Array-oriented Understanding and Implementing of High order Finite Element Method in Python

Huayi Wei

March 6, 2020

Abstract

In this paper, we first present a general formula of the Lagrange basis functions on simplex mesh. Then we show how to compute the basis function and their gradients in array-oriented way. At last, we discuss the management of the degree of freedom.

1 Introduction

The finite element method (FEM) is a popular and important numerical method for solving partial differential equations (PDEs) which form the basis for many mathematical models in the physical sciences and many other fields.

There have been many papers and books discussing the theory of FEM, but most of them have paid little attention to the the implementation details of the FEM algorithm. Of course, there are already many excellent open source and commercial finite element software or package available. But for scientific computing students and researchers, simply using out-of-the-box software packages does not help them much more to in-depth understand and flexibly apply the finite element method.

Array is a very useful data structure which stores multiple data under single name with same data type. Array programming or array-oriented programming generalizes the operations on scalars to vectors, matrices, and higher-dimensional arrays. And notice that, array and its operaton are at the core of most algorithms in scientific numerical computing, such as FEM, Finite Diference Method (FDM), Finite Volume Method (FVM) and many others. So if we understand these algorithms in array-oriented way and, furthermore, have a optimized numerical linear algebra package at hand, we can easily describe scientific computing problem using very high-level code, and which is transparent and easy to maintain and extend.

There are already a lot of array programming languages, such as APL [4], J[3], Matlab [6], Python with Numpy extension [10] and so on. Matlab is widely used in academia and industry. One can find many attempts for array-oriented FEM programming in Matlab. Chen [1] developed an efficient MATLAB package, iFEM, for the adaptive finite element methods. Funken etc. developed a similar Matlab package p1afem [2]. Valdman etc. discussed the fast FEM matrix assembly for the nodal and edge element in [11, 7]. All of these efforts have attempted to improve the efficiency of scripting languages by leveraging the vectorization of Matlab to avoid the appearance of for-loops. However, the above work mainly focuses on the low-order finite element. In addition, Matlab is a closed-source commercial software with a quite poor base language which can become restrictive for advanced users.

Python is a well thought out programming language, allowing to write very readable and well structured code. It contains very rich scientific computing libraries and many other libraries, such as web server management, serial port access, etc. More importantly, it is free and open-source software, widely spread, and with a vibrant community. Many finite element software written in other languages provides a Python interface. FEniCS [5] is a sophisticated finite element simulation package with high-level Python and C++ interface, which enables users to quickly translate scientific models into efficient finite element code. FEniCS creates a separation of concerns between employing the finite element method and implementing it, which encapsulates the implementation details of the finite element algorithm . As a package similar to FEniCS, Firedrake [8] gives a new contribution to the automation and abstraction of the FEM, which introduce a new abstraction, PyOP2 [9], to create a separation within the implementation layer between the local discretization of mathematical operators and their parallel execution over the mesh. But such separation in using and implementing may hinders the user's understanding and flexibility in using the FEM. Especially for the researchers of finite element algorithms, it is very difficult to test and integrate the new finite element or similar algorithm.

In this paper, we first try to understand the finite element algorithm in an array-oriented way, especially for the high-order finite element, including the mesh data structure, basis function construction, degree of freedom management, matrix assembly. In the Python world, NumPy [10] arrays are the standard representation for numerical data and enable efficient implementation of numerical computations in a high-level language. Vectorizing calculations, avoiding copying data in memory, and minimizing operation counts are the three basic techniques for improving Numpy performance. Based on NumPy and the array-oriented understanding, we developed a simple, easy-to-use, and efficient high-order finite element package FEALPy:Finite Element Analysis Library in Python [12]. In addition to array-oriented, we have also designed FEALPy in an object-oriented way. Python has good support for object-oriented programming that allows us to organize FEALPy based on mathematical concepts in the finite element, such as mesh, degree of freedom, finite element space, finite element function.

