QUAIL

Stanford University

An open-source discontinuous Galerkin solver in Python

Eric Ching, Brett Bornhoft, Ali Lasemi, Matthias Ihme

Contents

Chapter 1

Introduction

Quail is an open-source discontinuous Galerkin (DG) code written in Python for numerically solving systems of partial differential equations (PDEs). The objective of this code is to serve as a teaching and prototyping platform for DG methods, with particular focus on code clarity, modularity, and ease of use. Vectorized NumPy operations are extensively utilized to improve computational efficiency. A suite of commonly employed physical models, numerical techniques, and processing tools is available in the code. Currently, Quail solves first- and second-order PDEs on 1D/2D unstructured meshes. This document provides a basic overview of the mathematical formulation and solver features. It will be periodically updated as the code evolves. It provides instructions on how to use the solver and guidelines for software design and future code development. Links to relevant video tutorials are given as well.

See `src/defaultparams.py` for a rundown of the available features/options that can be specified by the user.

Chapter 2

Mathematical formulation

2.1 Spatial discretization

Two discretization schemes are available in Quail: a standard DG discretization and an ADER-DG discretization. This section outlines the derivation of the local semidiscrete form of each scheme. Additional details on the standard DG discretization can be found in Ref. [1]. This section focuses on the discretization for first-order hyperbolic systems. A description of the discretization for second-order systems will be added in the future.

Quail solves linear and nonlinear systems of PDEs of the following form:

$$\partial_t \boldsymbol{U} + \nabla \cdot \boldsymbol{F} = \boldsymbol{S}, \tag{2.1}$$

where $\boldsymbol{U}(\boldsymbol{x}, t) : \mathbb{R}^{N_d} \times \mathbb{R}^+ \rightarrow \mathbb{R}^{N_s}$ is the conservative state vector, $\boldsymbol{F}(\boldsymbol{U}) : \mathbb{R}^{N_s} \rightarrow \mathbb{R}^{N_s \times N_d}$ is the flux, and $\boldsymbol{S}(\boldsymbol{U}) : \mathbb{R}^{N_s} \rightarrow \mathbb{R}^{N_s}$ is the source term. In addition, $\boldsymbol{x} \in \mathbb{R}^{N_d}$ is the spatial coordinate vector, $t \in \mathbb{R}^+$ is the time, N_s is the number of state variables, and N_d is the number of spatial dimensions.

Let Ω denote the computational domain, which is partitioned into N_e non-overlapping discrete elements such that $\Omega = \cup_{e=1}^{N_e} \Omega_e$. $\partial\Omega_e$ is the boundary of element Ω_e. The space of test functions is defined as

$$\mathcal{V}_h^p = \{\phi \in L^2(\Omega), \phi|_{\Omega_e} \in \mathcal{P}_p(\Omega_e) \ \forall \ \Omega_e \in \Omega\}, \tag{2.2}$$

where ϕ is the test function and $\mathcal{P}_p$ denotes the space of polynomial functions of degree no greater than p. The approximation to the global solution, $\boldsymbol{U}_h \simeq \boldsymbol{U}$, can be expanded as

$$\boldsymbol{U}_h = \oplus_{e=1}^{N_e} \boldsymbol{U}_h^e, \tag{2.3}$$

where $\boldsymbol{U}_h^e$ is the local discrete solution,

$$\boldsymbol{U}_h^e(\boldsymbol{x}, t) = \sum_{n=1}^{N_b} \widetilde{\boldsymbol{U}}_n^e(t) \phi_n(\boldsymbol{x}), \tag{2.4}$$

$\widetilde{\boldsymbol{U}}_n^e(t) : \mathbb{R}^+ \rightarrow \mathbb{R}^{N_s}$ is the nth vector of basis coefficients, and $\{\phi_n\}_{\{n=1,\ldots,N_b\}}$ is a basis of $\mathcal{P}_p(\Omega_e)$.

2.1.1 DG

To solve for the local discrete solution, we require $\boldsymbol{U}_h^e$ to satisfy

$$\int_{\Omega_e} \phi_m \partial_t \boldsymbol{U}_h^e d\Omega + \int_{\Omega_e} \phi_m \nabla \cdot \boldsymbol{F}(\boldsymbol{U}_h^e) d\Omega = \int_{\Omega_e} \phi_m \boldsymbol{S}(\boldsymbol{U}_h^e) d\Omega \ \forall \ \phi_m. \tag{2.5}$$

Combining the first term in Eq. (2.5) with Eq. (2.4) allows us to write

$$\int_{\Omega_e} \phi_m \partial_t \boldsymbol{U}_h^e d\Omega = \sum_{n=1}^{N_b} d_t \widetilde{\boldsymbol{U}}_n^e(t) \int_{\Omega_e} \phi_m \phi_n d\Omega = \sum_{n=1}^{N_b} d_t \widetilde{\boldsymbol{U}}_n^e(t) M_{mn}^e, \tag{2.6}$$

where $M^e_{mn} = \int_{\Omega_e} \phi_m \phi_n d\Omega$ represents the element-local mass matrix.

The second integral in Eq. (2.5) is the convective term. Upon performing integration by parts, this term can be expressed as

$$\int_{\Omega_e} \phi_m \nabla \cdot \boldsymbol{F}(\boldsymbol{U}^e_h) d\Omega = -\int_{\Omega_e} \nabla \phi_m \cdot \boldsymbol{F}(\boldsymbol{U}^e_h) d\Omega + \oint_{\partial\Omega_e} \phi_m \widehat{\boldsymbol{F}}(\boldsymbol{U}^{e+}_h, \boldsymbol{U}^{e-}_h, \widehat{\boldsymbol{n}}) d\Gamma, \tag{2.7}$$

where $\widehat{\boldsymbol{n}}$ is the outward-pointing unit normal vector, $(\cdot)^+$ and $(\cdot)^-$ denote interior and exterior information on $\partial\Omega_e$, respectively, and $\widehat{\boldsymbol{F}}$ is the numerical flux (or flux function). The second term on the RHS assumes an element not adjacent to a boundary. Boundary treatment will be discussed in Section 2.6.1.

Combining Eqs. (2.5), (2.6), and (2.7) gives the following:

$$\sum_{n=1}^{N_b} d_t \widetilde{\boldsymbol{U}}^e_n(t) M^e_{mn} = \boldsymbol{R}_m = \int_{\Omega_e} \nabla \phi_m \cdot \boldsymbol{F}(\boldsymbol{U}^e_h) d\Omega - \oint_{\partial\Omega_e} \phi_m \widehat{\boldsymbol{F}}(\boldsymbol{U}^{e+}_h, \boldsymbol{U}^{e-}_h, \widehat{\boldsymbol{n}}) d\Gamma + \int_{\Omega_e} \phi_m \boldsymbol{S}(\boldsymbol{U}^e_h) d\Omega, \tag{2.8}$$

where $\boldsymbol{R}_m$ is the mth residual vector. The time derivative on the LHS is treated using the explicit time stepping schemes described in Section 2.5.1.

2.1.2 ADER-DG

The "**A**rbitrary high-order scheme using **der**ivatives" (ADER) follows the work of Dumbser et al., where we utilize a DG discretization in space and time [2]. This scheme was first introduced in the context of finite volume schemes [2], but was later extended to DG methods with applications to stiff source terms [3]. The method is cast as a predictor-corrector type scheme. First, a predicted space-time solution is found through a weak solution to our system of partial differential equations (PDEs), without considering jump conditions through the interaction between neighboring elements. Then, we utilize the predicted solution vector to advance our solution in time through a DG discretization in space-time.

To solve for the local discrete solution, we multiply the governing PDE system in Eq. (2.1) by the basis and integrate over a space-time control volume $\Omega^{st}_e = \Omega_e \times [t_n, t_{n+1}]$. This leads to

$$\int_{t_n}^{t_{n+1}} \int_{\Omega_e} \phi_m \partial_t \boldsymbol{U}^e_h d\Omega dt + \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \phi_m \nabla \cdot \boldsymbol{F}(\boldsymbol{U}^e_h) d\Omega dt = \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \phi_m \boldsymbol{S}(\boldsymbol{U}^e_h) d\Omega dt \ \forall \ \phi_m. \tag{2.9}$$

For ADER-DG, higher-order accuracy in time is not achieved through multiple stages but rather through an element-local space-time predictor solution.

