Discussion of Python Implementation Techniques for Discontinuous Galerkin Methods

Ulf Vestberg

Bachelor's thesis
2023:K13

Lund University

Faculty of Science
Centre for Mathematical Sciences
Numerical Analysis

Centrum Scientiarum Mathematicarum

Abstract

This paper discusses implementation techniques for integration methods within the Discontinuous Galerkin Methods. These methods are used to approximate solutions for differential equations. To do so, one must compute a polynomial u, of degree P, which is an approximation of a function f. This approximation is done by L^2 projection, which uses orthogonal polynomials, chosen to be the Legendre polynomials, as well as integration. This is first demonstrated for one dimension. For two or three dimension, full tensor product or sum-factorization is used. These implementations give the same approximation of u, but sum-factorization uses fewer computations than full tensor product. It is demonstrated how the L^2 projection is computed in Python and also that sum-factorization is faster to compute than the full tensor product based approach.

Contents

1 Introduction

In this paper it is shown how to give a best approximation of a function $f \in L^2[a,b]$, where $f \in L^2[a,b]$ is projected on $V \subset L^2[a,b]$ with

$$\dim V = P + 1 < \infty \tag{1.1}$$

and V being a polynomial space. To find the best approximation, one uses a set of basis functions that are orthogonal polynomials $\psi_i(x)$ over the domain $[a,b]$, such that

$$V = \text{span}\{\psi_0(x), ..., \psi_P(x)\}. \tag{1.2}$$

The set of orthogonal polynomials are the Legendre polynomials. To compute the projection, one computes a set of coefficients $\hat{u}_i$, by integration such that

$$\hat{u}_i = \int_a^b f(x)\psi_i(x)dx, \; i = 0, ..., P. \tag{1.3}$$

In general, one cannot solve integrals analytically. Hence, the integration will be approximated numerically. This is done is by Gauss-Legendre quadrature. Once the coefficients u_i are computed, one can compute the projected function $u \in V$ with either full tensor product, or, by the faster implementation of sum-factorization.

In general, sum-factorization and full tensor product gives the same approximation. The difference of the implementations are that sum-factorization requires fewer computations than full tensor product. This is shown in Chapter 6, where the numerical experiments are presented. These experiments are performed in Python. In that chapter, algorithms, approximations and approximation errors are shown. The chapter is then summarized by presenting the computation times for all implementations.

2 L^2 projection

The idea of L^2 projection is for a function $f \in L^2(a,b)$, to find a polynomial of degree P, $u \in \mathcal{P}_P$, such that

$$\|f-u\|_2 = \inf_{v\in\mathcal{P}_P} \|f-v\|_2. \tag{2.1}$$

The polynomial u is a best approximation of degree P to the function f in the 2-norm on the interval (a,b) [4, p.256]. Given $f \in L^2(a,b)$, such a polynomial exists and is unique [4, p.258]. One can describe the polynomial u using the monomial basis, by the formula

$$u(x) = \sum_{i=0}^{P} c_i x^i. \tag{2.2}$$

To find $u(x)$, we choose coefficients

$$c_i,\ i = 0,..,P, \tag{2.3}$$

such that we minimize the error

$$\|e_n\|_2 = \|f-u\|_2 = (\int_0^1 |f(x)-u(x)|^2)^{1/2}. \tag{2.4}$$

Without loss of generality, we consider the square of (2.4) and obtain the following minimization problem

$$\begin{aligned} E(c_0,...,c_P) &= \int_0^1 [f(x)-u(x)]^2 dx \\ &= \int_0^1 [f(x)]^2 dx - 2\sum_{i=0}^{P} c_i \int_0^1 f(x)x^i dx + \sum_{i=0}^{P}\sum_{j=0}^{P} c_i c_j \int_0^1 x^{i+j} dx. \end{aligned} \tag{2.5}$$

The minimum is obtained when all partial derivatives of E, with respect to c_i are 0 [4, p.258]. This means that for the coefficients c_i, one has to solve the following system of equations

$$\sum_{j=0}^{P} M_{ij} c_j = \beta_i,\ i = 0,...,P, \tag{2.6}$$

where M_{ij} and β_i, are defined by

$$M_{ij} = \int_0^1 x^{i+j} dx = \frac{1}{1+i+j},\ \beta_i = \int_0^1 f(x)x^i dx, \tag{2.7}$$

see for example [4, p.259].

Once all coefficients c_i are computed, we get a polynomial that is the best approximation of f in $\mathcal{P}_P$ with respect to the 2-norm. The example above demonstrates how such a polynomial can be computed on the interval $(0,1)$. It is possible to compute u on any interval (a,b) for any positive, continuous and integrable weight function ω on (a,b) [4, p.259].

3 Orthogonal polynomials

3.1 Introduction

In Chapter 2, it was described how to compute the polynomial best approximation $u \in \mathcal{P}_P$ to a function f in the 2-norm. The matrix M from the previous chapter is ill-conditioned, which makes it difficult to solve, for the coefficients c_i [4, p.250]. To counteract this, we want to construct M, such that, M is a diagonal matrix. This can be achieved by choosing a different basis. In particular a basis consisting of orthogonal polynomials. The polynomial u will be defined by

$$u(x) = \sum_{i=0}^{P} \hat{u}_i \psi_i(x), \tag{3.1}$$

for coefficients $\hat{u}_i$ and basis functions $\psi_i(x)$. The basis functions in (3.1) are orthogonal polynomials.

Definition 3.1. *Given a weight function $\omega(x)$, which is positive, continuous and integrable on the interval (a, b), we say that the sequence of polynomials $\{\psi_0(x), \psi_1(x), ...\}$, is a system of orthogonal polynomials on the interval (a, b) with respect to ω, if each ψ_i is of exact degree i and if*

$$\int_a^b \omega(x)\psi_i(x)\psi_j(x)dx \begin{cases} = 0, \ \forall i \neq j \\ \neq 0, \ \text{when } i = j \end{cases} \tag{3.2}$$

3.2 Legendre polynomials

Legendre polynomials are orthogonal polynomials and the basis of choice for this project.

Definition 3.2. *The set of Legendre polynomials $L_i(x)$ are orthogonal with respect to the basis function $\omega(x) = 1$, on the interval (a, b) [4, p.263].*

Definition 3.3. *The Legendre polynomials satisfy*

$$L_i(x) = \frac{d^i}{dx^i}((x^2 - 1)^i)\frac{i!}{(2i)!} \tag{3.3}$$

in $(-1, 1)$ for degree i [2, p.127].

Legendre polynomials can be defined on other domains than $(-1, 1)$. For example, Legendre polynomials defined on the domain $(0, 1)$ are very useful. Legendre polynomials on this domain will help with integration, which is discussed in the next chapter. We can also compute Legendre polynomials in any other interval (a,b). This is required to compute $u(x)$ on an arbitrary interval (a, b).

Definition 3.4. *If there exists a set of orthogonal polynomials on an interval $(-1, 1)$, with respect to the weight function $\omega(x) = 1$, the polynomials $\psi_i((2x - a - b)/(b - a))$, represent an orthogonal system on the interval (a, b), with respect to $\omega(x) = 1$.*

For this project, we mainly use normalized Legendre polynomials $\hat{L}_i(x)$, of degree i, that are defined on the interval $(0, 1)$. To compute these polynomials, we use the previously specified Legendre polynomials $L_i(x)$, that are defined on the interval $(-1, 1)$. The polynomials $\hat{L}_i(x)$ are computed by the formula

$$\hat{L}_i(x) = L_i(2x-1)/\|L_i(2x-1)\|_2 = \sqrt{(2i+1)}L_i(2x-1). \tag{3.4}$$

The main purpose of the polynomial $\hat{L}_i(x)$ is to serve as a basis for the polynomial approximation of f. This is discussed in the next chapter.

The use of Legendre polynomials as basis functions, yields a diagonal matrix M, which makes the computation of the best approximation in the L^2 sense easy. This will be discussed in the next chapter.

Because these polynomials are orthogonal with respect to each other, we get a matrix M which is diagonal. We evaluate the coefficients $\hat{u}$ by the system of equation

$$M\hat{u} = \beta. \tag{3.5}$$

Where the entries of M and β are defined by

$$M_{ij} = \delta_{ij}\|\psi_i(x)\|_2^2, \tag{3.6}$$
$$\beta_i = \langle f(x), \psi_i(x)\rangle, \tag{3.7}$$

where δ_{ij} is the Kronecker delta.

4 Gauss-Legendre quadrature

In this chapter it is shown how definite integrals are computed. In general, it is not possible to solve integrals analytically. Hence, we will approximate each definite integral numerically. One can compute numerical approximations of definite integrals by Gauss-Legendre quadrature. This is done by summing a set of quadrature points, with corresponding weight points, on the domain that the function is integrated over. The purpose of this chapter is to explain what the correct quadrature points and weight points should be and how this gives us numerical approximations of definite integrals. An error approximation of the integration is also shown in the chapter.