2 Finite Element Method

In this section, let us discuss the finite element method in an array-oriented way using the Poisson equation as an example.

$$-\Delta u = f \text{ in } \Omega, u = 0 \text{ on } \partial\Omega \tag{1}$$

Where $\Omega \subset \mathbb{R}^d$. Using integration by parts, the weak form of the Poisson equation (1) is: find $u \in H_0^1(\Omega)$ such that:

$$a(u,v) = (f,v), \text{ for all } v \in H_0^1(\Omega), \tag{2}$$

where

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \mathrm{d}x, \quad (f,v) = \int_\Omega fv \mathrm{d}x.$$

Let $\mathcal{T}$ is a triangulation on Ω which is a composite of the d-simplexes. We define high-order finite element space on $\mathcal{T}$ as

$$V_p = \{v \in C(\bar{\Omega}) : v|_\tau \in \mathcal{P}_p, \forall \tau \in \mathcal{T}\} \tag{3}$$

where $\mathcal{P}_p$ is the polynomial space of degree p defined on each simplex τ. The key to constructing high-order finite element space V_p is to construct the high-order shape functions on each simplex, which can be constructed by the barycentric coordinates on d-simplex.

2.1 The Barycentric Coordinates on d-simplex

Let $\{\boldsymbol{x}_i := (x_{i,0}, x_{i,1}, \ldots, x_{i,d-1})\}_{i=0}^d$ be set of points in $\mathbb{R}^d$, and assume they are not lie in one hyper-plane, namely, the d vectors $\boldsymbol{x}_0\boldsymbol{x}_1$, $\boldsymbol{x}_0\boldsymbol{x}_2$, $\cdots$, and $\boldsymbol{x}_0\boldsymbol{x}_d$ are linear independent, which is equivalent to the following matrix:

$$\boldsymbol{A} = \begin{pmatrix} x_{0,0} & x_{1,0} & \cdots & x_{d,0} \\ x_{0,1} & x_{1,1} & \cdots & x_{d,1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{0,d-1} & x_{1,d-1} & \cdots & x_{d,d-1} \\ 1 & 1 & \cdots & 1 \end{pmatrix} \tag{4}$$

is non-singular.

Given any point $\boldsymbol{x} = (x_0, x_1, \cdots, x_{d-1})^T \in \mathbb{R}^d$, one can get a set of real values $\boldsymbol{\lambda} := (\lambda_0(\boldsymbol{x}), \lambda_1(\boldsymbol{x}), \cdots, \lambda_d(\boldsymbol{x}))^T$ by solving the following linear system:

$$\boldsymbol{A}\boldsymbol{\lambda} = \boldsymbol{x}, \tag{5}$$

which satisfy:

$$\boldsymbol{x} = \sum_{i=0}^d \lambda_i(\boldsymbol{x})\boldsymbol{x}_i, \text{ with} \sum_{i=0}^d \lambda_i(\boldsymbol{x}) = 1. \tag{6}$$

The convex hull of $\{\boldsymbol{x}_i\}_{i=0}^d$

$$\tau = \{\boldsymbol{x} = \sum_{i=0}^{d} \lambda_i \boldsymbol{x}_i | 0 \leq \lambda_i \leq 1, \sum_{i=0}^{d} \lambda_i = 1\} \tag{7}$$

is called a geometric d-simplex generated by $\{\boldsymbol{x}_i\}_{i=0}^d$. For example, an interval is a 1-simplex, a triangle is a 2-simplex, and a tetrahedron is a 3-simplex.

$\lambda_0(\boldsymbol{x})$, $\lambda_1(\boldsymbol{x})$, $\cdots$, and $\lambda_d(\boldsymbol{x})$ are the barycentric coordinates of $\boldsymbol{x}$ about $\{\boldsymbol{x}_i\}_{i=0}^d$. Furthermore, $\lambda_0(\boldsymbol{x})$, $\lambda_1(\boldsymbol{x})$, $\cdots$, and $\lambda_d(\boldsymbol{x})$ are linear functions about $\boldsymbol{x}$, and

$$\lambda_i(\boldsymbol{x}_j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}, i, j = 0, \cdots, d \tag{8}$$

2.2 The General Formula of High-order Lagrange Shape Functions

In this section, we discuss the construction of the Lagrange shape functions of degree $p \geq 1$ on the d-simplex. Let $\boldsymbol{m}$ be a $d+1$ multi-index vector $(m_0, m_1, \cdots, m_d)$ with

$$m_i \geq 0, i = 0, 1, \cdots, d, \text{ and } \sum_{i=0}^{d} m_i = p.$$

The number of all possible $\boldsymbol{m}$ is

$$n_p := \binom{d}{p+d}$$

Let α indicate a one-dimensional index starting from 0 to $n_p - 1$ of $\boldsymbol{m}$, and the Table 1 shows the numbering rules.