2.1.2.1 Space-Time Predictor Step

We introduce the element local space-time predictor solution as

$$\boldsymbol{q}^e_h(\boldsymbol{x}, t) = \sum_k \psi_k(\boldsymbol{\xi}, \tau) \widetilde{\boldsymbol{q}}^e_k \tag{2.10}$$

to represent the predictor solution of our space-time element. We note that $\boldsymbol{q}^e_h(\boldsymbol{x}, t^n) = \boldsymbol{U}^e_h(\boldsymbol{x})$. The set of basis functions $\{\psi_k\}_{k=1,\dots,N^{st}_b}$ are constructed using tensor products of the one dimensional basis (here chosen to be Lagrange polynomials). To solve for this local discrete solution, we require $\boldsymbol{q}^e_h$ to satisfy Eq. (2.9). Therefore, our equation governing the predicted solution is as follows

$$\int_{t_n}^{t_{n+1}} \int_{\Omega_e} \psi_k \partial_t \boldsymbol{q}^e_h d\Omega dt + \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \psi_k \nabla \cdot \boldsymbol{F}(\boldsymbol{q}^e_h) d\Omega dt = \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \psi_k \boldsymbol{S}(\boldsymbol{q}^e_h) d\Omega dt \ \forall \ \psi_k. \tag{2.11}$$

At this point, rather than integrating the convective flux term by parts, we instead integrate the first term in Eq. (2.11) by parts. This leads to

$$\begin{aligned} &\int_{\Omega_e} \psi_k \boldsymbol{q}^e_h d\Omega|_{t_{n+1}} - \int_{\Omega_e} \psi_k \boldsymbol{U}^e_h d\Omega|_{t_n} - \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \frac{\partial \psi_k}{\partial t} \boldsymbol{q}^e_h d\Omega dt \\ &\qquad + \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \psi_k \nabla \cdot \boldsymbol{F}(\boldsymbol{q}^e_h) d\Omega dt = \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \psi_k \boldsymbol{S}(\boldsymbol{q}^e_h) d\Omega dt \end{aligned} \tag{2.12}$$

The first two terms in Eq. (2.12) can be thought of as an upwind temporal flux where the solution $\boldsymbol{U}_h^e(\boldsymbol{x}, t_n)$ is known. Next, the polynomial representations of $\boldsymbol{q}_h^e$ and $\boldsymbol{U}_h^e$ are substituted into Eq. (2.12) and we recast our integrals in reference space. We define our reference coordinates in space-time to be $\boldsymbol{\xi}, \tau$ respectively where $-1 \leq \boldsymbol{\xi} \leq 1$ and $-1 \leq \tau \leq 1$ and $\Omega_e^{st} \in \mathbb{R}^{N_d+1}$. Changing to reference space gives us

$$\begin{aligned}\int_{\boldsymbol{\xi}} \psi_k \psi_l d\boldsymbol{\xi} \widetilde{\boldsymbol{q}}_l^e|_{\tau=1} - \int_{\boldsymbol{\xi}} \psi_k \phi_m d\boldsymbol{\xi} \widetilde{\boldsymbol{U}}_m^e|_{\tau=-1} - \int_{-1}^{1} \int_{\boldsymbol{\xi}} \frac{\partial \psi_k}{\partial \tau} \psi_l d\boldsymbol{\xi} d\tau \widetilde{\boldsymbol{q}}_l^e \\ + \frac{\Delta t}{2} \int_{-1}^{1} \int_{\boldsymbol{\xi}} \psi_k \nabla_{\boldsymbol{\xi}} \cdot \boldsymbol{F}(\psi_l \widetilde{\boldsymbol{q}}_l^e) \boldsymbol{J}_e^{-T} d\boldsymbol{\xi} d\tau = \frac{\Delta t}{2} \int_{-1}^{1} \int_{\boldsymbol{\xi}} \psi_k \boldsymbol{S}(\psi_l \widetilde{\boldsymbol{q}}_l^e) d\boldsymbol{\xi} d\tau,\end{aligned} \tag{2.13}$$

where, for brevity, we rely on index notation to imply summing over repeated indices. $\boldsymbol{J}_e$ is the geometric Jacobian for Ω_e, which will be further described in Section 2.2.

We choose to represent the flux and source terms as polynomials with the same form as in Eq. (2.10).

$$\boldsymbol{\mathcal{F}}_h^{*e}(\boldsymbol{\xi}, \tau) = \sum_k \psi_k(\boldsymbol{\xi}, \tau) \boldsymbol{F}_k^{*e} \tag{2.14}$$

$$\boldsymbol{\mathcal{S}}_h^{*e}(\boldsymbol{\xi}, \tau) = \sum_k \psi_k(\boldsymbol{\xi}, \tau) \boldsymbol{S}_k^{*e} \tag{2.15}$$

where $\boldsymbol{F}^* = \boldsymbol{F}\frac{\Delta t}{2}$ and $\boldsymbol{S}^* = \boldsymbol{S}\frac{\Delta t}{2}$. This then leads to the following equation for $\widetilde{\boldsymbol{q}}_h^e$.

$$\begin{aligned}\underbrace{\left(\int_{\boldsymbol{\xi}} \psi_k \psi_l d\boldsymbol{\xi}|_{\tau=1} - \int_{-1}^{1} \int_{\boldsymbol{\xi}} \frac{\partial \psi_k}{\partial \tau} \psi_l d\boldsymbol{\xi} d\tau \right)}_{\boldsymbol{K}} \widetilde{\boldsymbol{q}}_l^e = \underbrace{\int_{\boldsymbol{\xi}} \psi_k \phi_m d\boldsymbol{\xi}|_{\tau=-1}}_{\boldsymbol{F_o}} \widetilde{\boldsymbol{U}}_m^e \\ - \underbrace{\int_{-1}^{1} \int_{\boldsymbol{\xi}} \psi_k \nabla_{\boldsymbol{x}} \psi_l d\boldsymbol{\xi} d\tau}_{\boldsymbol{K}_{\boldsymbol{x}}^e} \cdot \boldsymbol{\mathcal{F}}_h^{*e} + \underbrace{\int_{-1}^{1} \int_{\boldsymbol{\xi}} \psi_k \psi_l d\boldsymbol{\xi} d\tau}_{\boldsymbol{M}^{st}} \boldsymbol{\mathcal{S}}_h^{*e}\end{aligned} \tag{2.16}$$

This is a local non-linear equation for the predicted solution vector in space-time that requires no knowledge of any neighboring elements. We solve this system using a fixed-point Picard iteration approach where

$$\widetilde{\boldsymbol{q}}_l^{n+1} = \boldsymbol{K}^{-1} \left(\boldsymbol{F}_o \boldsymbol{U}_m^e - \boldsymbol{K}_{\boldsymbol{x}}^e \boldsymbol{\mathcal{F}}_l^{*e}(\widetilde{\boldsymbol{q}}_l^n) + \boldsymbol{M}^{st} \boldsymbol{\mathcal{S}}_l^{*e}(\widetilde{\boldsymbol{q}}_l^n) \right) \tag{2.17}$$

In the case of stiff source terms, Eq. (2.17) is not entirely sufficient. To account for the stiffness, we take the source term coefficients implicitly such that

$$\widetilde{\boldsymbol{q}}_l^{n+1} - \boldsymbol{K}^{-1} \boldsymbol{M}^{st} \boldsymbol{\mathcal{S}}_l^{*e}(\widetilde{\boldsymbol{q}}_l^{n+1}) = \boldsymbol{K}^{-1} \left(\boldsymbol{F}_o \widetilde{\boldsymbol{U}}_m^e - \boldsymbol{K}_{\boldsymbol{x}}^e \boldsymbol{\mathcal{F}}_l^{*e}(\widetilde{\boldsymbol{q}}_l^n) \right). \tag{2.18}$$

The source term is then represented using a simple linearization:

$$\boldsymbol{\mathcal{S}}_l^{*e,n+1} \approx \boldsymbol{\mathcal{S}}_l^{*e,n} + \Delta t \left(\widetilde{\boldsymbol{q}}_l^{n+1} - \widetilde{\boldsymbol{q}}_l^n \right) \left(\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{U}} \right) \Big|_{\bar{\boldsymbol{U}}}^{T} \tag{2.19}$$

After substituting Eq. (2.19) into Eq. (2.18) we can cast the iterative approach in the following way

$$\underbrace{\left(\frac{(\boldsymbol{M}^{st})^{-1} \boldsymbol{K}}{\Delta t} \right)}_{\boldsymbol{A}} \widetilde{\boldsymbol{q}}_l^{n+1} + \widetilde{\boldsymbol{q}}_l^{n+1} \underbrace{\left(-\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{U}} \right) \Big|_{\bar{\boldsymbol{U}}}^{T}}_{\boldsymbol{B}} = \underbrace{\frac{1}{\Delta t} \boldsymbol{\mathcal{S}}_l^{*e,n} - \widetilde{\boldsymbol{q}}_l^n \left(\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{U}} \right) \Big|_{\bar{\boldsymbol{U}}}^{T} + \frac{(\boldsymbol{M}^{st})^{-1}}{\Delta t} \left(\boldsymbol{F}_o \widetilde{\boldsymbol{U}}_m^e - \boldsymbol{K}_{\boldsymbol{x}}^e \boldsymbol{\mathcal{F}}_l^{*e}(\widetilde{\boldsymbol{q}}_l^n) \right)}_{\boldsymbol{C}} \tag{2.20}$$

In this form, our local nonlinear solver is known as the Sylvester equation, which we solve iteratively [4].