Definition 4.1. *For a set of quadrature points $\hat{x}_i$ and weight points $\hat{\omega}_i$, $i = 0, ..., P$, a definite integral of an arbitrary function $g(x)$ over a domain $[a, b]$ is approximated by the formula*

$$I = \int_a^b g(x)dx \approx \sum_{i=0}^{P} \hat{\omega}_i g(\hat{x}_i), \tag{4.1}$$

see for example [4, p.280].

The domain $[a, b]$ can also be split into several sub-domains. If we have N sub-domains, we can define these domains as

$$[a, b] = \{[a, a + h], [a + h, a + 2h], ..., [b - h, b]\}. \tag{4.2}$$

This is used to compute a set of integrals I_k, such that

$$I_k = \int_{a+kh}^{a+(k+1)h} g(x)dx, \; k = 0, ..., P, \tag{4.3}$$

where

$$h = \frac{b - a}{N}, \tag{4.4}$$

is the increment of x for each step.

The quadrature points and weight points are defined below. It is desired for the quadrature points $\hat{x}_i$ and weight points $\hat{\omega}_i$ to satisfy the following properties.

$$\begin{aligned}
\int_a^b dx &= \sum_{i=0}^{P} \hat{\omega}_i, \\
\int_a^b x dx &= \sum_{i=0}^{P} \hat{\omega}_i \hat{x}_i, \\
&\vdots \\
\int_a^b x^{2P+1} dx &= \sum_{i=0}^{P} \hat{\omega}_i \hat{x}_i^{2P+1}.
\end{aligned}$$

If this is is true for all quadrature points and weights, the quadrature is exact for polynomials of at most degree $2P + 1$ [4, p.280-281].

The quadrature points $\hat{x}_i$ are given by the zeros of the Legendre polynomial of degree $P + 1$ on the interval (a, b) [4, p.279]. Once we have computed the quadrature points, it is straight forward to compute the weight points, since they are based on the quadrature points and Legendre polynomial of degree $P + 2$ [4, p.279].

We use everything discussed in this chapter to handle integration to compute all β_i in equation (3.7), which is defined by

$$\beta_i = \int_a^b f(x)\psi_i(x)dx,\ i = 0, ..., P, \tag{4.5}$$

where $f(x)$ is an arbitrary function and $\psi_i(x)$ are Legendre polynomials on the interval (a, b) of degree i.

The first step of evaluating the definite integral is to compute all points of ψ_{pi}, which is a matrix of shape $(P + 1) \times (P + 1)$. What we want is the quadrature points $\hat{x}_p$ and weight points $\hat{\omega}_p$ over the domain $[0, 1]$. Next, we define the basis function $\psi_i(x)$ as the Legendre polynomial of degree i on the domain $[0, 1]$. This gives us that the points of ψ_{pi} are defined by

$$\psi_{pi} = \hat{\omega}_p \hat{L}_i(\hat{x}_p). \tag{4.6}$$

Because the integration will only be carried out on a reference domain, we only need to compute the values of ψ_{pi} once. What we want to achieve with the integration is to compute the coefficients $\hat{u}_i$ which are computed by the formula

$$\hat{u}_i = \int_a^b f(x)\psi_i(x)dx \approx \sum_{p=0}^{P} f(\hat{x}_p)\psi_{pi},\ 0 \leq i \leq P. \tag{4.7}$$

Once the points of $\hat{u}_i$ are computed, we can use (3.1) to evaluate the polynomial $u(x)$ in any point $x \in [a, b]$.

5 Sum-factorization

Previously, we have demonstrated how to compute u of one dimension. For this project, we are generally interested in computing u of two or three dimensions. For the 1D case, we have shown how to compute $u(x)$ for some points $\hat{u}_i$ and basis functions $\psi_i(x)$.

When the dimension is greater than 1, we still use Legendre polynomials as basis functions. To compute the function $u(x, y)$ for two dimensions, or $u(x, y, z)$ for three dimension, the implementations that are used are either by full tensor product or sum-factorization. We start by defining the tensor product implementation. For two dimensions, the full tensor product is defined by

$$u(x, y) = \sum_{i=0}^{P} \sum_{j=0}^{P} \psi_i(x)\psi_j(y)\hat{u}_{ij} \tag{5.1}$$

and for three dimensions, it is defined by

$$u(x, y, z) = \sum_{i=0}^{P} \sum_{j=0}^{P} \sum_{k=0}^{P} \psi_i(x)\psi_j(y)\psi_k(z)\hat{u}_{ijk}. \tag{5.2}$$

Full tensor product is expensive computationally. In total, the number of operations are roughly

$$2(P+1)^2 \tag{5.3}$$

multiplications and

$$(P+1)^2 \tag{5.4}$$

summations, for a total of

$$3(P+1)^2 \tag{5.5}$$

operations, for the 2D case and

$$3(P+1)^3 \tag{5.6}$$

multiplications and

$$(P+1)^3 \tag{5.7}$$

summations, for a total of

$$4(P+1)^3 \tag{5.8}$$

operations, for the 3D case.

We can reduce the number of operations by computing (5.1) and (5.2) differently. This is done by sum-factorization.

Lemma 5.1. *For the 2D case, $u(x,y)$ can be computed by full tensor product and sum-factorization*

$$u(x,y) = \sum_{i=0}^{P}\sum_{j=0}^{P}\psi_i(x)\psi_j(y)\hat{u}_{ij} = \sum_{j=0}^{P}\psi_j(y)\left\{\sum_{i=0}^{P}\psi_i(x)\hat{u}_{ij}\right\} \tag{5.9}$$

which are equivalent [5, p.5167].

Proof. If we rearrange the factors we get the computations

$$u(x,y) = \sum_{i=0}^{P}\sum_{j=0}^{P}\psi_i(x)\psi_j(y)\hat{u}_{ij} =$$
$$\psi_0(y)\sum_{i=0}^{P}\psi_i(x)\hat{u}_{i0} + ... + \psi_P(y)\sum_{i=0}^{P}\psi_i(x)\hat{u}_{iP} =$$
$$\sum_{j=0}^{P}\psi_j(y)\left\{\sum_{i=0}^{P}\psi_i(x)\hat{u}_{ij}\right\}.$$

□

Lemma 5.2. *For the 3D case, $u(x,y,z)$ can be computed by full tensor product and sum-factorization*

$$\begin{aligned} u(x,y,z) &= \sum_{i=0}^{P}\sum_{j=0}^{P}\sum_{k=0}^{P}\psi_i(x)\psi_j(y)\psi_k(z)\hat{u}_{ijk} = \\ &\sum_{k=0}^{P}\psi_k(z)\left\{\sum_{j=0}^{P}\psi_j(y)\left\{\sum_{i=0}^{P}\psi_i(x)\hat{u}_{ijk}\right\}\right\} \end{aligned} \tag{5.10}$$

which are equivalent.

Proof. If we rearrange the factors we get the computations

$$u(x,y,z) = \sum_{i=0}^{P}\sum_{j=0}^{P}\sum_{k=0}^{P}\psi_i(x)\psi_j(y)\psi_k(z)\hat{u}_{ijk} =$$
$$\psi_0(z)\sum_{i=0}^{P}\sum_{j=0}^{P}\psi_i(x)\psi_j(y)\hat{u}_{ij0} + ... + \psi_P(z)\sum_{i=0}^{P}\sum_{j=0}^{P}\psi_i(x)\psi_j(y)\hat{u}_{ijP} =$$
$$\sum_{k=0}^{P}\psi_k(z)\left\{\sum_{i=0}^{P}\sum_{j=0}^{P}\psi_i(x)\psi_j(y)\hat{u}_{ijk}\right\} =$$
$$\sum_{k=0}^{P}\psi_k(z)\left\{\psi_0(y)\sum_{i=0}^{P}\psi_i(x)\hat{u}_{i0k} + ... + \psi_P(y)\sum_{i=0}^{P}\psi_i(x)\hat{u}_{iPk}\right\} =$$
$$\sum_{k=0}^{P}\psi_k(z)\left\{\sum_{j=0}^{P}\psi_j(y)\left\{\sum_{i=0}^{P}\psi_i(x)\hat{u}_{ijk}\right\}\right\}.$$

□

Sum-factorization requires fewer operations than full tensor product to compute. For the 2D case, there are roughly

$$((P+1)+1)(P+1) = (P+1)^2 + (P+1) \tag{5.11}$$

multiplications and

$$(P+1)^2 \tag{5.12}$$

summations, for a total of

$$2(P+1)^2 + (P+1) \tag{5.13}$$

operations. For the 3D case, there are roughly

$$((P+1)+2)(P+1)^2 = (P+1)^3 + 2(P+1)^2 \tag{5.14}$$

multiplications and

$$(P+1)^3 \tag{5.15}$$

summations, for a total of

$$2(P+1)^3 + 2(P+1)^2 \tag{5.16}$$

operations.