α	$\boldsymbol{m}_\alpha$				
0	p	0	0	$\cdots$	0
1	p-1	1	0	$\cdots$	0
2	p-1	0	1	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
d	p-1	0	0	$\cdots$	1
d+1	p-2	2	0	$\cdots$	0
d+2	p-2	1	1	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
2d-1	p-2	1	0	$\cdots$	1
2d	p-2	0	2	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n_p	0	0	0	$\cdots$	p

Table 1: The numbering rules of multi-index $\boldsymbol{m}_\alpha$.

Given the α-th multi-index $\boldsymbol{m}_\alpha$, one can construct a polynomial function of degree p on a d-simplex τ as following:

$$\phi_\alpha = \frac{1}{\boldsymbol{m}_\alpha!}\prod_{i=0}^{d}\prod_{j=0}^{m_i-1}(p\lambda_i - j) \tag{9}$$

with

$$\boldsymbol{m}_\alpha! = m_0!m_1!\cdots m_d!, \text{ and } \prod_{j=0}^{-1}(p\lambda_i - j) = 1, \quad i = 0, 1, \cdots, d$$

For every multiindex $\boldsymbol{m}_\alpha$, one can get a point $\boldsymbol{x}_\alpha$ in a d-simplex τ,

$$\boldsymbol{x}_\alpha = \sum_{i=0}^{d}\frac{m_i}{p}\boldsymbol{x}_i.$$

where m_i is the i-th component of $\boldsymbol{m}_\alpha$. It is easy to show that $\boldsymbol{x}_\alpha$ is the interpolation point of ϕ_α, and

$$\phi_\alpha(\boldsymbol{x}_\beta) = \begin{cases}1, & \alpha = \beta\\ 0, & \alpha \neq \beta\end{cases} \text{ with } \alpha, \beta = 0, 1, \cdots, n_p - 1 \tag{10}$$

Given a triangulation $\mathcal{T}$ in $\mathbb{R}^d$, one can construct the arbitray high order continuous piecewise polynomial space based on (9).

2.3 The Array-oriented Computing of ϕ_α and $\nabla\phi_\alpha$

First let's discuss how to use an array-oriented approach to evaluate the function values of ϕ_α on a barycentric point $(\lambda_0, \lambda_1, \cdots, \lambda_d)$.

First construct a vector

$$P = (\frac{1}{0!}, \frac{1}{1!}, \frac{1}{2!}, \cdots, \frac{1}{p!}),$$

and a matrix

$$A := \begin{pmatrix} 1 & 1 & \cdot & 1 \\ p\lambda_0 & p\lambda_1 & \cdots & p\lambda_d \\ p\lambda_0 - 1 & p\lambda_1 - 1 & \cdots & p\lambda_d - 1 \\ \vdots & \vdots & \ddots & \vdots \\ p\lambda_0 - (p-1) & p\lambda_1 - (p-1) & \vdots & p\lambda_d - (p-1) \end{pmatrix}, \tag{11}$$

then compute the cumulative product of every column elements of A and multiply a diagonal matrix formed by the vector P in the left side, one can get matrix

$$B = \text{diag}(P) \begin{pmatrix} 1 & 1 & \cdots & 1 \\ p\lambda_0 & p\lambda_1 & \cdots & p\lambda_d \\ \prod_{j=0}^{1}(p\lambda_0 - j) & \prod_{j=0}^{1}(p\lambda_1 - j) & \cdots & \prod_{j=0}^{1}(p\lambda_d - j) \\ \vdots & \vdots & \ddots & \vdots \\ \prod_{j=0}^{p-1}(p\lambda_0 - j) & \prod_{j=0}^{p-1}(p\lambda_1 - j) & \cdots & \prod_{j=0}^{p-1}(p\lambda_d - j) \end{pmatrix} \tag{12}$$

Notice that matrix B contain all the basic components of ϕ_α, then we can rewrite ϕ_α in (9) into:

$$\phi_\alpha = \prod_{i=0}^{d} B_{m_i,i}$$

where m_i is the i-th component of $\boldsymbol{m}_\alpha$.