2.1.2.2 Space-Time Correction Step

The polynomial representation of the space-time solution vector is then substituted into Eq. (2.9) followed by integration by parts of the flux function's term. Therefore, our correction step becomes

$$\begin{aligned}\left(\int_{\Omega_e} \phi_m \phi_k d\Omega\right)\left(\tilde{\boldsymbol{U}}_m^{e,n+1} - \tilde{\boldsymbol{U}}_m^{e,n}\right) - \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \nabla\phi_m \cdot \boldsymbol{F}(\boldsymbol{q}_h^e) d\Omega dt \\ + \int_{t_n}^{t_{n+1}} \oint_{\partial\Omega_e} \phi_m \widehat{\boldsymbol{F}}(\boldsymbol{q}_h^{e+}, \boldsymbol{q}_h^{e-}, \widehat{\boldsymbol{n}}) d\Gamma dt = \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \phi_m \boldsymbol{S}(\boldsymbol{q}_h^e) d\Omega dt\end{aligned} \tag{2.21}$$

Combining Eq. (2.6) and Eq. (2.21) we have:

$$\begin{aligned}\sum_{n=1}^{N_b} d_t \widetilde{\boldsymbol{U}}_n^e(t) M_{mn}^e = \boldsymbol{R}_m = \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \nabla\phi_m \cdot \boldsymbol{F}(\boldsymbol{q}_h^e) d\Omega dt \\ - \int_{t_n}^{t_{n+1}} \oint_{\partial\Omega_e} \phi_m \widehat{\boldsymbol{F}}(\boldsymbol{q}_h^{e+}, \boldsymbol{q}_h^{e-}, \widehat{\boldsymbol{n}}) d\Gamma dt + \int_{t_n}^{t_{n+1}} \int_{\Omega_e} \phi_m \boldsymbol{S}(\boldsymbol{q}_h^e) d\Omega dt\end{aligned} \tag{2.22}$$

Here, the jump conditions between element neighbors are taken into consideration via a numerical flux function.

2.2 Geometric mapping

In order to simplify operations such as interpolation and numerical integration, elements and faces are mapped to reference space. In this section, we provide details on these mappings for 2D elements.

2.2.1 Element mapping

The mapping $\boldsymbol{x} : \widehat{\Omega} \to \Omega_e$, with $\widehat{\Omega}$ denoting the reference element, is given as

$$x(\xi, \eta) = \sum_{i=1}^{N_n} \psi_i(\xi, \eta) x_i^e, \tag{2.23a}$$

$$y(\xi, \eta) = \sum_{i=1}^{N_n} \psi_i(\xi, \eta) y_i^e, \tag{2.23b}$$

where ξ and η are the reference space coordinates, N_n is the number of nodes defining the element geometry, ψ_i is the ith basis function for the geometry interpolation, and (x_i^e, y_i^e) is the physical position of the ith node. Standard Lagrange basis polynomials are chosen for $\{\psi_i\}$. Note that the basis functions and polynomial order for solution and geometry approximation can be different. Figure 2.1 shows the geometric mapping for a $q = 1$ quadrilateral and a $q = 2$ triangle, where q refers to the order of the geometry interpolation. The reference element for quadrilaterals is a square with a side length of two, and that for triangles is an isosceles right triangle with unity leg length. In 1D, the reference element is simply a line segment of length two. The geometric Jacobian is given by

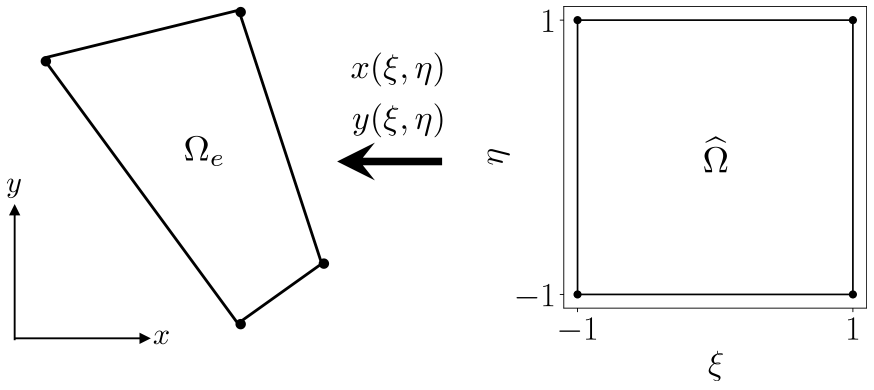
$$\boldsymbol{J}_e = \begin{bmatrix} \dfrac{\partial x(\xi,\eta)}{\partial \xi} & \dfrac{\partial x(\xi,\eta)}{\partial \eta} \\ \dfrac{\partial y(\xi,\eta)}{\partial \xi} & \dfrac{\partial y(\xi,\eta)}{\partial \eta} \end{bmatrix}, \tag{2.24}$$

which can be used to "switch" between reference and physical space when computing gradients. Specifically,

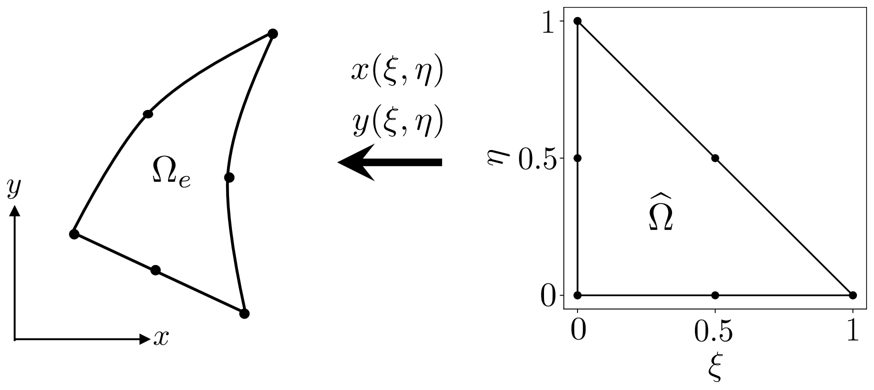
$$\begin{bmatrix} \dfrac{\partial \phi}{\partial \xi} \\ \dfrac{\partial \phi}{\partial \eta} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial x(\xi,\eta)}{\partial \xi} & \dfrac{\partial x(\xi,\eta)}{\partial \eta} \\ \dfrac{\partial y(\xi,\eta)}{\partial \xi} & \dfrac{\partial y(\xi,\eta)}{\partial \eta} \end{bmatrix} \begin{bmatrix} \dfrac{\partial \phi}{\partial x} \\ \dfrac{\partial \phi}{\partial y} \end{bmatrix} = \boldsymbol{J}_e^T \begin{bmatrix} \dfrac{\partial \phi}{\partial x} \\ \dfrac{\partial \phi}{\partial y} \end{bmatrix}, \tag{2.25}$$

and

$$\begin{bmatrix} \dfrac{\partial \phi}{\partial x} \\ \dfrac{\partial \phi}{\partial y} \end{bmatrix} = \boldsymbol{J}_e^{-T} \begin{bmatrix} \dfrac{\partial \phi}{\partial \xi} \\ \dfrac{\partial \phi}{\partial \eta} \end{bmatrix}. \tag{2.26}$$



(a) Geometric mapping for $q = 1$ quadrilateral elements



(b) Geometric mapping for $q = 2$ triangular elements

Figure 2.1: Geometric mapping for (a) $q = 1$ quadrilateral elements and (b) $q = 2$ triangular elements.

2.2.2 Face mapping

Here, we focus on the ith face, Γ_i^e, of the element boundary, $\partial\Omega_e$, where

$$\partial\Omega_e = \bigcup_{i=1}^{N_f} \Gamma_i^e, \tag{2.27}$$

where N_f is the number of faces comprising the element boundary. The mapping $\boldsymbol{x} : \widehat{\Gamma} \to \Gamma_i^e$, with $\widehat{\Gamma}$ denoting the reference face, is given as

$$x(\zeta) = \sum_{j=1}^{q+1} \Psi_j(\zeta) x_j^{e,i}, \tag{2.28a}$$

$$y(\zeta) = \sum_{j=1}^{q+1} \Psi_j(\zeta) y_j^{e,i}, \tag{2.28b}$$

where ζ is the reference coordinate, Ψ_j is the jth basis function for the face geometry interpolation, and $(x_j^{e,i}, y_j^{e,i})$ is the physical position of the jth node of the ith face. 1D Lagrange basis polynomials are chosen for $\{\Psi_j\}$, which are of the same type and order as those for the geometry interpolation of the element (Eq. (2.23)). Figure 2.2 shows the geometric mapping for a $q = 2$ element.