The above argument shows that theoretically it should be faster to compute u using sum-factorization rather than full tensor product. In the next chapter, it is shown in Python that it is faster to compute u with sum-factorization, than with full tensor product.

6 Numerical experiments

6.1 Introduction

In this chapter, the Python implementations of the project are discussed. The main goal of this project is to test computation times of the full tensor product and sum-factorization implementations and compare the difference between these. This will summarize the chapter. Before discussing computation times, it is explained how Dune uses the previously discussed aspects, such as L^2 projection, integration, basis functions, full tensor product and sum-factorization. There are also some explanation about the algorithms of the tensor product and sum-factorization implementations, followed by showing the approximation and approximation error, given a test function.

6.2 Dune framework

Most implementations used in this chapter are from a tool called Dune [3]. The way Dune is used is to create grids of one, two or three dimensions, depending on the function we want to approximate, as well as, basis functions, quadrature points and weight points. Dune also does interpolation automatically for computed coefficients. One can implement the mentioned tools by the code

Python code

```
import os, io
# ensure some compilation output for this example
os.environ['DUNE_LOG_LEVEL'] = 'info'
print("Using DUNE_LOG_LEVEL=",os.getenv('DUNE_LOG_LEVEL'))

from ufl import SpatialCoordinate, pi, sin, dot

from dune.generator import algorithm
from dune.geometry import quadratureRule
from dune.grid import structuredGrid as leafGridView
from dune.fem.space import dglegendre
from dune.fem.function import uflFunction, integrate

import numpy as np

_code = \
"""
#ifndef FEM_EVAL_BASIS_HH
#define FEM_EVAL_BASIS_HH
#include <vector>
#include <array>
#include <utility>
#include <dune/python/pybind11/numpy.h>
#include <dune/common/fvector.hh>
#include <dune/common/dynvector.hh>

template <class Space, class Entity, class Point>
std::vector< typename Space::RangeType >
evaluateBasis(const Space& space, const Entity& entity, const Point& x)
{
  const auto basisSet = space.basisFunctionSet( entity );
  std::vector< typename Space::RangeType > basis( basisSet.size() );
  basisSet.evaluateAll( x, basis );
  return basis;
}
#endif

"""
```

Once all required tools are imported from Dune, quadrature points and weight points can be evaluated. These points are used to compute a set of basis points that are used to compute the L^2 projection, which is a polynomial of deg $\leq P$, with either full tensor product or sum-factorization. The points are evaluated by Dune by the code

Python code

```
def basisfunctions_1d(P):

    gridView = leafGridView([0], [1], [1])
    space = dglegendre(gridView, dimRange = 1, order = P)

    basisEval = None

    w = []

    basis_points = []

    for e in space.grid.elements:

        # loop over all quadrature points
        for p in quadratureRule(e.type, 2 * P + 1):

            # coordinate of quadrature point (\hat{x})
            x = p.position

            # weight point
            w.append(p.weight) # * geo.integrationElement(x)

            if basisEval is None:
                basisEval = algorithm.load('evaluateBasis', \
                                           io.StringIO(_code), space, e, x)

            # evaluate all basis function at the quadrature point
            basis_points.append(basisEval(space, e, x))

    psi_Vals = np.array(basis_points).reshape(P + 1, P + 1) * np.vstack(w)

    return psi_Vals
```

If one computes these points mathematically, one uses the quadrature points $\hat{x}_p$ and weight points $\hat{\omega}_p$, with Legendre polynomials $\hat{L}_i(x)$, of degree i, defined on $(0, 1)$. Then the basis points ψ_{pi} are defined as

$$\psi_{pi} = \omega_p \hat{L}_i(\hat{x}_p),\ p, i = 0, ..., P. \tag{6.1}$$

The values of ψ_{pi} are given for 1D and used to construct basis functions for higher dimension by means of full tensor product. These are stored in a separate matrix for each dimension. The advantage this gives, is that fewer computations are required when performing full tensor product. This is the way Dune does the L^2 projection. The way Dune evaluates basis values, is by the code

Python code

```
def basisfunctions(P, d):

    gridView = leafGridView(d * [0], d * [1], d * [1])
    space = dglegendre(gridView, dimRange = 1, order = P)

    basisEval = None

    w = []

    basis_points = []

    for e in space.grid.elements:

        # loop over all quadrature points
        for p in quadratureRule(e.type, 2 * P + 1):

            # coordinate of quadrature point (\hat{x})
            x = p.position

            # weight point
            w.append(p.weight) # * geo.integrationElement(x)

            if basisEval is None:

                basisEval = algorithm.load('evaluateBasis', \
                                           io.StringIO(_code), space, e, x)

            # evaluate all basis function at the quadrature point
            basis_points.append(basisEval(space, e, x))

    phi_Vals = np.array(basis_points).reshape((P + 1)**d, (P + 1)**d) * \
        np.vstack(w)

    return phi_Vals
```

This algorithm creates a matrix ϕ of size $(P+1)^d \times (P+1)^d$, compared to ψ, which is a matrix of size $(P+1) \times (P+1)$. The way these points are evaluated given elements of $\psi_{..}$, is by

$$\phi_{(P+1)p+q,val_{ij}} = \psi_{qi}\psi_{pj},\ p,q,i,j = 0,...,P, \tag{6.2}$$

for two dimensions, and

$$\phi_{(P+1)^2p+(P+1)q+r,val_{ijk}} = \psi_{ri}\psi_{qj}\psi_{pk},\ p,q,r,i,j,k = 0,...,P, \tag{6.3}$$

for three dimensions. The elements of val shall be seen as the way Dune orders the way to compute elements. The set val is defined as

$$val_{ij} = \begin{cases} i^2 + j,\ j < i \\ j(j+1) + i,\ j \geq i \end{cases} \tag{6.4}$$

for two dimensions, and

$$val_{ijk} = \begin{cases} k((k+1)^2 + j) + i + j,\ k \geq j,\ k \geq i \\ i(i^2 + 2k) + j + k,\ k < i,\ j < i \\ j(j^2 + 2k + 1) + i + k,\ k < j,\ j \geq i \end{cases} \tag{6.5}$$

for three dimensions. To demonstrate with an example. If we have $i = 2,\ j = 1,\ k = 4$, we see from equation (6.5), that Dune will store these values in column $4((4+1)^2 + 1) + 2 + 1 = 107$ of ϕ, for all values of $p,\ q,\ r$.

The set of *val* is used as a demonstration tool to show how one can compute the matrix ϕ from the matrix ψ. The set of *val* is useful when computing either the full tensor product, or sum-factorization, to not violate the built-in ordering of Dune. If *val* is not used in this way for the L^2 projection, the approximation becomes incorrect.

Next step of the project is to introduce a test function and the space we want to approximate over. The test function of choice for the project is the sin function of dimension d. In general, we always want to approximate a function over several grids over some space. The boundary for all axes of the space is defined as a and b. The dimension of the function is defined as d and the number of grids is defined as N. Note that the total number of grids are N^d, meaning, for a two dimensional square, with boundary $a = 0$ and $b = 1$ and $N = 20$, there are 400 grids, portioned evenly over the unit square. Mathematically, the grids over the space are defined as

$$\begin{gathered}\Omega = \Omega_1 \cup \Omega_2 \cup ... \cup \Omega_{399} \cup \Omega_{400} = \\ [0, 0.05] \times [0, 0.05] \cup [0.05, 0.1] \times [0, 0.05] \cup ... \\ \cup [0.95, 1] \times [0.9, 0.95] \cup [0.95, 1] \times [0.95, 1]\end{gathered} \tag{6.6}$$

The way we use this in Dune, is for given values of P, N, d, a and b, as input, we have to store the test function f, as the output `fct`. We also have to create an output for the approximated function u_h. This is the output `uh`. The code below represents the discussed parts

Python code

```
def spaces(P, N, d, a, b):

    gridView = leafGridView(d * [a], d * [b], d * [N])
    space = dglegendre(gridView, dimRange = 1, order = P)

    x = SpatialCoordinate(space)
    f = sin(x[0] * pi)
    for i in range(1, d):
        f *= sin(x[i] * pi)

    fct = uflFunction(gridView, name = 'f', order = P, ufl = f)

    uh = space.interpolate([0], name = 'uh')

    return fct, uh, f, gridView
```