Next, let's discuss how to use an array-oriented approach to evaluate the gradient values of ϕ_α on a barycentric point $(\lambda_0, \lambda_1, \cdots, \lambda_d)$. According the product rule of derivative, one first need to compute the gradient of $\prod_{j=0}^{m_i-1}(p\lambda_i - j)$ in (9), namely

$$\nabla \prod_{j=0}^{m_i-1}(p\lambda_i - j) = p \sum_{j=0}^{m_i-1} \prod_{0\le k\le m_i-1, k\ne j} (p\lambda_i - k)\nabla\lambda_i.$$

In array-oriented way, we first need to construct $d+1$ matrices

$$D^i = \begin{pmatrix} p & p\lambda_i & \cdots & p\lambda_i \\ p\lambda_i - 1 & p & \cdots & p\lambda_i - 1 \\ \vdots & \vdots & \ddots & \vdots \\ p\lambda_i - (p-1) & p\lambda_i - (p-1) & \cdots & p \end{pmatrix}, \quad 0 \le i \le d,$$

then get a matrix D with element

$$D_{i,j} = \sum_{m=0}^{j}\prod_{k=0}^{j} D^i_{k,m}, \quad 0 \le i \le d, \text{ and } 0 \le j \le p-1.$$

Next we can compute the gradient of matrix B as following

$$\begin{aligned}
\nabla B =& \mathrm{diag}(P)\begin{pmatrix} 0 & 0 & \cdots & 0 \\ D_{0,0}\nabla\lambda_0 & D_{1,0}\nabla\lambda_1 & \cdots & D_{d,0}\nabla\lambda_d \\ \vdots & \vdots & \ddots & \vdots \\ D_{0,p-1}\nabla\lambda_0 & D_{1,p-1}\nabla\lambda_1 & \cdots & D_{d,p-1}\nabla\lambda_d \end{pmatrix} \\
=& \mathrm{diag}(P)\begin{pmatrix} \mathbf{0} \\ D \end{pmatrix}\begin{pmatrix} \nabla\lambda_0 & & & \\ & \nabla\lambda_1 & & \\ & & \ddots & \\ & & & \nabla\lambda_d \end{pmatrix} \\
=& F\begin{pmatrix} \nabla\lambda_0 & & & \\ & \nabla\lambda_1 & & \\ & & \ddots & \\ & & & \nabla\lambda_d \end{pmatrix},
\end{aligned}$$

where

$$F = \mathrm{diag}(P)\begin{pmatrix} \mathbf{0} \\ D \end{pmatrix}. \tag{13}$$

3 The Array-oriented Programming of High Order Finite Element Method

3.1 Mesh

We need two basic array to represent a mesh. Take a triangle mesh as a example,

3.2 The Degrees of Freedom

Following the numbering rule of the multi-index in the Table 1, we can get the local numbering of the degrees of freedom on the d-simplex, please see Figures **??** and 3.

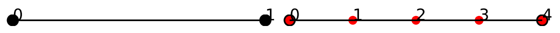

Figure 1: The local numbering of the interval vertices and the degrees of freedom of the basis function with $p = 4$.

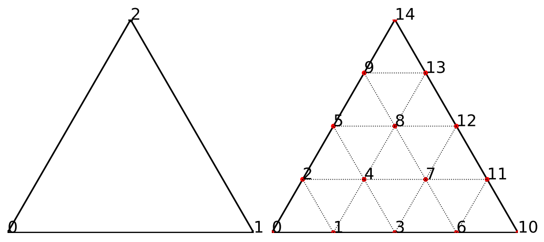

Figure 2: The local numbering of the triangle vertices and the degrees of freedom of the basis function with $p = 4$.