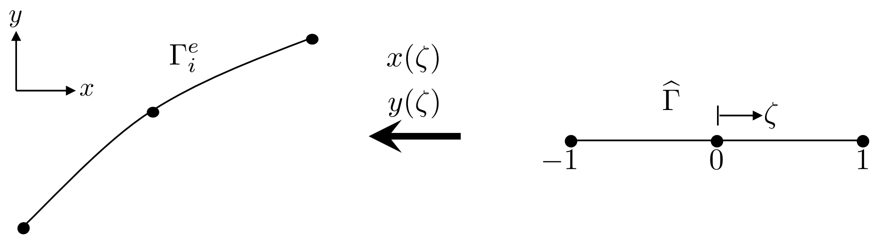


Figure 2.2: Geometric mapping for (a) $q = 1$ quadrilateral elements and (b) $q = 2$ triangular elements.

The outward-facing (unnormalized) normal vector is given as

$$\boldsymbol{n} = \frac{dy}{d\zeta}\widehat{\boldsymbol{e}}_x - \frac{dx}{d\zeta}\widehat{\boldsymbol{e}}_y. \tag{2.29}$$

The unit normal vector can then be computed as

$$\widehat{\boldsymbol{n}} = \frac{\boldsymbol{n}}{|\boldsymbol{n}|}. \tag{2.30}$$

2.3 Numerical integration

The integrals in the previous section are evaluated using numerical quadrature. For details on the available quadrature types, see `general.QuadratureType` in `src` and `src/numerics/quadrature`. To simplify integral evaluation, quadrature is performed in reference space (see Section 2.2). An example of using numerical quadrature to evaluate volume integrals is as follows:

$$\int_{\Omega_e} f(\boldsymbol{x}) d\Omega \approx \sum_{k=1}^{N_q} f(\boldsymbol{x}_k) \mathsf{j}(\boldsymbol{\xi}_k) w_k, \tag{2.31}$$

where N_q is the number of quadrature points, w_k is the kth quadrature weight, and $\boldsymbol{x}_k = \boldsymbol{x}(\boldsymbol{\xi}_k)$ and $\boldsymbol{\xi}_k$ are the coordinates of the kth quadrature point in physical space and reference space, respectively. j is the determinant of the geometric Jacobian. An example of using numerical quadrature to evaluate surface integrals is as follows:

$$\oint_{\Gamma_i^e} g(\boldsymbol{x}) d\Gamma \approx \sum_{k=1}^{N_q} g(\boldsymbol{x}_k) |\boldsymbol{n}(\boldsymbol{\xi}_k)| w_k. \tag{2.32}$$

Note that the number of quadrature points for the surface integral is typically different from that for the volume integral; however, for simplicity, we continue to use N_q to denote the number of quadrature points.

2.4 Matrix Form of the DG discretization

This section recasts the DG spatial discretization from Section 2.1.1 in matrix form, which facilitates code implementation. Here, we use *matrix* as a general term to indicate an array of arbitrary dimensions. For convenience, we adopt the subscript notation employed by the *explicit* mode of the `Numpy` function `einsum`, which is used extensively in `Quail`. This notation is not exactly the same as Einstein notation in that repeated indices do not necessarily denote summation. For example, suppose there are two matrices, $[A]_{ijk}$ and $[B]_{jkl}$. The operation

$$[A]_{ijk}[B]_{jkl} \to [C]_{ikl} \tag{2.33}$$

indicates summation over the j index and entrywise multiplication over the k index.

The first term is the flux volume integral given by

$$\int_{\Omega_e} \nabla\phi_m \cdot \boldsymbol{F}(\boldsymbol{U}_h^e) d\Omega. \tag{2.34}$$

Since ϕ_m is a scalar, then $\nabla\phi_m$ is a vector of size N_d, and its matrix form is indexed as $[\nabla\phi_m]_l$. $\boldsymbol{F}(\boldsymbol{U}_h^e)$ is the flux of each state variable in every direction, so it has shape $N_s \times N_d$. The matrix form of the flux will be referred to as $[F]$ and is indexed as $[F]_{kl}$. The dot product between these two quantities is computed along spatial directions, which yields

$$[\nabla\phi_m]_l[F]_{kl} \rightarrow k. \tag{2.35}$$

If this operation is performed for all basis functions, then the basis gradient matrix now has shape $[\nabla\phi]_{nl}$, so the operation becomes $nl, kl \rightarrow nk$. This operation is performed for every element and at every quadrature point, so evaluating the gradient of the basis at these points gives

$$[\nabla\phi]^e_{ijnl}[F]_{ijkl} \rightarrow ijnk. \tag{2.36}$$

Performing numerical quadrature to evaluate the integral involves multiplying by the Jacobian determinant, j, for each element and quadrature point, as well as the quadrature weight w for the jth quadrature point, then summing over quadrature points. This results in

$$\int_{\Omega_e} \nabla\phi \cdot \boldsymbol{F}(\boldsymbol{U}^e_h)d\Omega \approx [\nabla\phi]^e_{ijnl}[F]_{ijkl}[\mathrm{j}]_{ij}[w]_j \rightarrow ink. \tag{2.37}$$

The next term is the surface flux integral given by

$$\oint_{\partial\Omega_e} \phi\widehat{\boldsymbol{F}}(\boldsymbol{U}^{e+}_h, \boldsymbol{U}^{e-}_h, \widehat{\boldsymbol{n}})d\Gamma. \tag{2.38}$$

Let $[\widehat{F}]$ denote the matrix form of the numerical flux function. This is evaluated at every element and quadrature point and produces a result of size $N_e \times N_q \times N_s$, yielding $[\widehat{F}]_{ijk}$. Note that unlike the analytic flux, $\boldsymbol{F}$, the numerical flux as denoted here is only in the direction normal to the face, so there is no l index for the spatial directions. Additionally, the number of quadrature points for the surface integral is typically different from that for the volume integral; however, for simplicity, we continue to use N_q and j to denote the number of and index for quadrature points, respectively. The basis function matrix has the same shape as its gradient without the dimensional index, i.e. $[\phi]^f_{ijn}$. The integrand then becomes

$$[\phi]^f_{ijn}[\widehat{F}]_{ijk} \rightarrow ijnk \tag{2.39}$$

Performing quadrature, the integrand is multiplied by $[|\boldsymbol{n}|]_j[w]_j$ and summed over the j quadrature points as

$$\oint_{\partial\Omega_e} \phi\widehat{\boldsymbol{F}}(\boldsymbol{U}^{e+}_h, \boldsymbol{U}^{e-}_h, \widehat{\boldsymbol{n}})d\Gamma \approx [\phi]_{ijn}[\widehat{F}]_{ijk}[|\boldsymbol{n}|]_j[w]_j \rightarrow ink \tag{2.40}$$

The final term is the source term integral given by

$$\int_{\Omega_e} \phi_m\boldsymbol{S}(\boldsymbol{U}^e_h)d\Omega \tag{2.41}$$

The matrix form of the source term is represented as $[S]$. With one source term per state variable (k), evaluated at each element (i) and quadrature point (j), this results in $[S]_{ijk}$. As before, the basis function matrix is $[\phi]_{ijn}$, and quadrature is performed by multiplying by $[\mathrm{j}]_j[w]_j$ and summing over the j quadrature points. This yields

$$\int_{\Omega_e} \phi_m\boldsymbol{S}(\boldsymbol{U}^e_h)d\Omega \approx [\phi]^e_{ijn}[S]_{ijk}[\mathrm{j}]_j[w]_j \rightarrow ink \tag{2.42}$$

The final matrix form of the residual (Eq. (2.8)), $[R]_{ink}$, is given by

$$[R]_{ink} = [\nabla\phi]^e_{ijnl}[F]_{ijkl}[\mathrm{j}]_j[w]_j - [\phi]^f_{ijn}[\widehat{F}]_{ijk}[|\boldsymbol{n}|]_j[w]_j + [\phi]^e_{ijn}[S]_{ijk}[\mathrm{j}]_j[w]_j \rightarrow ink. \tag{2.43}$$

A summary of each index is given below:

- i: elements (N_e)
- j: quadrature points (N_q)
- k: state variables (N_s)
- l: spatial directions (N_d)
- n: basis functions (N_b)

V

2.5 Temporal discretization

Quail has two primary approaches for time integration: fully explicit schemes and operator splitting (typically for handling stiff source terms). These are briefly described in the Sections 2.5.1 and 2.5.2 with relevant references. Further details can be found by referencing `general.StepperType` in `src`.

2.5.1 Time stepping schemes

2.5.1.1 Forward Euler

The forward Euler scheme is a standard first-order explicit scheme.

2.5.1.2 Fourth-Order Runge-Kutta Scheme (RK4)

The RK4 scheme is a standard four-stage, fourth-order scheme. An example of this method can be found in the text by Hesthaven and Warburton [1].

2.5.1.3 Low Storage Fourth-Order Runge-Kutta Scheme (LSRK4)

The LSRK4 scheme is a fourth-order RK scheme that minimizes storage requirements. For more details, see Carpenter and Kennedy [5].

2.5.1.4 Strong Stability Preserving Third-Order Runge-Kutta Scheme (SSPRK3)

The SSPRK3 scheme employed here is a total-variation-diminishing, low-storage third-order scheme first presented by Spiteri et al. [6]

2.5.1.5 ADER Space-Time Discretization

The ADER-DG scheme uses a predictor-corrector type method for advancing in time. Details of the ADER-DG scheme are found in section 2.1.2.