6.3 Approximation and algorithm explanations

We now have all required tools from Dune to compute the L^2 projection of a function. We will do this with 4 implementations and test 3 of these, with various approaches. The first implementation that is used, which is the only implementation to not be tested, is a straight forward built-in implementation. We use the test function

$$f(x, y) = \sin(\pi x) \sin(\pi y) \tag{6.7}$$

and variables

$$P = 3,\ N = 20,\ d = 2,\ a = 0,\ b = 4. \tag{6.8}$$

We use this input, with the output `f` and `gridView` and do the L^2 projection of $f(x, y)$, with the code

Python code

```
space = dglegendre(gridView, dimRange = 1, order = P)
uh = space.interpolate(f, name = 'uh')
```

The command

Python code

```
uh.plot()
```

plots the approximated solution u_h. The approximation, given the values above, is shown below

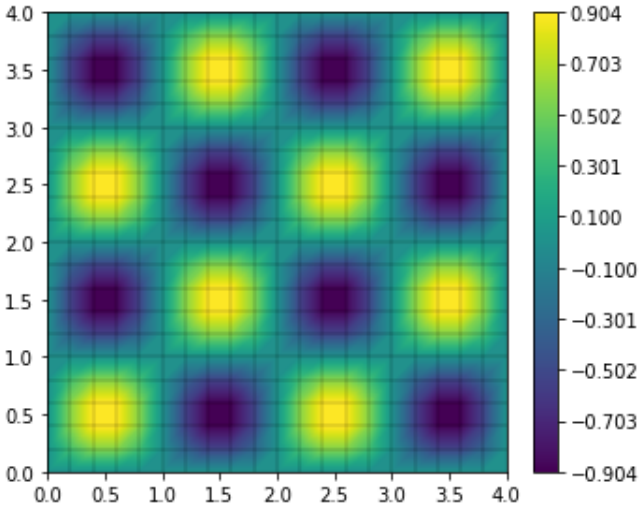

Figure 1: Approximation of u_h, over the domain $\Omega = [0, 4] \times [0, 4]$, polynomial degree 3 and 20 grid per axis, given $f(x, y) = \sin(\pi x)\sin(\pi y)$. The bar to the right represents the value of $u_h(x, y)$.

Going forward, this plot works as sheet to see if a programmed algorithm appears to be correct. In the Chapter 6.4, it is demonstrated that the error of the approximation decreases for increasing polynomial degree P and increasing number of grids N.

6.3.1 Full tensor product

We are now discussing the algorithms we want to test. These are called, full tensor product, Dune implementation and sum-factorization. There will also be a brief discussion, about different approaches, that can be used with the implementations.

The first implementation to test is the full tensor product. In short, what we can say about the full tensor product implementation is that this implementation computes the interpolation points $\hat{u}_{ij}$, for two dimensions, or, $\hat{u}_{ijk}$, for three dimensions, with full number of multiplications, for given values of $\psi_{..}$. Once these points are computed, we compute the coefficients $\hat{u}_{val}$, with the formula

$$\hat{u}_{val_{ij}} = \sum_{0 \leq p,q \leq P} \psi_{qi} \psi_{pj} \hat{f}_{pq} \tag{6.9}$$

for two dimensions, where $\hat{f}_{pq} = f(\hat{x}_q, \hat{y}_p)$, and

$$\hat{u}_{val_{ijk}} = \sum_{0 \leq p,q,r \leq P} \psi_{ri} \psi_{qj} \psi_{pk} \hat{f}_{pqr} \tag{6.10}$$

for three dimensions, where $\hat{f}_{pqr} = f(\hat{x}_r, \hat{y}_q, \hat{z}_p)$.

It is now explained how the computations are being performed. The algorithm we create to perform these computations, uses the input `fct` and `uh`. We must also use the basis matrix, defined as `psi` and the dimension, defined as `d`. In the algorithm, we have a `string` input as well. For this demonstration, we call `string` `'looping'`. The `string` element will be discussed later in this chapter. For now, the way to compute the full tensor product in two dimensions in Python, is by the code

Python code

```
def tensor_product(fct, uh, psi, d, string):

    uLocal = uh.setLocalContribution()

    space = uh.space

    P = space.order

    # set all entries of u to zero
    uh.clear()

    if d == 2:

        count = []

        val = np.empty((P + 1, P + 1), dtype = int)

        for i in range(P + 1):
            for j in range(P + 1):

                count.append([i, j])

                # values for uLocal indices
                if j < i:
                    val[i, j] = i**2 + j
                elif j >= i:
                    val[i, j] = j * (j + 1) + i

        for e in space.grid.elements:

            # bind uLocal to current element
            uLocal.bind(e)

            f_hat = np.empty((P + 1, P + 1))

            # loop over all quadrature points
            for index, coord in enumerate(quadratureRule(e.type, 2 * P)):

                p, q = count[index]

                # f-evaluations
                f_hat[p, q] = fct(e, coord.position)

            if string == 'looping':

                for i in range(P + 1):
                    for j in range(P + 1):

                        uLocal[val[i, j]] = \
                            sum(psi[q, i] * psi[p, j] * f_hat[p, q] \
                                for p in range(P + 1) for q in range(P + 1))

            uLocal.unbind()
```

6.3.2 Dune implementation

This is the implementation that uses the way Dune solves the L^2 projection. The difference between the Dune implementation and the previously discussed tensor product implementation, is that the Dune implementation uses the values of $\phi_{..}$, which reduces the number of multiplications performed. For this implementation, we are only required to store $\hat{f}$ as a vector. We call this $\bar{f} = \mathbf{vec}(\hat{f})$.

The algorithm for this implementation is straightforward. We perform the computation with the code

Python code

```
def dune(fct, uh, phi, string):

    uLocal = uh.setLocalContribution()

    space = uh.space

    P = space.order

    Q = len(phi)

    # set all entries of u to zero
    uh.clear()

    for e in space.grid.elements:

        # bind uLocal to current element
        uLocal.bind(e)

        f_bar = np.empty(Q)

        # loop over all quadrature points
        for p, coord in enumerate(quadratureRule(e.type, 2 * P + 1)):

            # f-evaluations
            f_bar[p] = fct(e, coord.position)

        if string == 'looping':

            for i in range(Q):

                uLocal[i] = sum(phi[p, i] * f_bar[p] for p in range(Q))

        uLocal.unbind()
```

Mathematically, we are computing the formula

$$\hat{u}_i = \sum_{p=0}^{(P+1)^d-1} \phi_{pi} \bar{f}_p. \tag{6.11}$$

One can see that for each computed point of $\hat{u}_i$, there are fewer multiplications performed, compared to the full tensor product. It is also worth noting, that the other implementations are based on the Dune implementation. In particular, no extra code is needed for ordering, such as *val*. For this reason, the Dune implementation is a fairly straightforward algorithm.

6.3.3 Sum-factorization

The sum-factorization implementation is theoretically the best of the three implementations. The way this implementation works is a little bit more complicated than the previous implementations. The way $\hat{u}$ is evaluated, for two dimensions, is by the formula

$$A_p = \sum_{q=0}^{P} \psi_{qi} F_{pq}, \tag{6.12}$$

$$\hat{u}_{val_{ij}} = \sum_{p=0}^{P} \psi_{pj} A_p. \tag{6.13}$$

.
For three dimensions, it is evaluated by

$$A_{pq} = \sum_{r=0}^{P} \psi_{ri} F_{pqr}, \tag{6.14}$$

$$B_p = \sum_{q=0}^{P} \psi_{qj} A_{pq}, \tag{6.15}$$

$$\hat{u}_{e,val_{ijk}} = \sum_{p=0}^{P} \psi_{pk} B_p. \tag{6.16}$$

What makes this algorithm effective, especially for three dimensions, is the fact that when we change k and let i and j be intact, we do not need to recompute A or B. The same is true for when we change j and let i be the same, we do not need to recompute A. This drastically decreases the number of required operations. The way this is computed for two dimensions, in Python is demonstrated below.