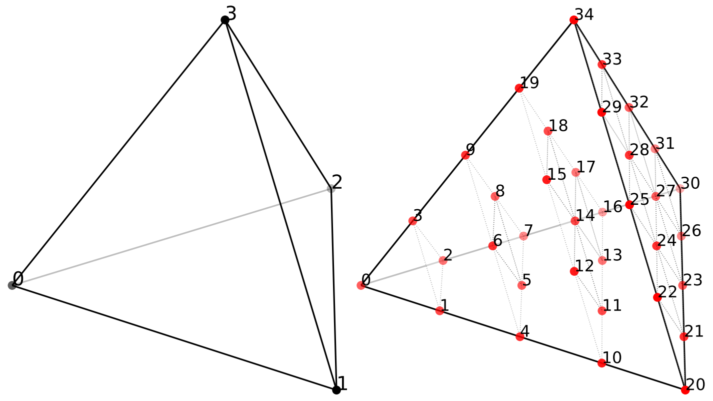

Figure 3: The local numbering of the tetrahedron vertices and the degrees of freedom for the basis functions with $p = 4$.

3.3 Lagrange Finite Element Space

In FEALPy, we design a LagrangeFiniteElementSapce class which can represent the continuous or discontinuous Lagrange finite element space in 1D, 2D or 3D Euclidean space. One can extend it to higher dimension Euclidean space by adding dofs management component on higher dimension simplex mesh.

Python code 1 The class of Lagrange Finite Element Space.

```python
class LagrangeFiniteElementSpace():
    def __init__(self, mesh, p=1, spacetype='C'):
        """
        The initialization function of the Lagrange Finite Element Space.

        Parameters
        ----------
        mesh : a mesh instance.
        p : a positive integer, the degree of the space.
        spacetype : the space type, 'C' means continuous space while 'D' means discontinuous space.
        """
        self.mesh = mesh
        self.p = p
        if spacetype is 'C':
            if mesh.meshType is 'interval':
                self.dof = CPLFEMDof1d(mesh, p)
                self.dim = 1
            elif mesh.meshType is 'tri':
                self.dof = CPLFEMDof2d(mesh, p)
                self.dim = 2
            elif mesh.meshType is 'tet':
                self.dof = CPLFEMDof3d(mesh, p)
                self.dim = 3
        elif spacetype is 'D':
            if mesh.meshType is 'interval':
                self.dof = DPLFEMDof1d(mesh, p)
                self.dim = 1
            elif mesh.meshType is 'tri':
                self.dof = DPLFEMDof2d(mesh, p)
                self.dim = 2
            elif mesh.meshType is 'tet':
                self.dof = DPLFEMDof3d(mesh, p)
                self.dim = 3
```

The following is the basis member function of LagrangeFiniteElementSpace which can compute the function value of ϕ_α at a set of barycentric points.

Python code 2 Compute ϕ_α

```
def basis(self, bc):
    """
    Compute the shape function gradient values at a set of barycentric points `bc`

    Parameters
    ----------
    bc : numpy.array, the shape of `bc` can be `(dim+1,)` or `(nb, dim+1)`,
    where `nb` is the number of barycentric points.

    Returns
    -------
    phi : numpy.array, the shape of `phi` can be `(ldof, )` or `(nb, ldof)`,
    where `ldof` is the number of basis functions with degree `p` on a simplex.
    """
    p = self.p   # The degree of polynomial basis function
    dim = self.dim # The dimension of Euclidean space
    multiIndex = self.dof.multiIndex # The multiindex matrix of $m_\alpha$

    # construct vector $P = (\frac{1}{1!}, \frac{1}{2!}, \cdots, \frac{1}{p!})$.
    c = np.arange(1, p+1, dtype=np.int)
    P = 1.0/np.multiply.accumulate(c)

    # construct the matrix $A$ in formula $(11)$.
    t = np.linspace(0, 1, p, endpoint=False)
    shape = bc.shape[:-1]+(p+1, dim+1)
    A = np.ones(shape, dtype=np.float)
    A[..., 1:, :] = bc[..., np.newaxis, :] - t.reshape(-1, 1)

    # construct matrix $B$ in formula $(12)$, and here we still use the memory of $A$
    np.cumprod(A, axis=-2, out=A)
    A[..., 1:, :] *= P.reshape(-1, 1)

    # compute $\phi_\alpha$
    phi = p**p*np.prod(A[..., multiIndex, [0, 1, 2]], axis=-1)
    return phi
```