2.5.2 Operator splitting

Operator splitting schemes are often used when a stiff source term significantly restricts the time step selection of an explicit scheme. In these methods, we typically use an implicit scheme for the source term. In Quail, we currently have a simple backwards difference scheme (BDF1) and the Trapezoidal method (see `general.SourceStepperType` in `src`).

2.5.2.1 Strang Splitting

This is a common second-order splitting technique first proposed by Strang in 1968 where interchanging the order of your split method leverages the advantage of symmetry in the system [7]. For example, consider the advection-reaction system:

$$\frac{d}{dt}\boldsymbol{U} = \mathcal{T}(\boldsymbol{U}) + \boldsymbol{S}(\boldsymbol{U}) \tag{2.44}$$

where $\mathcal{T} = \nabla \cdot \boldsymbol{F}$ and $\boldsymbol{S}$ is composed of the source term operators.

This system is naturally split by the physics so it is advantageous to use operator splitting. Following the Strang procedure we have the following operations:

$$d_t\boldsymbol{U}^{(1)} = \mathcal{T}(\boldsymbol{U}^{(1)}), \quad \boldsymbol{U}^{(1)}(t_n) = \boldsymbol{U}_n \tag{2.45}$$

$$d_t\boldsymbol{U}^{(2)} = \boldsymbol{S}(\boldsymbol{U}^{(2)}), \quad \boldsymbol{U}^{(2)}(t_n) = \boldsymbol{U}^1(t_n + \Delta t/2) \tag{2.46}$$

$$d_t\boldsymbol{U}^{(3)} = \mathcal{T}(\boldsymbol{U}^{(3)}), \quad \boldsymbol{U}^{(3)}(t_n + \Delta t/2) = \boldsymbol{U}^2(t_n + \Delta t) \tag{2.47}$$

$$\boldsymbol{U}_{n+1} = \boldsymbol{U}^{(3)}(t_n + \Delta t) \tag{2.48}$$

This method is second-order-accurate and strongly stable.

2.5.2.2 Simpler balanced splitting

In simpler balanced splitting, first proposed by Wu et al. [8], stores a vector as:

$$c_n = -\mathcal{T}(\boldsymbol{U}_n) \tag{2.49}$$

From this definition, it is actually possible to reduce the amount of operations due to the first step being in equilibrium. Therefore, the system proceeds as:

$$\begin{aligned} d_t\boldsymbol{U}^{(1)} =& \boldsymbol{S}(\boldsymbol{U}^{(1)}) - c_n, \quad \boldsymbol{U}^{(1)}(t_n) = \boldsymbol{U}_n & (2.50)\\ d_t\boldsymbol{U}^{(2)} =& \mathcal{T}(\boldsymbol{U}^{(2)}) + c_n, \quad \boldsymbol{U}^{(2)}(t_n + \Delta t/2) = \boldsymbol{U}^1(t_n + \Delta t) & (2.51)\\ \boldsymbol{U}_{n+1} =& \boldsymbol{U}^{(2)}(t_n + \Delta t) & (2.52)\end{aligned}$$

2.6 Physics

2.6.1 Boundary conditions

In Quail, boundary conditions (BCs) are treated with, adopting the terminology introduced by Mengaldo et al. [9], either a *weak-prescribed* or *weak-Riemann* approach. In the former, the boundary flux in the surface integral (second term on RHS of Eq. 2.7) is computed using the analytic flux, $\boldsymbol{F}$, instead of the numerical flux. $\boldsymbol{U}_h^b$ denotes the prescribed boundary state used to calculate $\boldsymbol{F}$. The surface integral at a given boundary face, Γ_b, for the convective flux is then written as

$$\boldsymbol{B}^b = \oint_{\Gamma_b} \phi_m \boldsymbol{F}(\boldsymbol{U}_h^b) \cdot \widehat{\boldsymbol{n}} d\Gamma. \tag{2.53}$$

In the *weak-Riemann* approach, the boundary flux is computed using the numerical flux, giving

$$\boldsymbol{B}^b = \int_{\Gamma_b} \phi_m \widehat{\boldsymbol{F}}(\boldsymbol{U}_h^{e+}, \boldsymbol{U}_h^b, \widehat{\boldsymbol{n}}) d\Gamma. \tag{2.54}$$

Two BCs that are applicable to all available equation sets are the `StateAll` and `Extrapolate` BCs. The former employs a *weak-Riemann* approach, with $\boldsymbol{U}_h^b$ fully prescribed by an analytical expression. The latter uses a *weak-prescribed* approach, with $\boldsymbol{U}_h^b = \boldsymbol{U}_h^{e+}$. The `Extrapolate` BC is useful for boundaries with no incoming characteristics (such as supersonic outflow when solving the Euler equations). Other BCs specific to certain equation sets are described below.

2.6.2 Equation sets

In this section, we describe the various equation sets available in Quail. See general.PhysicsType in `src` and `src/physics` for more information. Only the basic equations are provided (without any source terms). For information regarding the available source terms, refer to `functions.SourceType` in each respective physics module.

Scalar advection

The scalar advection equation is given as

$$\frac{\partial u}{\partial t} + \nabla \cdot (cu) = 0, \tag{2.55}$$

where c is the advection velocity (currently assumed constant in Quail).

Scalar advection-diffusion

The scalar advection-diffusion equation is given as

$$\frac{\partial u}{\partial t} + \nabla \cdot (cu) = \alpha \nabla^2 u, \tag{2.56}$$

where c is the advection velocity and α is the diffusion coefficient (both currently assumed constant in Quail).

1D Burgers equation

The 1D inviscid Burgers equation takes the following form:

$$\frac{\partial u}{\partial t} + \nabla \cdot \left(\frac{1}{2}u^2\right) = 0. \tag{2.57}$$

This equation is nonlinear and often used as a model problem for more complicated conservation laws that can involve discontinuities.

Euler equations

The compressible Euler equations models continuum fluid flow without viscous effects. Assuming a calorically perfect gas, the state variables and inviscid flux are written as

$$\boldsymbol{U} = \begin{bmatrix} \rho \\ \rho\boldsymbol{u} \\ \rho E \end{bmatrix}, \quad \boldsymbol{F} = \begin{bmatrix} \rho\boldsymbol{u} \\ \rho\boldsymbol{u} \otimes \boldsymbol{u} + P\mathbb{I} \\ \boldsymbol{u}(\rho E + P) \end{bmatrix}, \tag{2.58}$$

where ρ is the fluid density, $\boldsymbol{u}$ is the velocity vector, P is the pressure, $\mathbb{I}$ is the identity matrix, and $E = R_g/(\gamma - 1)T + \frac{1}{2}|\boldsymbol{u}|^2$ is the total energy per unit mass (with R_g denoting the specific gas constant, γ the specific heat ratio, and T the temperature). Pressure is computed from the ideal gas law, $P = \rho R_g T$.

Two boundary conditions for this equation set available in Quail are the `SlipWall` and `PressureOutlet`. The `SlipWall` BC uses a weak-prescribed approach, with

$$\boldsymbol{U}_h^b = \begin{bmatrix} \rho^+ \\ (\rho u)^+ - \rho^+(\boldsymbol{u}^+ \cdot \widehat{\boldsymbol{n}}^+)\widehat{n}_x^+ \\ (\rho v)^+ - \rho^+(\boldsymbol{u}^+ \cdot \widehat{\boldsymbol{n}}^+)\widehat{n}_y^+ \\ (\rho E)^+ \end{bmatrix}, \tag{2.59}$$

in 2D.

