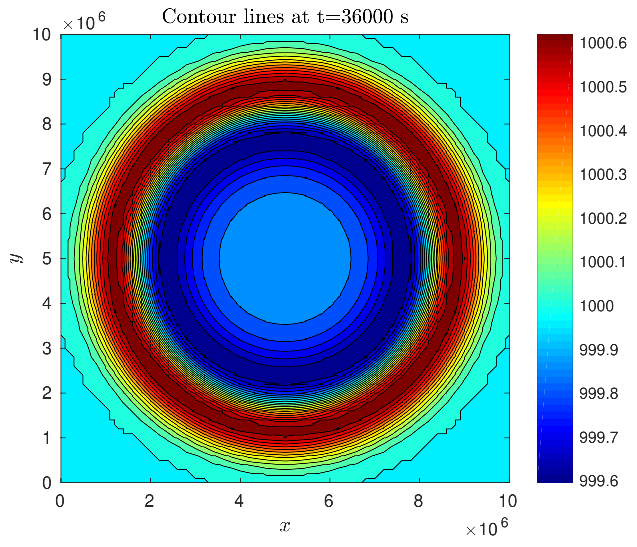
where $h_0 = 1000$ m, $h_1 = 5$ m and $\sigma = L/20$ m. This test case can be included in the previous framework by setting

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

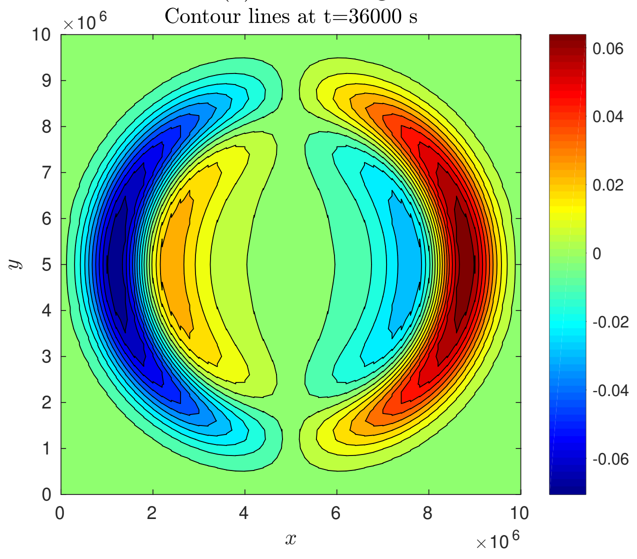
The final simulation time is $T = 36000$ s, so that the wave crest travels for a distance equal to $\Delta r = cT = \sqrt{gh_0}\ T = 3.5656 \cdot 10^6$ m in the radial direction. The simulation has been run using $d_1 \times d_2 = 50 \times 50$ elements, a discretization degree $r = 3$ and a time step $\Delta t = 100$ 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.



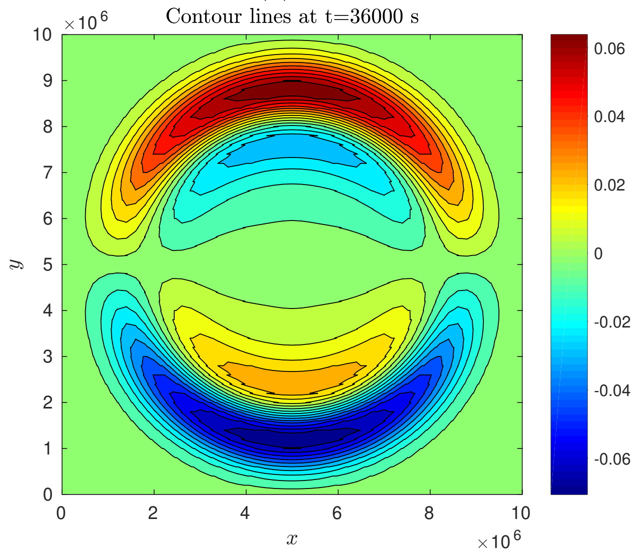
(a) Initial height.