Python code

```
def sum_factorization(fct, uh, psi, d, string):

    uLocal = uh.setLocalContribution()

    space = uh.space

    P = space.order

    # set all entries of u to zero
    uh.clear()

    if d == 2:

        count = []

        val = np.empty((P + 1, P + 1), dtype = int)

        for i in range(P + 1):
            for j in range(P + 1):

                count.append([i, j])

                # values for uLocal indices
                if j < i:
                    val[i, j] = i**2 + j
                elif j >= i:
                    val[i, j] = j * (j + 1) + i

        for e in space.grid.elements:

            # bind uLocal to current element
            uLocal.bind(e)

            f_hat = np.empty((P + 1, P + 1))

            # loop over all quadrature points
            for index, coord in enumerate(quadratureRule(e.type, 2 * P + 1)):

                p, q = count[index]

                # f-evaluations
                f_hat[p, q] = fct(e, coord.position)

            if string == 'looping':

                for i in range(P + 1):

                    A = np.empty(P + 1)

                    for p in range(P + 1):

                        A[p] = sum(psi[q, i] * f_hat[p, q] \
                                   for q in range(P + 1))

                    for j in range(P + 1):

                        uLocal[val[i, j]] = sum(psi[p, j] * A[p] \
                                                for p in range(P + 1))

            uLocal.unbind()
```

6.3.4 Discussion about approaches

So far, only algorithms with looping approach have been discussed. Unfortunately, looping works poorly in Python. For this reason, other approaches have been tested to speed up computation. The best approach found, is `numpy`'s builtin matrix-vector multiplication. In Chapter 6.5, it is shown that this approach is

faster than looping in Python, for any implementation. The code of the sum-factorization implementation is sufficient to get an idea how to code the other implementations with the matrix-vector approach. The code for using matrix-vector multiplication in dimension two, is performed by

Python code

```
if string == 'matvec':

    for i in range(P + 1):

    A = f_hat @ psi.T[i]

    for j in range(P + 1):

        B = A @ psi.T[j]

        for k in range(P + 1):

            uLocal[val[i, j, k]] = B @ psi.T[k]
```

This shows the difference when we call string as `matvec`, instead of `looping`, when the input for `sum_factorization` is chosen.

6.4 Error estimation

We now want to verify the approximation of u_h. What we can say about u_h, in mathematical sense, is that this approximation is exact of order P, i.e., the polynomial degree of u_h. The error of u_h is defined by

$$u_h - f \approx \frac{Cx^{P+1}}{(P+1)!}, \tag{6.17}$$

see for example [4, p.183].

We want to compute the absolute error of the approximation using the L^2-norm. This is because when L^2 projection is used, this is a best approximation in the L^2-norm. The definition of the error is

$$\text{error} = \sqrt{\int_\Omega (u_h - f)^2 dx}. \tag{6.18}$$

We want to compute this error with respect to $h = (b-a)/N$. First, we compute

$$\int_\Omega (u_h - f)^2 dx \approx \frac{NC^2}{((P+1)!)^2} \int_0^h x^{2P+2} dx = \frac{(b-a)C^2}{((P+1)!)^2} \int_0^h \frac{x^{2P+2}}{h} dx. \tag{6.19}$$

We use this to show that the error, given the L^2 norm, is

$$\begin{aligned} error \approx \sqrt{\frac{(b-a)C^2}{((P+1)!)^2} \int_0^h \frac{x^{2P+2}}{h} dx} = \sqrt{\frac{b-a}{2P+3} \Big(\frac{Ch^{P+1}}{(P+1)!} \Big)^2} = \\ \frac{Ch^{P+1}}{(P+1)!} \sqrt{\frac{b-a}{2P+3}}. \end{aligned} \tag{6.20}$$

After experimenting, we get that the test function, which we assume to be an upper bound of the actual error, is

$$\text{test} = \frac{2(b-a)}{(P+1)!\sqrt{2P+3}} h^{P+1}. \tag{6.21}$$

The code used to approximate the error, compared to the exact solution, is

Python code

```
def L2error(fct, uh, f, gridView, P):

    fct = uflFunction(gridView, name = 'f', order = P, ufl = [f])

    err = uflFunction(gridView, name = 'err', order = P, \
                      ufl = uh - fct)

    L2err = np.sqrt(integrate(gridView, dot(err, err), order = 2 * P + 3))

    return L2err
```

This code gives the absolute error, given an approximation of `uh`. The error, for given values of a, b, P and d, with various values of h is plotted below. Because the factor h^{P+1} has an exponent, it is appropriate to use a loglog plot, to plot the error. We use the test function, and compare it with the error we get from `L2error`. The error plot for $a = 0$, $b = 4$, $d = \{2, 3\}$ is shown below

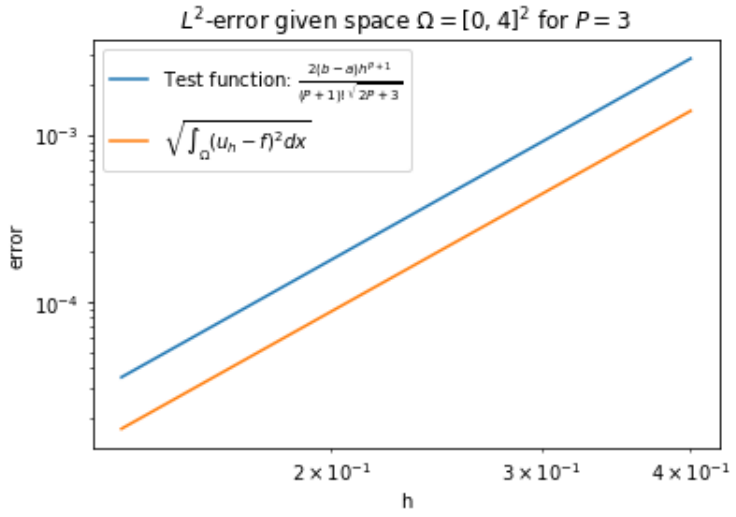

Figure 2: Error of the L^2 projection u_h, given $f(x, y) = \sin(\pi x)\sin(\pi y)$, over the domain $\Omega = [0, 4] \times [0, 4]$ and polynomial degree 3. The plot shows that the error is decreased for decreasing h and is bounded from above by the test function.

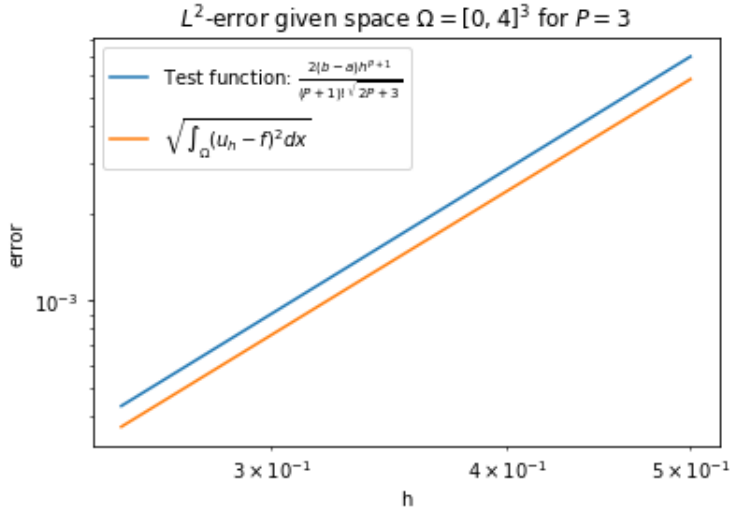

Figure 3: Error of the L^2 projection u_h, given $f(x, y, z) = sin(\pi x)sin(\pi y)sin(\pi z)$, over the domain $\Omega = [0, 4] \times [0, 4] \times [0, 4]$ and polynomial degree 3. The plot shows that the error is decreased for decreasing h and is bounded from above by the test function.

The error of the approximation is what one can expect. We have the same slopes of both the L^2 error and the test function, given h. We also have that the L^2 error is bounded from above by the test function.

6.5 Computation time

It is now verified that we have well approximated solutions. Now we can start testing computation times for all implementations. In Python, there are many ways to time how quickly an algorithm is performed. The timing method that will be used for this project is called `timeit` [1]. This tool is imported by the code:

Python code

```
import timeit
```

One way to measure the computation time of an implementation is to define within the code of the implementation algorithm where the timing will start and where it will end, and compute the time difference of when the timing started and timing ended. This will not be the method of choice in this case. Since we will always compute the time for a full algorithm and not just parts of it, we rather want to use a timing method for this purpose. The method of choice is to create a string of the algorithm and import all used input of the algorithm. The way to compute the computation times for the algorithms, is by the code