The `PressureOutlet` BC for subsonic outflow uses a weak-prescribed approach, with one input parameter, P^b, the outlet pressure. This BC relies on the outgoing Riemann invariant, $J^+ = \boldsymbol{u}^+ \cdot \widehat{\boldsymbol{n}}^+ + \frac{2a^+}{\gamma-1}$, where a is the speed of sound. The following steps are taken to compute the boundary state:

- Calculate ρ^b by extrapolating the boundary entropy from the interior, i.e. $s^b = s^+ = \ln \frac{P}{\rho^\gamma}\Big|_{\boldsymbol{U}_h^+}$, and solving for ρ^b
- From J^+, $a^b = \sqrt{\frac{\gamma P^b}{\rho^b}}$, and the tangential interior velocity $\boldsymbol{u}_t^+ = \boldsymbol{u}^+ - (\boldsymbol{u}^+ \cdot \widehat{\boldsymbol{n}}^+)\widehat{\boldsymbol{n}}^+$, calculate the boundary velocity as

$$\boldsymbol{u}^b = \left(J^+ - \frac{2a^b}{\gamma - 1}\right)\widehat{\boldsymbol{n}}^+ + \boldsymbol{u}_t^+ \tag{2.60}$$

- Calculate $(\rho\boldsymbol{u})^b$ using ρ^b and $\boldsymbol{u}^b$
- Calculate $(\rho E)^b$ using ρ^b, $\boldsymbol{u}^b$, and P^b

Reactive Euler equations

The compressible reactive Euler equations models continuum fluid flow without viscous effects, but includes a single-step irreversible reaction in the source terms. Assuming a calorically perfect gas, the state variables, inviscid flux, and source term are written as

$$\boldsymbol{U} = \begin{bmatrix} \rho \\ \rho\boldsymbol{u} \\ \rho E \\ \rho Y \end{bmatrix}, \quad \boldsymbol{F} = \begin{bmatrix} \rho\boldsymbol{u} \\ \rho\boldsymbol{u} \otimes \boldsymbol{u} + P\mathbb{I} \\ \boldsymbol{u}(\rho E + P) \\ \rho\boldsymbol{u}Y \end{bmatrix}, \boldsymbol{S} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \dot{\omega} \end{bmatrix}, \tag{2.61}$$

where ρ is the fluid density, $\boldsymbol{u}$ is the velocity vector, P is the pressure, $\mathbb{I}$ is the identity matrix, Y is the species mass fraction, and $\rho E = \frac{P}{\gamma-1} + \frac{1}{2}\rho|\boldsymbol{u}|^2 + \rho q Y$ (with R_g denoting the specific gas constant, γ the specific heat ratio, q the heat release, and T the temperature). Pressure is computed from the ideal gas law, $P = \rho R_g T$.

The reaction rate, $\dot{\omega}$, is defined using the Arrhenius form:

$$\dot{\omega} = -A\rho Y e^{-T_a/T} \tag{2.62}$$

where T_a is the activation temperature and A is a constant for the given reaction mechanism.

Navier-Stokes equations

The compressible Navier-Stokes equations model continuum fluid flow with viscous effects. Assuming a calorically perfect gas, the state variables and flux are written as

$$\boldsymbol{U} = \begin{bmatrix} \rho \\ \rho\boldsymbol{u} \\ \rho E \end{bmatrix}, \quad \boldsymbol{F} = \begin{bmatrix} \rho\boldsymbol{u} \\ \rho\boldsymbol{u}\otimes\boldsymbol{u} + P\mathbb{I} + \boldsymbol{\tau} \\ \boldsymbol{u}(\rho E + P) - \boldsymbol{q} + (\boldsymbol{\tau}\cdot\boldsymbol{u}) \end{bmatrix}, \tag{2.63}$$

where ρ is the fluid density, $\boldsymbol{u}$ is the velocity vector, P is the pressure, $\mathbb{I}$ is the identity matrix, and $E = R_g/(\gamma - 1)T + \frac{1}{2}|\boldsymbol{u}|^2$ is the total energy per unit mass (with R_g denoting the specific gas constant, γ the specific heat ratio, and T the temperature). The viscous stress tensor ($\boldsymbol{\tau}$) and heat flux vector ($\boldsymbol{q}$) are written as

$$\boldsymbol{\tau} = \mu\left(\nabla\boldsymbol{u} + (\nabla\boldsymbol{u})^T - \frac{2}{3}(\nabla\cdot\boldsymbol{u})\mathbb{I}\right), \tag{2.64}$$

$$\boldsymbol{q} = -\kappa\nabla T, \tag{2.65}$$

where μ is the dynamic viscosity and κ is the thermal conductivity. Pressure is computed from the ideal gas law, $P = \rho R_g T$.

2.7 Numerical flux functions

The local Lax-Friedrichs flux function is defined as

$$\widehat{\boldsymbol{F}}(\boldsymbol{U}_h^{e+}, \boldsymbol{U}_h^{e-}, \widehat{\boldsymbol{n}}) = \frac{1}{2}\left[\widehat{\boldsymbol{F}}(\boldsymbol{U}_h^{e+}) + \widehat{\boldsymbol{F}}(\boldsymbol{U}_h^{e-})\right]\cdot\widehat{\boldsymbol{n}} + \frac{\lambda}{2}(\boldsymbol{U}_h^{e+} - \boldsymbol{U}_h^{e-}), \tag{2.66}$$

where λ is the maximum wave speed of the system, i.e. the maximum absolute value of the eigenvalues of the flux Jacobian, $\partial\boldsymbol{F}/\partial\boldsymbol{U}$.

The Roe flux function is available in Quail for the Euler equations. Details on this flux function can be found in Refs. [1] and [10].

For second-order diffusion flux terms a generalized symmetric-interior penalty (SIP) method is used. Details on this method can be found in Ref. [11].

2.8 Troubled Cell Indicators

2.8.1 TVB Modified Minmod Indicator

Quail offers a TVB modified minmod function as a troubled cell indicator [12] where the free parameter M can be utilized to increase the limiting in the solution by increasing the number of troubled elements. See [13] for details of the implementation. We note that we pass the physical state coefficients rather than the characteristic state coefficients into the indicator function. This indicator is located in `src/numerics/limiting/tools.py`.

2.9 Stabilization

Stabilization is critical in DG discretizations that involve sharp discontinuities. In this section, we discuss the stabilization approaches available in Quail. Further details can be found in `src/numerics/limiting` `src/solver/DG.py`.

2.9.1 Positivity-Preserving Limiter (PPL)

The positivity-preserving limiter by Zhang and Shu [14] is available in Quail for the Euler equations. This limiter can guarantee positivity of density and pressure, although spurious oscillations may not be suppressed. Additional details can be found in Ref. [15].

2.9.2 WENO Limiter

A WENO-limiter, **currently only implemented for one dimensional cases**, is available in Quail for all equation sets. This limiter currently requires the use of a troubled cell indicator (or shock indicator). For more details on troubled cell indicator's see Section 2.8.1. The local solution for each troubled element is reconstructed via a convex combination of the solution state of the element and its neighbors. The left and right eigenvectors (as derived in [16] for both reacting and non-reacting Euler equations) of the flux Jacobian are used to transform the solution into and out of the characteristic space. This convex combination occurs in characteristic space using the WENO procedure as defined in [13].

2.9.3 Artificial Viscosity

Chapter 3

Using Quail

3.1 Running the solver

To run Quail, the user should add the `quail` script file to their `$PATH.`From here, assuming an appropriate input deck, just run the driver script with the following syntax:

```
quail <input_deck>
```

The driver has a few additional options. Running `quail-h` results in the following output:

```
usage: quail [-h] [-p POST] [inputdeck]

This script is the driver for Stanford's Quail solver

positional arguments:
inputdeck             this file contains all requested parameters for the
                      solver

optional arguments:
  -h, --help           show this help message and exit
  -p POST, --post POST  post-processing script to execute
```

For more information on running Quail, see the tutorials and example directories.

3.2 Post-processing

Quail's solver and post-processing scripts operate separately. This allows users to run each individually or together. Each post-processing script can be quite unique. Readers are referred to the `example` directory for a variety of cases and options. In a typical Quail setup, the solver will output a pickle file (`*.pkl`) that stores the state of the solver based on a set of instantiated classes. This allows the user to access the most recent state executed by the solver. Given a pickle file and a post-processing script we utilize the following command to execute post-processing:

```
quail -p <post_process_script>
```

In this script, we would unpack the pickle file and can utilize Quail's built-in plotting functions (or third-party plotters) to visualize the data. A user can automatically call the post-processing script at the end of the run-time using the `AutoProcess` flag in the `Output` dictionary (see Sec. 3.3 for more details on input dictionaries). This flag is by default set `True` and will automatically search for a file called `post_process.py` at the end of the solve. For more information on post-processing with Quail, see the tutorials and example directories.

Another feature of Quail is the option to define custom functions in separate files to perform case-specific data processing. These custom functions are run before the time loop and then once per time step. See the following directory for an illustrative example: `examples/scalar/1D/damping_sine_wave`.

3.3 Input deck

Quail input deck's utilize Python dictionaries to pass information into the solver. By using Python dictionaries, input decks can then use other Python features (e.g. `numpy`, `sympy`, ...). The major groups of dictionaries include `Restart`, `TimeStepping`, `Numerics`, `Mesh`, `Physics`, `InitialCondition`, `ExactSolution`, `BoundaryConditions`, `SourceTerms`, and `Output`.

For a detailed list of the available input parameters and appropriate syntax, users are directed to `src/defaultparams.py`. For more detailed information, see the tutorials and example directories.

3.4 Mesh generation

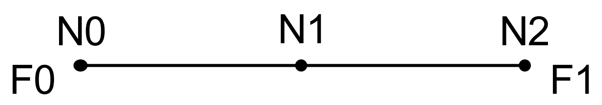
Two options are available in Quail for mesh generation. First, $q = 1$ uniform meshes (q refers to the order of the geometry approximation) in 1D and for 2D rectangular domains can be directly generated using input parameters in the input deck. In 2D, quadrilateral and triangular elements are supported. See the example cases and `src/defaultparams.py` for additional information. The second option is to use Gmsh [17], an open-source high-order mesh generator[1]. Versions 2.2 and 4.1 of MSH ASCII files can be read. Segments, triangles, and quadrilaterals are supported. Currently, only one element type per mesh is permitted. See the example cases and `src/meshing/gmsh.py` for details.