(b) Height.



(c) Horizontal velocity.



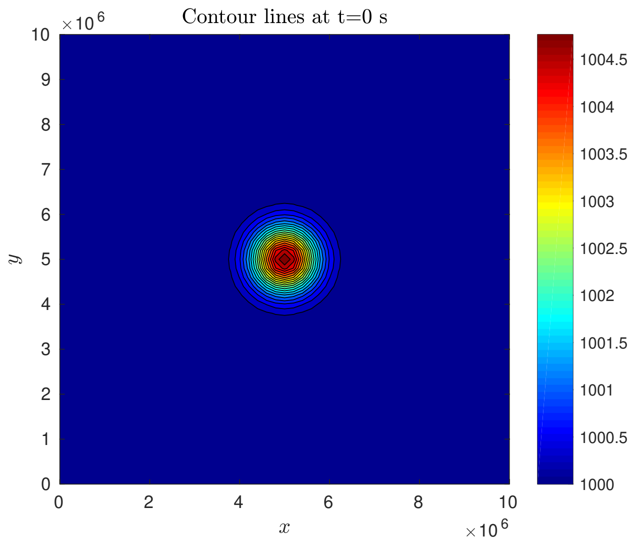
(d) Vertical velocity.

Figure 2: Numerical results for the propagation of pure gravity waves.

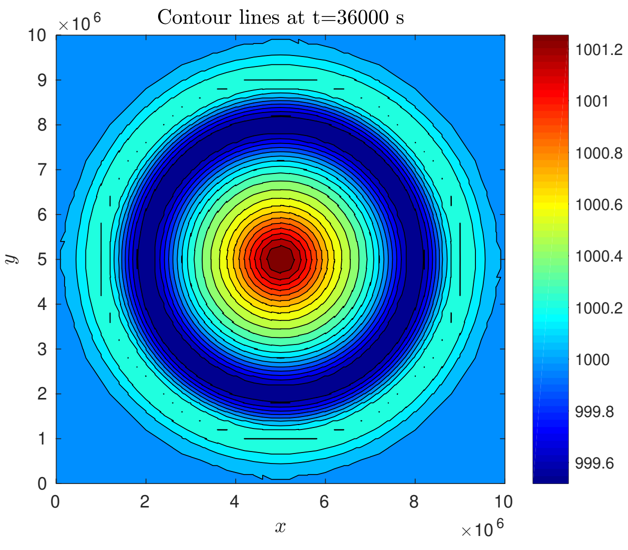
We stress on the fact that the boundary conditions play a marginal role, since the final time is small enough to guarantee that the waves do not exit from the computational domain. We also note some wiggles in the height profile in Figure 2(b). This is due to the way the plot is generated, and does not significantly affect the quality of the solution. Increasing the number of elements and/or the polynomial degree, this visual effect is reduced.

4.3 Geostrophic adjustment

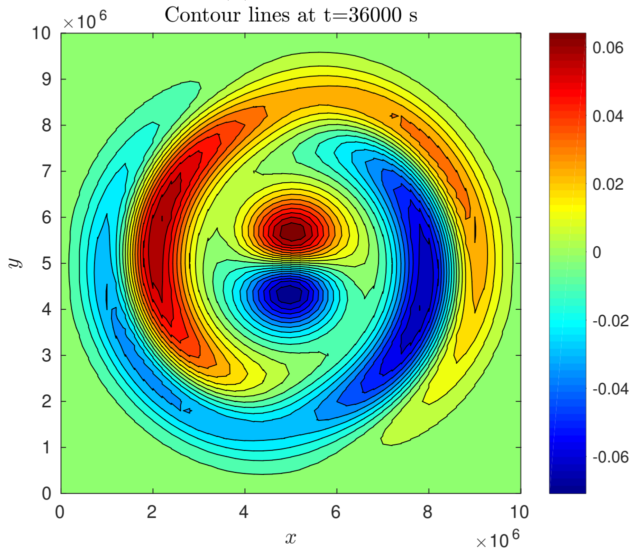
A more challenging test case involves the presence of a nonzero Coriolis force. We want to show that the scheme is able to reproduce the process named *geostrophic adjustment*. In the previous scenario, all the energy was initially concentrated close to the center of the domain. Then, due to gravity, it entirely propagates away from the domain and it is contained in the crests of the wave. On the contrary, here 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. The total number of elements is not varied, and the discretization degree and the time step are not changed, too. 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.