Python code

```
N_vals = []

app1 = 'looping'
app2 = 'matvec'
```

```
method = np.array([['dune(fct, uh, phi_Vals, app1)', \
                    'from __main__ import dune, fct, uh, phi_Vals, app1'], \
                   ['tensor_product(fct, uh, psi_Vals, d, app1)', \
                    'from __main__ import tensor_product, fct, uh, \
                        psi_Vals, d, app1'],
                    ['sum_factorization(fct, uh, psi_Vals, d, app1)', \
                     'from __main__ import sum_factorization, fct, uh, \
                         psi_Vals, d, app1'], \
                    ['dune(fct, uh, phi_Vals, app2)', \
                     'from __main__ import dune, fct, uh, phi_Vals, app2'], \
                    ['tensor_product(fct, uh, psi_Vals, d, app2)', \
                     'from __main__ import tensor_product, fct, uh, \
                         psi_Vals, d, app2'], \
                    ['sum_factorization(fct, uh, psi_Vals, d, app2)', \
                     'from __main__ import sum_factorization, fct, uh, \
                         psi_Vals, d, app2']])

dune_looping_time = []
tp_looping_time = []
sf_looping_time = []

dune_matvec_time = []
tp_matvec_time = []
sf_matvec_time = []

for i in range(P_min, P_max + 1):

    dune_looping_time_k = []
    tp_looping_time_k = []
    sf_looping_time_k = []

    dune_matvec_time_k = []
    tp_matvec_time_k = []
    sf_matvec_time_k = []

    N_vals.append(i)

    fct, uh, f, gridView = spaces(i, N, d, a, b)

    phi_Vals = basisfunctions(i, d)

    psi_Vals = basisfunctions_1d(i)

    for k in range(count):

        dune_looping_time_k.append(timeit.timeit(method[0, 0], method[0, 1], \
                                                 number = num) / num)

        tp_looping_time_k.append(timeit.timeit(method[1, 0], method[1, 1], \
                                               number = num) / num)

        sf_looping_time_k.append(timeit.timeit(method[2, 0], method[2, 1], \
                                               number = num) / num)

        dune_matvec_time_k.append(timeit.timeit(method[3, 0], method[3, 1], \
                                                number = num) / num)

        tp_matvec_time_k.append(timeit.timeit(method[4, 0], method[4, 1], \
                                              number = num) / num)

        sf_matvec_time_k.append(timeit.timeit(method[5, 0], method[5, 1], \
                                              number = num) / num)

    dune_looping_time.append(np.mean(dune_looping_time_k))
    tp_looping_time.append(np.mean(tp_looping_time_k))
    sf_looping_time.append(np.mean(sf_looping_time_k))

    dune_matvec_time.append(np.mean(dune_matvec_time_k))
    tp_matvec_time.append(np.mean(tp_matvec_time_k))
    sf_matvec_time.append(np.mean(sf_matvec_time_k))
```

This is the way the computation times for the algorithms are computed. We want to test the computation times for the three different implementations that have previously been discussed, with looping and matrix-vector approaches. It is advised to use a looping approach over matrix-vector multiplication [5, p. 5169]. In Python however, we see that the matrix-vector approach in general is much faster, than using the looping approach.

We have discussed what to test and how the test is performed. We will now see the computation times for all algorithms. The plot of the two dimensional case and three dimensional cases for a given N and various values of P is included below

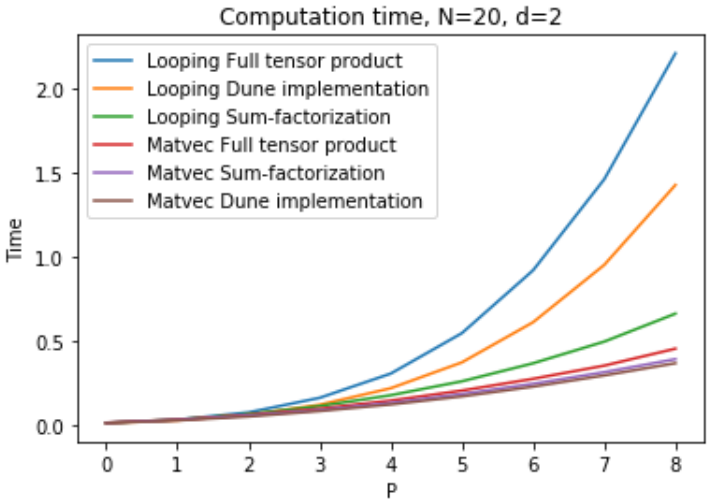

Figure 4: Computation times for each performed algorithm, for two dimensions and 20 grids per axis. The time is measured for various polynomial degrees.

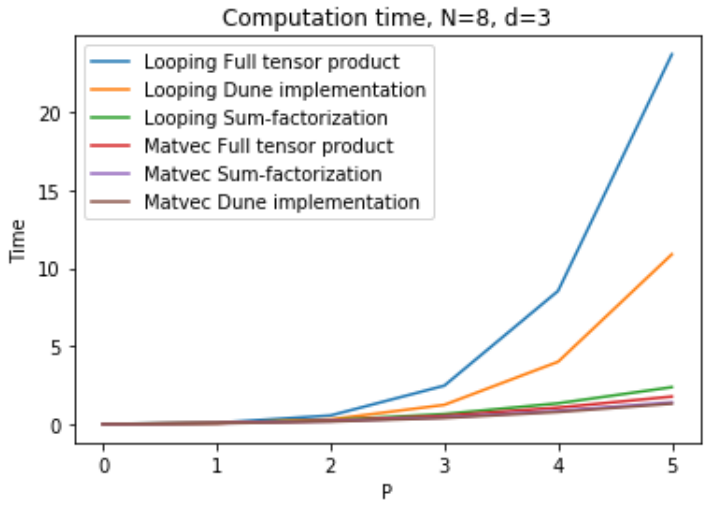

Figure 5: Computation times for each performed algorithm, for three dimensions and eight grids per axis. The time is measured for various polynomial degrees.

One sees that for the matrix-vector approach, we have that all three implementations almost have the same computation times. If we compare the implementations using the looping approach, we see that the sum-factorization implementation gets comparatively better than the other implementations for large P and for higher dimension. This can be seen by the number of operations for each implementation. For the full tensor product there is roughly $C_1(P+1)^{2d}$ number of operations. For the Dune implementation it is $C_2(P+1)^{2d}$, where C_1 is larger than C_2. Lastly, we have that sum-factorization have $C_3(P+1)^{d+1}$ number of operations. If we look at the three dimensional case, we see that the looping sum-factorization implementation, is almost as quick to compute than matrix-vector approaches. This shows that sum-factorization can be efficient in a programming language, where looping is faster than in Python.

7 Summary

The goal of this project was to show that for the aspects of the Discontinuous Galerkin Methods, that have been presented in this paper, sum-factorization is an effective implementation in Python. In Chapter 6, it was demonstrated that in Python, the sum-factorization implementation was fast in comparison with the Dune implementation and the full tensor product implementation. When the looping approach is used, sum-factorization is by far the fastest implementation. Unfortunately, it is not quite as effective to use with `numpy`'s matrix-vector command. The sum-factorization is still a sufficiently good implementation even when looping. In the plot of the computation times, which can be seen in Chapter 6.5, we can see that the sum-factorization of any approach is about as fast as all implementations of the matrix-vector approach. The conclusion of this is that the way to use sum-factorization is likely to use a different programming language where looping performs well. In conclusion it can be said that the implementation of sum-factorization performs well overall and gives results that are satisfactory.

8 Appendix

The appendix is used to show all of the code that is used for the project. Most of the code is discussed and shown in Section 6. A big thanks goes out to Robert Klöfkorn for setting up some of the code. Especially the imported tools from dune.