The number of nodes as a function of q for each supported element type is given in Table 3.1. The node positions for a given element are equidistant. The node ordering (N0, N1, etc.) and face ordering (F0, F1, etc.) for $q = 2$ line segments, triangles, and quadrilaterals are given in Figure 3.1. For $q = 1$ elements, only the corner vertices, i.e. principal nodes, are included. The orderings for other polynomial orders follow directly. Note that the pattern for node orderings employed in Quail is different from that in Gmsh.

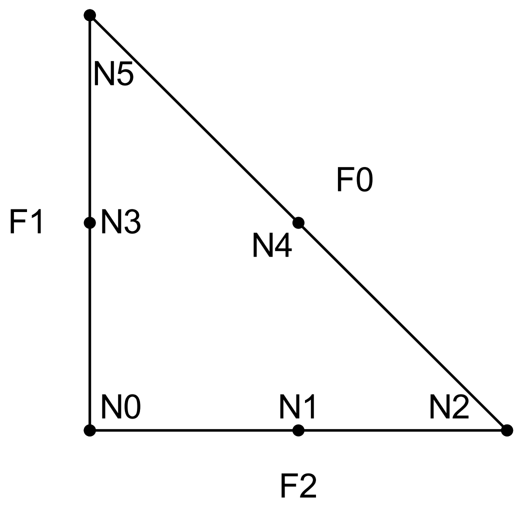
Table 3.1: Number of nodes as a function of q, the order of the geometry approximation, for each supported element type.

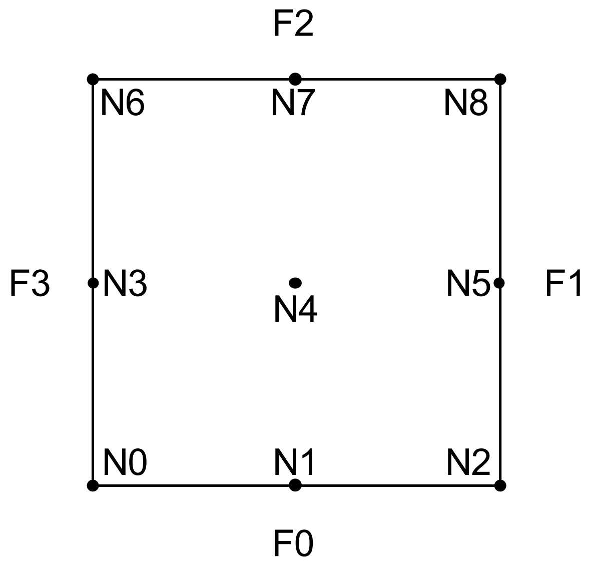
Segments	Triangles	Quadrilaterals
$q+1$	$\frac{(q+1)(q+2)}{2}$	$(q+1)^2$

[1]See https://gmsh.info/ for downloads, documentation, and tutorials.



(a) Node and face orderings for $q = 2$ line segments





(b) Node and face orderings for $q = 2$ triangles

(c) Node and face orderings for $q = 2$ quadrilaterals

Figure 3.1: Node (N0, N1, etc.) and face (F0, F1, etc.) orderings for $q = 2$ line segments, triangles, and quadrilaterals.

3.5 Examples

A suite of example cases can be found in the `examples` directory. Below is a list of the current cases, organized by physics type and number of spatial dimensions. Additional details are found in README files in the corresponding directories.

- Scalar
  - 1D
    - Constant advection of a sine wave
    - Constant advection of a sine wave with a damping source term
    - Burgers equation with an initial sine wave
    - Constant advection-diffusion of a Gaussian profile
  - 2D
    - Constant advection of a Gaussian profile
    - Constant advection-diffusion of a Gaussian profile
- Euler
  - 1D
    - Smooth isentropic flow
    - Moving shock wave

    * Sod shock tube problem
    * Sinusoidal density wave with a stiff source term
  - 2D
    * Isentropic vortex propagation
    * Inviscid Taylor-Green vortex
    * Subsonic flow over a cosine-shaped bump
    * Riemann problem with a gravity source term
- Navier-Stokes
  - 2D
    * Manufactured Solution

3.6 Additional tools

In the `tools` directory are additional tools for performing dissipation and dispersion analysis and plotting basis functions. See the README file in the root Quail directory for more information.

Chapter 4

Developing Quail

4.1 Code style

Some general guidelines to follow are:

- Use tabs with a length of four spaces
- Indent continued lines *twice*
- Limit lines to a maximum of 78 characters

For more detailed guidelines, please refer to the PEP 8 style guide for Python code[1], which we largely aim to follow.

4.1.1 Commenting style

The following subsections outline how functions and classes should be commented.

4.1.1.1 Classes

```
'''
Class: ClassName
-------------------
This class contains information about "blank"

Attributes:
-----------
    attribute name: brief description
'''
```

4.1.1.2 Function definitions

```
'''
Method: name_of_function
--------------------------
Brief description of function

Inputs:
-------
    input name: brief description
Outputs:
--------
```

[1] https://www.python.org/dev/peps/pep-0008/

```
        output name: brief description
    Notes:
    ------
        additional notes
    '''
```

4.2 Debugging

This section provides some debugging tips. Breakpoints can be set by inserting either of the following lines to the code:

- `import pdb; pdb.set_trace()`
  - To continue running the code, use the `continue` command.
  - To exit the program, press `CTRL+d` or use the `exit()` command.
- `import code; code.interact(local=locals())`
  - To continue running the code, press `CTRL+d`
  - To exit the program, use the `exit()` command.
- `breakpoint()`
  - To continue running the code, type `c`.
  - To exit the program, use the `exit()` command.

4.3 Basic code structure

The driver function for Quail is in `src/quail`. Other top-level modules are: `src/errors.py`, which contains user-defined exceptions; `src/defaultparams`, which contains default parameters for input decks, along with descriptions; and `src/general.py`, which contains constants and general Enumerations. Sub-directories in `src` are as follows:

- `meshing`, which contains modules for mesh-related classes and functions. Mesh generation tools are included here. For additional information on mesh generation, node ordering, and face ordering, see Section 3.4.
- `numerics`, which contains modules for basis functions, quadrature, limiters, and time stepping.
- `physics`, which contains modules for various equation sets and related analytic functions, boundary conditions, and numerical fluxes.
- `processing`, which contains modules for plotting, post-processing, and reading and writing data files.
- `solver`, which contains modules for various solvers (DG and ADER-DG).

Section 4.4 provides additional information on some of the above modules to aid in development.

4.4 Base classes

This section provides information on some of the important base classes in order to facilitate development. Detailed comments on each of these classes are also provided in the source files. More classes will be added to this section in the future.

Note that an *abstract* property or method must be overridden by a derived class. More information can be found https://docs.python.org/3/library/abc.html.

4.4.1 Numerics

This subsection focuses on the `numerics` module.

4.4.1.1 `StepperBase`

This is the abstract base class for the `numerics/timestepping/stepper` module.

Abstract properties
None

Attributes (non-abstract)

- `res`
  - Pre-allocated array to store the residual
- `dt`
  - Time step size
- `num_time_steps`
  - Number of time steps
- `get_time_step`
  - Function to compute the time step size according to the user-specified parameters, i.e., based on prescribed time step size, number of time steps, or CFL
- `balance_const`
  - For use in operator splitting

Abstract methods

- `take_time_step`
  - Advances the solution in time by one step

Non-abstract methods
None

Derived classes

- `FE`, `RK4`, `RK4`, `LSRK4`, `SSPRK3`, `ADER`, `Strang`, and `ODEIntegrator`
  - Found in `src/numerics/timestepping/stepper.py`

4.4.1.2 `LimiterBase`

This is the abstract base class for the `numerics/limiting` module.

Abstract properties

- `COMPATIBLE_PHYSICS_TYPES`
  - Physics types compatible with the given limiter

Attributes (non-abstract)
None

Abstract methods

- `precompute_helpers`
  - Preallocate and/or precompute helper attributes, e.g., values of basis functions at specific points and IDs of neighboring elements
- `limit_solution`
  - Limit the solution

Non-abstract methods

- `check_compatibility`
  - Check for compatibility with a specific physics type

Derived classes

- `PositivityPreserving` and `PositivityPreservingChem`
  - Found in `src/numerics/limiting/positivitypreserving.py`
- `WENO`
  - Found in `src/numerics/limiting/wenolimiter.py`

4.4.2 Physics

This subsection focuses on the `physics` module.

4.4.2.1 `FcnBase`

This is an abstract base class found in the `physics/base/data` module. It is used for describing an analytical function for initial conditions, exact solutions, and/or boundary conditions.