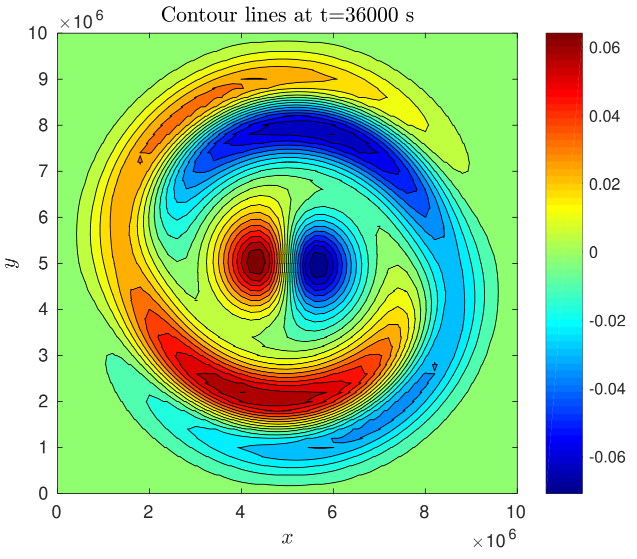
(a) Initial height.



(b) Height.



(c) Horizontal velocity.



(d) Vertical velocity.

Figure 3: Numerical results for the geostrophic adjustment test case.

5 Performance evaluation

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

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

type. These definitions trigger several architecture-based optimizations, like the *layout maps* of GT storages. For instance, in case of a CPU implementation, the dimension with higher stride is the third one, while in a GPU version it is the first one. Other optimizations are present, but from the perspective of a user of the G4GT library, they are at the same time less relevant and more difficult to control.
Concretely, the performance evaluation was performed using the Roofline model [13], which is based on two main variables. The first one is the so-called *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 after the filter of the cache hierarchy. The second index is the number of *attainable Gflops per second*, i.e. the concrete performance measure. These values can be plotted in a two-dimensional graph. 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, an efficient kernel in terms of performances should be close to the determined limit.
In the following, we make a key simplifying assumption, namely we ignore the cache effects. In other words, every access to a variable is taken into account for the computation of the required bytes. This is in contrast with the definition provided by the model, but a precise estimate of the DRAM traffic is far away from being an easy task. In this latter scenario, using some performance analysis tools becomes mandatory. An option relies on the *Intel* compiler, which directly returns the memory traffic after the cache filter. However, this slows down the compilation process (which is already rather slow). Moreover, G4GT has never been tested using the Intel compiler, so that a full validation of the results should be performed first. Again, this represents one of the possible extensions of the project.

5.1 Matrix-vector multiplication

Going back to the Discontinuous Galerkin solver, we observe that in the linear advection problem (11), the operator $\boldsymbol{A}$ in (7) is linear and it can be checked that the fully discretized scheme can be reduced to matrix-vector multiplications, plus standard arithmetic operations with vectors. Thus, a single matrix-vector product can be viewed as a small and simple representative model for the global problem. This is the reason why we perform an evaluation on this kernel before generalizing to the DG code.
Focusing on a square matrix with n rows and columns, it can be verified that a standard implementation of the matrix-vector multiplication involves a number of flops equal to

$$\#\text{flops} = n(n + (n-1) + 1) = 2n^2$$

On the other hand, the memory traffic amounts to

$$\#\text{bytes} = 8(2n+1)n$$

assuming that double precision is exploited and excluding the temporary variables. By definition, the operational intensity is the ratio between the former and the latter quantity. In the following we will consider $n = M^3$, since we want to mimic the DG code, in which a three-dimensional tensor product of the basis functions is used.