Python code

```
import os, io
# ensure some compilation output for this example
os.environ['DUNE_LOG_LEVEL'] = 'info'
print("Using DUNE_LOG_LEVEL=",os.getenv('DUNE_LOG_LEVEL'))

from ufl import SpatialCoordinate, pi, sin, dot

from dune.generator import algorithm
from dune.geometry import quadratureRule
from dune.grid import structuredGrid as leafGridView
from dune.fem.space import dglegendre
from dune.fem.function import uflFunction, integrate

import numpy as np
import matplotlib.pyplot as plt
import timeit

_code = \
"""
#ifndef FEM_EVAL_BASIS_HH
#define FEM_EVAL_BASIS_HH
#include <vector>
#include <array>
#include <utility>
#include <dune/python/pybind11/numpy.h>
#include <dune/common/fvector.hh>
#include <dune/common/dynvector.hh>

template <class Space, class Entity, class Point>
std::vector< typename Space::RangeType >
evaluateBasis(const Space& space, const Entity& entity, const Point& x)
{
  const auto basisSet = space.basisFunctionSet( entity );
  std::vector< typename Space::RangeType > basis( basisSet.size() );
  basisSet.evaluateAll( x, basis );
  return basis;
}
#endif

"""

def dune(fct, uh, phi, string):

    uLocal = uh.setLocalContribution()

    space = uh.space

    P = space.order

    Q = len(phi)

    # set all entries of u to zero
    uh.clear()

    for e in space.grid.elements:

        # bind uLocal to current element
        uLocal.bind(e)

        f_bar = np.empty(Q)

        # loop over all quadrature points
```

```
        for p, coord in enumerate(quadratureRule(e.type, 2 * P + 1)):

            # f-evaluations
            f_bar[p] = fct(e, coord.position)

        if string == 'looping':

            for i in range(Q):

                uLocal[i] = sum(phi[p, i] * f_bar[p] for p in range(Q))

        elif string == 'matvec':

            for i in range(Q):

                uLocal[i] = f_bar @ phi.T[i]

        uLocal.unbind()

def basisfunctions(P, d):

    gridView = leafGridView(d * [0], d * [1], d * [1])
    space = dglegendre(gridView, dimRange = 1, order = P)

    basisEval = None

    w = []

    basis_points = []

    for e in space.grid.elements:

        # loop over all quadrature points
        for p in quadratureRule(e.type, 2 * P + 1):

            # coordinate of quadrature point (\hat{x})
            x = p.position

            # weight point
            w.append(p.weight) # * geo.integrationElement(x)

            if basisEval is None:

                basisEval = algorithm.load('evaluateBasis', \
                                           io.StringIO(_code), space, e, x)

            # evaluate all basis function at the quadrature point
            basis_points.append(basisEval(space, e, x))

    phi_Vals = np.array(basis_points).reshape((P + 1)**d, (P + 1)**d) * \
        np.vstack(w)

    return phi_Vals

def tensor_product(fct, uh, psi, d, string):

    uLocal = uh.setLocalContribution()

    space = uh.space

    P = space.order

    # set all entries of u to zero
    uh.clear()

    if d == 1:

        for e in space.grid.elements:

            # bind uLocal to current element
            uLocal.bind(e)
```

```
            f_hat = np.empty(P + 1)

            # loop over all quadrature points
            for p, coord in enumerate(quadratureRule(e.type, 2 * P + 1)):

                # f-evaluations
                f_hat[p] = fct(e, coord.position)

            if string == 'looping':

                for i in range(P + 1):

                    uLocal[i] = \
                        sum(psi[p, i] * f_hat[p] for p in range(P + 1))

            elif string == 'matvec':

                for i in range(P + 1):

                    uLocal[i] = f_hat @ psi.T[i]

            uLocal.unbind()

    elif d == 2:

        count = []

        val = np.empty((P + 1, P + 1), dtype = int)

        for i in range(P + 1):
            for j in range(P + 1):

                count.append([i, j])

                # values for uLocal indices
                if j < i:
                    val[i, j] = i**2 + j
                elif j >= i:
                    val[i, j] = j * (j + 1) + i

        for e in space.grid.elements:

            # bind uLocal to current element
            uLocal.bind(e)

            f_hat = np.empty((P + 1, P + 1))

            # loop over all quadrature points
            for index, coord in enumerate(quadratureRule(e.type, 2 * P)):

                p, q = count[index]

                # f-evaluations
                f_hat[p, q] = fct(e, coord.position)

            if string == 'looping':

                for i in range(P + 1):
                    for j in range(P + 1):

                        uLocal[val[i, j]] = \
                            sum(psi[q, i] * psi[p, j] * f_hat[p, q] \
                                for p in range(P + 1) for q in range(P + 1))

            elif string == 'matvec':

                for i in range(P + 1):
                    for j in range(P + 1):

                        uLocal[val[i, j]] = (f_hat @ psi.T[i]) @ psi.T[j]

            uLocal.unbind()
```

```
    elif d == 3:

        count = []

        val = np.empty((P + 1, P + 1, P + 1), dtype = int)

        for i in range(P + 1):
            for j in range(P + 1):
                for k in range(P + 1):

                    count.append([i, j, k])

                    # values for uLocal indices
                    if k >= j and k >= i:
                        val[i, j, k] = k * ((k + 1)**2 + j) + i + j
                    elif k < i and j < i:
                        val[i, j, k] = i * (i**2 + 2 * k) + j + k
                    elif k < j and j >= i:
                        val[i, j, k] = j * (j**2 + 2 * k + 1) + i + k

        for e in space.grid.elements:

            # bind uLocal to current element
            uLocal.bind(e)

            f_hat = np.empty((P + 1, P + 1, P + 1))

            # loop over all quadrature points
            for index, coord in enumerate(quadratureRule(e.type, 2 * P)):

                p, q, r = count[index]

                # f-evaluations
                f_hat[p, q, r] = fct(e, coord.position)

            if string == 'looping':

                for i in range(P + 1):
                    for j in range(P + 1):
                        for k in range(P + 1):

                            uLocal[val[i, j, k]] = \
                                sum(psi[r, i] * psi[q, j] * psi[p, k] * \
                                    f_hat[p, q, r] for p in range(P + 1) \
                                        for q in range(P + 1) \
                                            for r in range(P + 1))

            elif string == 'matvec':

                for i in range(P + 1):
                    for j in range(P + 1):
                        for k in range(P + 1):

                            uLocal[val[i, j, k]] = \
                                ((f_hat @ psi.T[i]) @ psi.T[j]) @ psi.T[k]

            uLocal.unbind()

def sum_factorization(fct, uh, psi, d, string):

    uLocal = uh.setLocalContribution()

    space = uh.space

    P = space.order

    # set all entries of u to zero
    uh.clear()

    if d == 1:
```

```
        for e in space.grid.elements:

            # bind uLocal to current element
            uLocal.bind(e)

            f_hat = np.empty(P + 1)

            # loop over all quadrature points
            for p, coord in enumerate(quadratureRule(e.type, 2 * P + 1)):

                # f-estimates
                f_hat[p] = fct(e, coord.position)

            if string == 'looping':

                for i in range(P + 1):

                    uLocal[i] = \
                        sum(psi[p, i] * f_hat[p] for p in range(P + 1))

            elif string == 'matvec':

                for i in range(P + 1):

                    uLocal[i] = f_hat @ psi.T[i]

            uLocal.unbind()

    elif d == 2:

        count = []

        val = np.empty((P + 1, P + 1), dtype = int)

        for i in range(P + 1):
            for j in range(P + 1):

                count.append([i, j])

                # values for uLocal indices
                if j < i:
                    val[i, j] = i**2 + j
                elif j >= i:
                    val[i, j] = j * (j + 1) + i

        for e in space.grid.elements:

            # bind uLocal to current element
            uLocal.bind(e)

            f_hat = np.empty((P + 1, P + 1))

            # loop over all quadrature points
            for index, coord in enumerate(quadratureRule(e.type, 2 * P + 1)):

                p, q = count[index]

                # f-evaluations
                f_hat[p, q] = fct(e, coord.position)

            if string == 'looping':

                for i in range(P + 1):

                    A = np.empty(P + 1)

                    for p in range(P + 1):

                        A[p] = sum(psi[q, i] * f_hat[p, q] \
                                   for q in range(P + 1))

                    for j in range(P + 1):
```

```
                        uLocal[val[i, j]] = sum(psi[p, j] * A[p] \
                                                for p in range(P + 1))

            if string == 'matvec':

                for i in range(P + 1):

                    A = f_hat @ psi.T[i]

                    for j in range(P + 1):

                        uLocal[val[i, j]] =  A @ psi.T[j]

            uLocal.unbind()

    elif d == 3:

        count = []

        val = np.empty((P + 1, P + 1, P + 1), dtype = int)

        for i in range(P + 1):
            for j in range(P + 1):
                for k in range(P + 1):

                    count.append([i, j, k])

                    # values for uLocal indices
                    if k >= j and k >= i:
                        val[i, j, k] = k * ((k + 1)**2 + j) + i + j
                    elif k < i and j < i:
                        val[i, j, k] = i * (i**2 + 2 * k) + j + k
                    elif k < j and j >= i:
                        val[i, j, k] = j * (j**2 + 2 * k + 1) + i + k