Abstract properties
None

Attributes (non-abstract)
None

Abstract methods

- `get_state`
  - Compute the state

Non-abstract methods
None

Derived classes
There are many derived classes. To name a few:

- `Uniform`
  - Found in `src/physics/base/functions.py`
- `Sine`, `Gaussian`, and others

  - Found in `src/physics/scalar/functions.py`

- `IsentropicVortex`, `DensityWave`, and others
  - Found in `src/physics/euler/functions.py`

4.4.2.2 `BCBase`

This is an abstract base class found in the `physics/base/data` module. It is used for imposing boundary conditions.

Abstract properties
None

Attributes (non-abstract)
None

Abstract methods

- `get_boundary_state`
  - Compute the exterior state at the boundary
- `get_boundary_flux`
  - Compute the flux at the boundary

Non-abstract methods
None

Derived classes

- `BCWeakRiemann`
  - Found in `src/physics/base/data.py`
  - This class computes the boundary flux using the numerical flux function
  - Classes derived from `BCWeakRiemann`:
    * `StateAll` in `src/physics/base/functions.py`
- `BCWeakPrescribed`
  - Found in `src/physics/base/data.py`
  - This class computes the boundary flux via the analytic flux based on the exterior state
  - Classes derived from `BCWeakPrescribed`:
    * `Extrapolate` in `src/physics/base/functions.py`
    * `SlipWall` and `PressureOutlet` in `src/physics/euler/functions.py`

4.4.2.3 `SourceBase`

This is an abstract base class found in the `physics/base/data` module. It is used for computing source terms.

Abstract properties
None

Attributes (non-abstract)

- `source_treatment`
  - Specifies whether the treat the source term explicitly or implicitly for ADER-DG

Abstract methods

- `get_source`
  - Compute the source term

Non-abstract methods

- `get_jacobian`
  - Compute the Jacobian of the source term

Derived classes
There are many derived classes. To name a few:

- `TaylorGreenSource` and `GravitySource`
  - Found in `src/physics/euler/functions.py`
- `Arrhenius`
  - Found in `src/physics/chemistry/functions.py`

4.4.2.4 `ConvNumFluxBase`

This is an abstract base class found in the `physics/base/data` module. It is used for convective numerical flux functions.

Abstract properties
None

Attributes (non-abstract)
None

Abstract methods

- `compute_flux`
  - Compute the convective numerical flux

Non-abstract methods

- `alloc_helpers`
  - Allocate helper arrays

Derived classes
There are many derived classes. To name a few:

- `LaxFriedrichs1D` and `Roe1D`
  - Found in `src/physics/euler/functions.py`

4.4.2.5 `DiffNumFluxBase`

This is an abstract base class found in the `physics/base/data` module. It is used for diffusive numerical flux functions.

Abstract properties
None

Attributes (non-abstract)
None

Abstract methods

- `compute_flux`
  - Compute the diffusive numerical flux

Non-abstract methods

- `alloc_helpers`
  - Allocate helper arrays

Derived classes

- `SIP` (Symmetric Interior Penalty)
  - Found in `src/physics/base/functions.py`

4.5 Glossary

Table 4.2 provides a glossary of commonly used variable names.

`U, Uc`	State coefficients (degrees of freedom)
`Uq`	State variables evaluated at a set of points (typically quadrature points)
`res`	Residual vector
`dim`	Number of dimensions
`order`	Order of solution approximation
`gorder`	Order of geometric approximation
`ns`	Number of state variables
`ne, num_elems`	Number of elements
`nf`	Number of faces
`nbf`	Number of boundary faces
`nq`	Number of quadrature points
`nq_st`	Number of quadrature points in space-time
`nb`	Number of basis functions (per state variable)
`nb_st`	Number of space-time basis functions (per state variable)
`quad_pts`	Coordinates of quadrature points in ref. space
`quad_wts`	Quadrature weights
`jac`	Jacobian $\nabla_{\boldsymbol{\xi}} \boldsymbol{x}$

`ijac`	Inverse of the Jacobian
`djac`	Determinant of the Jacobian
`normals`	Outward-pointing normals
`basis_val`	Basis value (evaluated at a set of points)
`basis_ref_grad`	Basis reference space gradient (evaluated at a set of points)
`basis_phys_grad`	Basis physical space gradient (evaluated at a set of points)
`Fq`	Flux evaluated at a set of points (typically quadrature points)
`Sq`	Source term evaluated at a set of points (typically quadrature points)
`[x, y]`	Coordinates in physical space
`MM`	Mass matrix
`iMM`	Inverse mass matrix
`interior_face, int_face`	Interior face
`boundary_face, bface`	Boundary face
`boundary_group, bgroup`	Boundary group
`*L`	Left quantity
`*R`	Right quantity
`*I`	Interior quantity
`*B`	Boundary quantity
`u`	Scalar (for scalar equations)
`[u, v]`	Velocity (for hydrodynamics)
`rho`	Density (for hydrodynamics)
`rhou, rhov`	Momentum (for hydrodynamics)
`rhoE`	State energy variable (for hydrodynamics)
`p`	Pressure (for hydrodynamics)
`T`	Temperature (for hydrodynamics)
`gamma`	Ratio of specific heats (for hydrodynamics)
`R`	Specific gas constant (for hydrodynamics)

Table 4.2: Glossary of variable names

Chapter 5

Tutorials

This chapter discusses video tutorials on using and developing QUAIL. These videos can be found on our YouTube channel at https://www.youtube.com/channel/UCElNsS_mm_0c6X41qVKBMew. More videos will be added in the future.

5.1 Using QUAIL

These videos focus on using QUAIL. Please refer to the electronic version to click on the corresponding links.

- Getting Started with QUAIL
- Creating an Animation
- Constructing an Input Deck

5.2 Developing QUAIL

These videos focus on developing QUAIL. Please refer to the electronic version to click on the corresponding links.

- Adding Initial Conditions and Exact Solutions
- Adding a New Physics Module
- Adding Boundary Conditions

Bibliography

[1] J. S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, volume 54. Springer, 2007.

[2] M. Dumbser, C. Enaux, and E.F. Toro. Finite volume schemes of very high order of accuracy for stiff hyperbolic balance laws. *Journal of Computational Physics*, 227:3971–4001, 2008.

[3] M. Dumbser and O. Zanotti. Very high order $p_n p_m$ schemes on unstructured meshes for the resistive relativistic mhd equations. *Journal of Computational Physics*, 228:6991–7006, 2009.

[4] R.H. Bartels and G.W. Stewart. Solution of the matrix equation $\boldsymbol{AX} + \boldsymbol{XB} = \boldsymbol{C}$. *Communications of the ACM*, 15, 1972.

[5] M. H. Carpenter and C. Kennedy. Fourth-order 2N-storage Runge-Kutta schemes. Technical report, NASA Langley Research Center, NASA Report TM 109112, Richmond, VA, USA, 1994.

[6] R. Spiteri and S.J. Ruuth. A new class of optimal high-order strong-stability-preserving time discretization methods. *SIAM Journal of Numerical Analysis*, 40(2):469–491, 2002.

[7] G. Strang. On the construction and comparison of difference schemes. *SIAM Journal of Numerical Analysis*, 5(3), 1968.

[8] H. Wu, P. Ma, and M. Ihme. Efficient time-stepping techniques for simulating turbulent reactive flows with stiff chemistry. *Computer Physics Communications*, 243:81–96, 2019.

[9] G. Mengaldo, D. De Grazia, F. Witherden, A. Farrington, P. Vincent, S. Sherwin, and J. Peiro. A guide to the implementation of boundary conditions in compact high-order methods for compressible aerodynamics. In *7th AIAA Theoretical Fluid Mechanics Conference*. AIAA-2014-2923, 2014.

[10] P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43(2):357–372, 1981.

[11] R. Hartmann and P. Houston. An optimal order interior penalty discontinuous Galerkin discretization of the compressible Navier–Stokes equations. *Journal of Computational Physics*, 227(22):9670–9685, 2008.

[12] C.-W. Shu. Tvb uniformly high-order schemes for conservation laws. *Mathematics of Computation*, 49:105–211, 1987.

[13] X. Zhong and C.-W. Shu. A simple weighted essentially nonoscillatory limiter for Runge-Kutta discontinuous Galerkin methods. *Journal of Computational Physics*, 232(1):397„1¤7415, 2013.

[14] X. Zhang and C.-W. Shu. On positivity-preserving high order discontinuous Galerkin schemes for compressible Euler equations on rectangular meshes. *Journal of Computational Physics*, 229(23):8918–8934, 2010.

[15] C. Wang, X. Zhang, C.-W. Shu, and J. Ning. Robust high order discontinuous Galerkin schemes for two-dimensional gaseous detonations. *Journal of Computational Physics*, 231(2):653–665, 2012.

[16] R. Fedkiw. A survey of chemically reacting, compressible ows. Ph.D. Thesis, University of California Los Angeles, 1997.

[17] C. Geuzaine and J. F. Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal of Numerical Methods in Engineering*, 79(11):1309–1331, 2009.