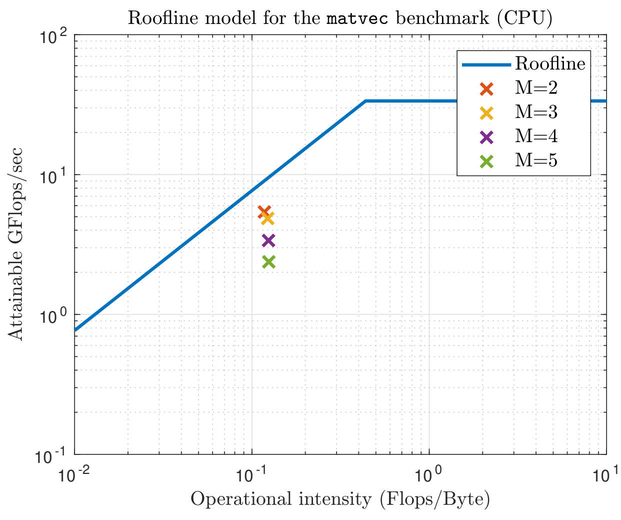
Concretely, we define a local $n \times n$ matrix in all the $d_1 \times d_2 \times 1$ elements of a structured mesh, performing n_{it} times the same operations. Thus, the total number of flops and bytes amount to

$$\#\text{flops} = n_{it} d_1 d_2 (2n) n$$

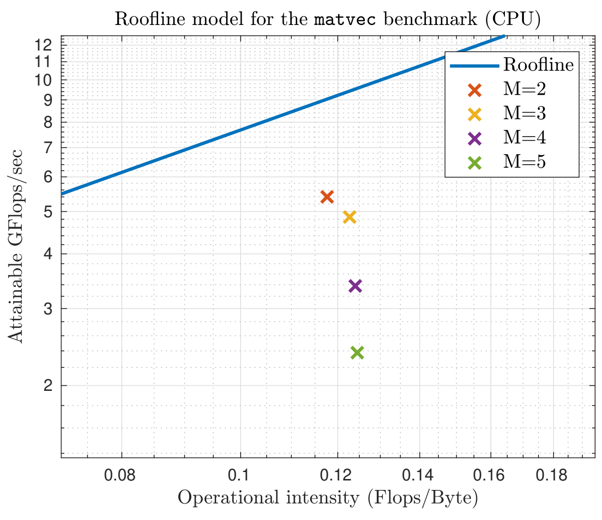
and

$$\#\text{bytes} = 8 n_{it} d_1 d_2 (2n + 1) n$$

respectively, while the operational intensity is not altered. We also focus on the cases $n = M^3$, $M = 2 \dots 5$. In other words, as in the matrix-vector multiplication example we described in Section 3, we consider block-diagonal matrices only. In the DG context, this is equivalent to ignore the boundary contributions, which would require communication between neighbors, resulting in extra-diagonal terms.
In this work we perform tests on a CPU only. A full validation in the GPU framework is left as a further improvement of the project. The main reason lies in the fact that a complete understanding GT syntax in a graphics-processing-unit scenario is not a trivial task. In particular, the behavior using *global accessors* is rather unclear. The simulation was performed on the compute nodes of Piz-Daint at CSCS, using an Intel Xeon E5-2695 v4 processor (single node). The roofline is easily plotted knowing the values of the peak memory bandwidth (76.8 GB/s) and the peak floating point performance (33.6 $Gflops/s$) as in [14]. Moreover, we considered a mesh consisting of $d_1 \times d_2 \times d_3 = 200 \times 200 \times 1$ elements, running $n_{it} = 40$ steps. Figure 4 shows the results.