        for e in space.grid.elements:

            # bind uLocal to current element
            uLocal.bind(e)

            f_hat = np.empty((P + 1, P + 1, P + 1))

            # loop over all quadrature points
            for index, coord in enumerate(quadratureRule(e.type, 2 * P + 1)):

                p, q, r = count[index]

                # f-evaluations
                f_hat[p, q, r] = fct(e, coord.position)

            if string == 'looping':

                for i in range(P + 1):

                    A = np.empty((P + 1, P + 1))

                    for p in range(P + 1):
                        for q in range(P + 1):

                            A[p, q] = sum(psi[r, i] * f_hat[p, q, r] \
                                          for r in range(P + 1))

                    for j in range(P + 1):

                        B = np.empty(P + 1)

                        for p in range(P + 1):

                            B[p] = sum(psi[q, j] * A[p, q] \
                                       for q in range(P + 1))
```

```
                        for k in range(P + 1):

                            uLocal[val[i, j, k]] = sum(psi[p, k] * B[p] \
                                                       for p in range(P + 1))

            elif string == 'matvec':

                for i in range(P + 1):

                    A = f_hat @ psi.T[i]

                    for j in range(P + 1):

                        B = A @ psi.T[j]

                        for k in range(P + 1):

                            uLocal[val[i, j, k]] = B @ psi.T[k]

            uLocal.unbind()

def basisfunctions_1d(P):

    gridView = leafGridView([0], [1], [1])
    space = dglegendre(gridView, dimRange = 1, order = P)

    basisEval = None

    w = []

    basis_points = []

    for e in space.grid.elements:

        # loop over all quadrature points
        for p in quadratureRule(e.type, 2 * P + 1):

            # coordinate of quadrature point (\hat{x})
            x = p.position

            # weight point
            w.append(p.weight) # * geo.integrationElement(x)

            if basisEval is None:
                basisEval = algorithm.load('evaluateBasis', \
                                           io.StringIO(_code), space, e, x)

            # evaluate all basis function at the quadrature point
            basis_points.append(basisEval(space, e, x))

    psi_Vals = np.array(basis_points).reshape(P + 1, P + 1) * np.vstack(w)

    return psi_Vals

######################################################################

#Fetching what we need from dune

def spaces(P, N, d, a, b):

    gridView = leafGridView(d * [a], d * [b], d * [N])
    space = dglegendre(gridView, dimRange = 1, order = P)

    x = SpatialCoordinate(space)
    f = sin(x[0] * pi)
    for i in range(1, d):
        f *= sin(x[i] * pi)

    fct = uflFunction(gridView, name = 'f', order = P, ufl = f)

    uh = space.interpolate([0], name = 'uh')
```

```
    return fct, uh, f, gridView

#Computes uh, with the implementations 'C', 'dune', 'tp' and 'sf' and
#approaches looping and matvec

def approx(P, N, d, a, b, implementation, approach):

    fct, uh, f, gridView = spaces(P, N, d, a, b)

    phi_Vals = basisfunctions(P, d)

    psi_Vals = basisfunctions_1d(P)

    if implementation == 'C':
        space = dglegendre(gridView, dimRange = 1, order = P)

        uh = space.interpolate(f, name = 'uh')

    elif implementation == 'dune':
        dune(fct, uh, phi_Vals, approach)

    elif implementation == 'tp':
        tensor_product(fct, uh, psi_Vals, d, approach)

    elif implementation == 'sf':
        sum_factorization(fct, uh, psi_Vals, d, approach)

    return uh

#Creating the error function

def L2error(fct, uh, f, gridView, P):

    fct = uflFunction(gridView, name = 'f', order = P, ufl = [f])

    err = uflFunction(gridView, name = 'err', order = P, \
                      ufl = uh - fct)

    L2err = np.sqrt(integrate(gridView, dot(err, err), order = 2 * P + 3))

    return L2err

#This plots the computed approximation for a given implementation

def approx_plt(P, N, d, a, b, implementation, approach):

    fct, uh, f, gridView = spaces(P, N, d, a, b)

    uh = approx(P, N, d, a, b, implementation, approach)

    if d == 1 or d == 2:
        uh.plot()

    elif d == 3:
        gridView.writeVTK('uh3d', pointdata = [uh])

#Computes the error of the approximation in the L2-norm

def approx_err(P, N, d, a, b, implementation, approach):

    fct, uh, f, gridView = spaces(P, N, d, a, b)

    uh = approx(P, N, d, a, b, implementation, approach)

    error = L2error(fct, uh, f, gridView, P)

    return error

#Creating lists of different grids

N2d = np.array([10, 20, 30])
N3d = np.array([8, 12, 16])
```

```
#Plot of the approximation errors

def approx_err_plt(P, N_vals, d, a, b, implementation, approach):

    h_vals = (b - a) / N_vals

    fact = np.math.factorial(P + 1)

    test = 2 * (b - a) * h_vals**(P + 1) / fact / (2 * P + 3)**(1 / 2)

    plt.loglog(h_vals, test, label = \
               r'Test function: $\frac{2(b-a)h^{P+1}}{(P+1)!\sqrt{2P+3}}$')

    error = [approx_err(P, i, d, a, b, implementation, approach) \
             for i in N_vals]

    plt.loglog(h_vals, error, label = '$\sqrt{\int_\Omega(u_h-f)^{2}dx}$')

    plt.title('$L^2$-error given space $\Omega=[{},{}]^{}$ for $P={}$'
              .format(a, b, d, P))

    plt.xlabel('h')
    plt.ylabel('error')
    plt.legend(loc = 'upper left')

#Measuring the computation time of each implementation

P_min = 0
P_max = 2

N = 20

d = 2

a = 0
b = 1

num = 25

count = 5

N_vals = []

app1 = 'looping'
app2 = 'matvec'

method = np.array([['dune(fct, uh, phi_Vals, app1)', \
                    'from __main__ import dune, fct, uh, phi_Vals, app1'], \
                   ['tensor_product(fct, uh, psi_Vals, d, app1)', \
                    'from __main__ import tensor_product, fct, uh, \
                        psi_Vals, d, app1'],
                    ['sum_factorization(fct, uh, psi_Vals, d, app1)', \
                     'from __main__ import sum_factorization, fct, uh, \
                         psi_Vals, d, app1'], \
                    ['dune(fct, uh, phi_Vals, app2)', \
                     'from __main__ import dune, fct, uh, phi_Vals, app2'], \
                    ['tensor_product(fct, uh, psi_Vals, d, app2)', \
                     'from __main__ import tensor_product, fct, uh, \
                         psi_Vals, d, app2'], \
                    ['sum_factorization(fct, uh, psi_Vals, d, app2)', \
                     'from __main__ import sum_factorization, fct, uh, \
                         psi_Vals, d, app2']])

dune_looping_time = []
tp_looping_time = []
sf_looping_time = []

dune_matvec_time = []
tp_matvec_time = []
sf_matvec_time = []
```

```
for i in range(P_min, P_max + 1):

    dune_looping_time_k = []
    tp_looping_time_k = []
    sf_looping_time_k = []

    dune_matvec_time_k = []
    tp_matvec_time_k = []
    sf_matvec_time_k = []

    N_vals.append(i)

    fct, uh, f, gridView = spaces(i, N, d, a, b)

    phi_Vals = basisfunctions(i, d)

    psi_Vals = basisfunctions_1d(i)

    for k in range(count):

        dune_looping_time_k.append(timeit.timeit(method[0, 0], method[0, 1], \
                                                 number = num) / num)

        tp_looping_time_k.append(timeit.timeit(method[1, 0], method[1, 1], \
                                               number = num) / num)

        sf_looping_time_k.append(timeit.timeit(method[2, 0], method[2, 1], \
                                               number = num) / num)

        dune_matvec_time_k.append(timeit.timeit(method[3, 0], method[3, 1], \
                                                number = num) / num)

        tp_matvec_time_k.append(timeit.timeit(method[4, 0], method[4, 1], \
                                              number = num) / num)

        sf_matvec_time_k.append(timeit.timeit(method[5, 0], method[5, 1], \
                                              number = num) / num)

    dune_looping_time.append(np.mean(dune_looping_time_k))
    tp_looping_time.append(np.mean(tp_looping_time_k))
    sf_looping_time.append(np.mean(sf_looping_time_k))

    dune_matvec_time.append(np.mean(dune_matvec_time_k))
    tp_matvec_time.append(np.mean(tp_matvec_time_k))
    sf_matvec_time.append(np.mean(sf_matvec_time_k))

    print(i)

def time_plt():

    plt.plot(N_vals, tp_looping_time, label = 'Looping Full tensor product')
    plt.plot(N_vals, dune_looping_time, label = 'Looping Dune implementation')
    plt.plot(N_vals, sf_looping_time, label = 'Looping Sum-factorization')

    plt.plot(N_vals, tp_matvec_time, label = 'Matvec Full tensor product')
    plt.plot(N_vals, sf_matvec_time, label = 'Matvec Sum-factorization')
    plt.plot(N_vals, dune_matvec_time, label = 'Matvec Dune implementation')

    plt.title('Computation time, N={}, d={}'.format(N, d))
    plt.xlabel('P')
    plt.ylabel('Time')
    plt.legend(loc = 'upper left')
```

References

[1] Python docs. timeit — Measure execution time of small code snippets, 2001. URL `https://docs.python.org/3/library/timeit.html`.

[2] Armin Iske. *Approximation Theory and Algorithms for Data Analysis*. Springer, 2018. doi: 10.1007/978-3-030-05228-7.

[3] Robert Klöfkörn et al. The Dune framework: Basic concepts and recent developments. *Elsevier*, 2020. doi: 10.1016/j.camwa.2020.06.007.

[4] Endre Süli and David Mayers. *An Introduction to Numerical Analysis*. Cambridge University Press, 2012. doi: 10.1017/CBO9780511801181.

[5] Peter E.J. Vos, Spencer J. Sherwin, and Robert M. Kirby. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *Journal of Computational Physics*, 2010. doi: 10.1016/j.jcp.2010.03.031.

Bachelor's Theses in Mathematical Sciences 2023:K13
ISSN 1654-6229

LUNFNA-4047-2023

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lu.se/