(a) Normal range



(b) Zoomed range

Figure 4: Performance of the matrix-vector multiplication kernel for the CPU case with different matrix sizes $n = M^3$ and $(d_1, d_2, n_{it}) = (200, 200, 40)$

First, we observe that no significant variations in the operational intensities are present, being all the values reasonably close to the asymptotic value of $2/16 = 0.125$ $flops/bytes$.
For all the considered matrix dimensions, we never obtain performances which are comparable to the roofline model. Being the kernel in the memory-bound range, we expect that the problem is related to a bad access to the data, together with a non-optimal usage of the cache. The performances get worse when the dimension of the problem increases, with a monotonically decreasing behavior with respect to the dimension $n = M^3$. This trend is rather counter-intuitive, since one may expect that the cache effects become more significant for higher matrix dimensions.
One could arrive to the same conclusion also by looking at the computational times. For instance, with $M = 5$ (i.e. a 125×125 local matrix), the required time is approximately 25 s, which is clearly too high for HPC applications.
Several strategies have been exploited in order to try to get better performances, but the results were not so good as expected. Thus, it is possible that some hidden issues are present. One possibility is that the number of operations inside the `matvec` stencil is not large enough in order to fully exploit

all the optimizations. One could define a kernel which performs n_{it} times the same matrix-vector multiplication, inserting this loop inside the functor and not in the main program. We believe that no significant differences appear, but it's worth giving a try. Indeed, its implementation should be rather straightforward. Furthermore, since GT is optimized for three dimensional problems, we expect that setting $d_3 > 1$ the performances improve. However, this is not the case.
Here, we conclude that the drop in terms of performance is explained by the fact that the full range of optimizations available in GT is not exploited. This might be related to certain conflicts which are present when compiling G4GT.

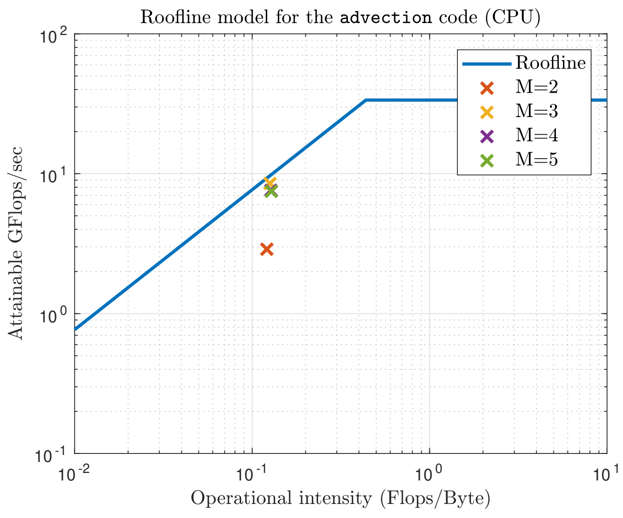
5.2 Discontinuous Galerkin solver

Going back to the original problem, we perform analyses similar to the ones described in the previous Subsection. Two simplifying assumptions are made in this context:

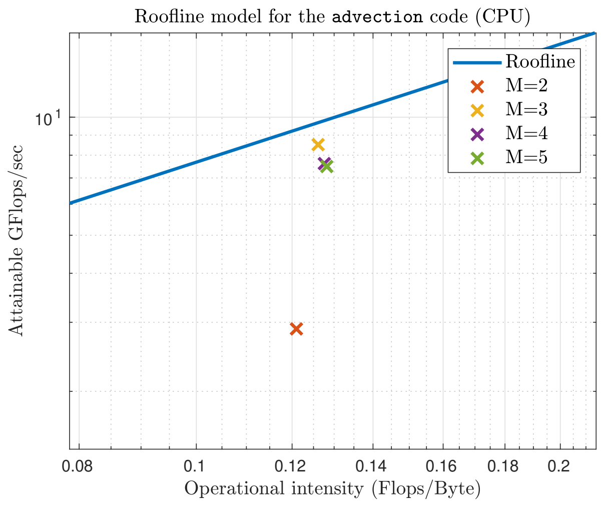
- The problem. We perform the analysis using the linear advection problem only. This might appear a limitation, but we believe that the qualitative results are not altered when switching to systems of conservation laws, i.e. SWEs.
- The time-integration scheme. Here, we consider the Explicit Euler (RK1) scheme only. Clearly, a different polynomial degree is picked to show the differences altering the degree itself. This assumption is dictated by the fact that the cache effects play a role which is more significant using RK2 or RK3. Indeed, we might expect that the temporary vectors which are employed in the scheme are always stored in the cache, so that they should not contribute to the DRAM traffic. However, since we explicitly ignore the cache effects, they play a significant role. In particular, with our assumption, we might expect that the RK2 and RK3 schemes behave similarly to the EE one. Indeed, they can be interpreted as the RK1 algorithm repeated two or three times respectively, at least from a qualitative point of view.

Here, we use $(d_1, d_2, n_{it}) = (200, 200, 40)$. We pick a constant advection field $\beta = (1, 1)^T$, a time step $\Delta t = 0.01\ s$ and the initial condition equal to $u_0(x, y) = 1$. In this context, we choose the number of one-dimensional quadrature points equal to the one-dimensional basis cardinality, both denoted by M. Clearly, we have $M = r + 1$. This choice allows us to simplify the required computations, while performing exact integrations in the scheme. For instance, the mass matrix is computed exactly, since the maximum polynomial degree is $2r = 2M - 2 < 2M - 1$. In the following, we focus only on the time-dependent part of the solver. Indeed, the first computations, such as the jacobian of the geometric mapping or the mass matrix, are never repeated in time. Moreover, if we consider a reasonably large number of steps, the pre-processing phase does not represent the dominant part of the code. Moreover, this part is far away from being fully optimized. Thus, if we took it into consideration, it would invalidate some results. It is also the only step where we make usage of Intrepid and Epetra. Since we have low control on Trilinos optimization, it is rather difficult to fully trigger all of them.
Changing the discretization degree, the operational intensities do not have significant variations. This behavior is similar to the matrix-vector multiplication example. Theoretically, one would expect an increasing intensity with respect to $M = r + 1$, but we recall that all the performances are evaluated using an Explicit Euler scheme. In general, one should use a time-integrating algorithm with an order which depends on the spatial discretization, in order to have comparable orders in space and time. Higher order schemes (e.g. RK2 and RK3) would involve larger number of flops with essentially the same memory traffic (since the temporary vectors would be stored in the cache), leading to an operational intensity which increases with respect to M.
Being the computing environment the same with respect to the `matvec` benchmark, the maximum achievable performance line does not change. The results are shown in Figure 5.

Except for the $M = 2$ case, corresponding to linear basis functions, we observe that the performances are similar when changing the number of degrees of freedom. Moreover, we find that the performances are better when compared to the `matvec` case. This partially contradicts the initial hypothesis which assumed the matrix-vector kernel as a simplified problem. However, since several stages are performed,



(a) Normal range



(b) Zoomed range

Figure 5: Performance evaluation of the DG solver in the CPU case with different number of degrees of freedom and $(d_1, d_2, n_{it}) = (200, 200, 40)$, with constant initial condition.

some memory optimizations can arise. In a single computational kernel we probably do not see such effects, and it is also possible that more operations should be performed in order to trigger a fully optimized code. We also recall that not only matrix-vector multiplications are performed, but we also have operations as vector copies, sums and assignments. They could contribute to the code speedup, too.

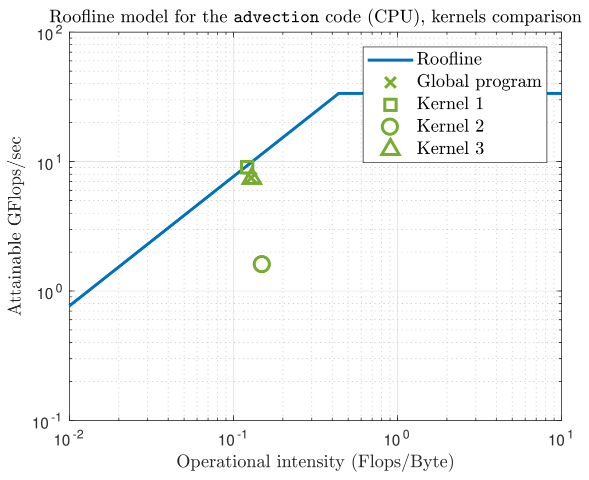
Since multiple stages are involved at each time step, it is quite easy to investigate which one prevails in the computation. Recalling the way the code is structured, we can recognize three different kernels, corresponding to the Subsections we identified in Section 3 while describing the temporal loop.

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

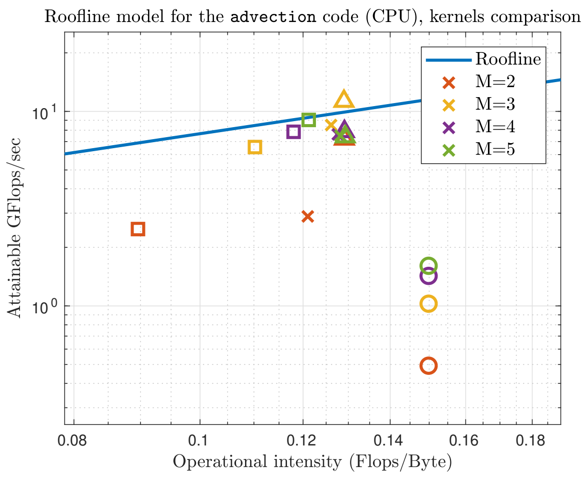
We observe that the third one has the largest influence on the global program in terms of total flops. For large values of M, it gathers the 85% of the total operations. Since all the integrals are performed in it, such a behavior could be expected. In particular, the computation of the internal part has the largest influence, since a three-dimensional integral has to be computed. Therefore, we expect that the performances of the third kernel are similar to the ones of the global program. They are plotted in Figure 6. Excluding again the case $M = 2$, our expectations are confirmed. Moreover, the first kernel has performances comparable to the third one, while the second one gives values which are approximately one order of magnitude inferior with respect to the theoretical line. Since it is the only kernel which involves exchanges between neighboring elements, it is reasonable that cache-misses or non efficient memory accesses are present. Thus, this kernel is the slowest one, in relation with the number of flops.

Finally, one comment has to be done for the linear case $M = 2$. It turns out to be that, even though the third kernel still has the highest percentage of total flops (around 65%), the first kernel has more influence and the performances are always below the others. Since the dimension of the local matrices and vectors are relatively small (8×8 and 8×1 respectively), it could be that some GT optimizations are not fully exploited and the caches do not have a remarkable impact. It is also the only case in which the computational times of the first and second kernels are comparable with respect to the third one.

We conclude the analysis by reporting in Table 2 the measured computational times for the global program and the different kernels. We observe that the computational time for the second kernel is



(a) Performances of the kernels ($M = 5$)



(b) Performances of the kernels (different M)

Figure 6: Performance of the different kernels of the advection problem for the CPU case with different number of degrees of freedom and $(d_1, d_2, n_{it}) = (200, 200, 40)$, with constant initial condition. The crosses denote the performances for the global program, the squares for kernel 1, the circles for kernel 2, the triangles for kernel 3. The colors follow the same convention of Figure 5.

M	Global [s]	Kernel 1 [s]	Kernel 2 [s]	Kernel 3 [s]
2	1.8898	0.391588	0.940605	0.557606
3	5.69921	1.12684	1.01622	3.55614
4	32.3204	4.26993	1.30034	26.7501
5	118.366	12.3155	1.80825	104.242

Table 2: Execution times for the advection problem for the CPU case with $(d_1, d_2, n_{it}) = (200, 200, 40)$.

approximately constant as M is varied, while the other significantly improves. Finally, for $M = 2$ the order of magnitude of the three computational times is essentially the same.

6 Conclusion

In this project we implemented a rather standard Discontinuous Galerkin solver for conservation laws. The main library we exploited is GridTools, with additions provided by Trilinos libraries Intrepid and Epetra. Even though most of the effort was spent on the non-adaptive case, the flexibility in changing the degree was added. Both the implementations give coherent results. We validated the potential of the scheme using a rather simple test case, built using a smooth periodic solution. More challenging examples involve the Shallow Water Equations. In all the cases, the main solution features are captured by the scheme. Examples are the propagation of pure gravity waves or the geostrophic adjustment phenomenon. 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. We found that in a simple matrix-vector multiplication example the performances are below the limits put by the Roofline model. However, the results improve when evaluating the behavior of the DG solver. They are somehow coherent with the Roofline model itself. As a further investigation, performance analysis tools can be used in order to obtain more precise information. Moreover, a scalability analysis could be performed, especially in the adaptive code. However, we believe that this should not represent a major concern.
Multiple extensions to this project can be implemented, from both the numerical point of view and the concrete code improvement. Let us start by the latter, where we would like to give some brief suggestions. 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, 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, at least in the matrix-vector multiplication scenario. Indeed, a further investigation on the obtained results is mandatory. Hopefully, this could be done using some performance analysis tools, in order to relax the assumptions on the cache effects. From the numerical side, the most natural extension of the code is towards the Shallow Water Equations on the sphere, together with a full support for conservation laws on the sphere itself. In this framework, some attempts have already been made. Essentially, the formulation is not drastically altered, except for the fact that, when integrations are performed, a term

$$\mathcal{J} = R^2 \cos\theta$$

appears due to the spherical coordinate system. The variable θ denotes the latitude, while R is the (constant) radius of the Earth. However, the main issue comes from the poles, which are singular points of the sphere. Picking a structured grid in terms of latitude-longitude would result in a very fine mesh close to the poles, where the rectangles would eventually collapse into triangles. This has an effect on the time step, which has to be picked very small in order to be compliant with the CFL condition. Moreover, the mass matrix is no longer diagonal, so that Trilinos libraries have to be exploited once more to perform its inversion.

References

[1] Z. J. Wang et al. *High-order CFD methods: Current status and perspective.* 2013. arXiv: `fld.1 [DOI: 10.1002]`.

[2] J. S. Hesthaven. *Numerical Methods for Conservation Laws: From Analysis to Algorithms.* SIAM Publishing, 2018, p. 555.

[3] J. S. Hesthaven, T. Warburton. *Nodal discontinuous Galerkin methods.* Vol. 54 TS - C. 2008, p. 500.

[4] S. Carcano. "Finite volume methods and Discontinuous Galerkin methods for the numerical modeling of multiphase gas-particle flows." Doctoral Thesis. Politecnico di Milano, 2014.

[5] A. Quarteroni. *Numerical Models for Differential Problems.* Milano: Springer, 2014.

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

[7] A. Quarteroni et al. *Matematica Numerica.* Milano: Springer, 2014.

[8] 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).

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

[10] G. Tumolo, L. Bonaventura. "A semi-implicit, semi-Lagrangian Discontinuous Galerkin framework for adaptive numerical weather prediction." In: *Q.J. Royal Meteorological Society* (2015).

[11] M. H. Carpenter, a. Kennedy. "Fourth-Order Kutta Schemes." In: *Nasa Technical Memorandum* 109112 (1994), pp. 1–26.

[12] S. Salsa. *Partial Differential Equations in Action.* 2008, p. 568. arXiv: `arXiv:1011.1669v3`.

[13] S. Williams, A. Waterman, D. Patterson. "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures." In: *Commun. ACM* (2009).

[14] `https://ark.intel.com/it/products/91316/Intel-Xeon-Processor-E5-2695-v4-45M-Cache-2_10-GHz`. [Online; accessed 04-Nov-2017].

[15] D. Williamson et al. "A standard test for numerical approximation to the shallow water equations in spherical geometry." In: *Journal of Computational Physics* (1992).