Python code 3 Compute $\nabla\phi_\alpha$

```python
def grad_basis(self, bc):
        """
        Compute the shape function gradient values at a set of barycentric
        points `bc`.

        Parameters
        ----------
        bc : numpy.array, the shape of `bc` can be `(dim+1,)` or `(NQ, dim+1)`,
        where `NQ` is the number of barycentric points.

        Returns
        -------
        gphi : numpy.array, the shape of `gphi` can be `(NC, ldof, dim)`
        or `(NQ, NC, ldof, dim)`, where `NC` is the number of simplexes in
        triangulation, and `ldof` is the number of basis functions with
        degree `p` on a simplex.
        """
        p = self.p   # The degree of polynomial basis function
        dim = self.dim # The dimension of Euclidean space
        multiIndex = self.dof.multiIndex # The multiindex matrix of $m_\alpha$
        ldof = self.number_of_local_dofs()

        # construct the vector $P = (\frac{1}{1!}, \frac{1}{2!}, \cdots, \frac{1}{p!})$
        c = np.arange(1, p+1, dtype=np.int)
        P = 1.0/np.multiply.accumulate(c)

        # construct the matrix $A$ in formula (11)
        t = np.linspace(0, 1, p, endpoint=False)
        shape = bc.shape[:-1]+(p+1, dim+1)
        A = np.ones(shape, dtype=np.float)
        A[..., 1:, :] = bc[..., np.newaxis, :] - t.reshape(-1, 1)

        # construct the matrix $F$ in formula (13)
        FF = np.einsum('...jk, m->...kjm', A[..., 1:, :], np.ones(p))
        FF[..., range(p), range(p)] = 1
        np.cumprod(FF, axis=-2, out=FF)
        F = np.zeros(shape, dtype=np.float)
        F[..., 1:, :] = np.sum(np.tril(FF), axis=-1).swapaxes(-1, -2)
        F[..., 1:, :] *= P.reshape(-1, 1)

        # construct the matrix $B$ in formula (12), and here we still use the memory of $A$
        np.cumprod(A, axis=-2, out=A)
        A[..., 1:, :] *= P.reshape(-1, 1)

        # compute $\nabla\phi_\alpha$
        Q = A[..., multiIndex, list(range(dim+1))]
        M = F[..., multiIndex, list(range(dim+1))]
        shape = bc.shape[:-1]+(ldof, dim+1)
        R = np.zeros(shape, dtype=np.float)
        for i in range(dim+1):
            idx = list(range(dim+1))
            idx.remove(i)
            R[..., i] = M[..., i]*np.prod(Q[..., idx], axis=-1)
        Dlambda = self.mesh.grad_lambda()
        gphi = np.einsum('...ij, kjm->...kim', p**p*R, Dlambda)
        return gphi
```

3.4 Finite Element Matrix Assembly

Python code 4 The assembly code of stiffness matrix.

```python
def stiff_matrix(space, qf, area):
    bcs, ws = qf.quadpts, qf.weights
    gphi = space.grad_basis(bcs)
    A = np.einsum('i, ijkm, ijpm, j->jkp', ws, gphi, gphi, area)

    cell2dof = space.dof.cell2dof
    ldof = space.number_of_local_dofs()
    I = np.einsum('k, ij->ijk', np.ones(ldof), cell2dof)
    J = I.swapaxes(-1, -2)

    gdof = space.number_of_global_dofs()
    A = csr_matrix((A.flat, (I.flat, J.flat)), shape=(gdof, gdof))
    return A
```

Python code 5 The assembly code of mass matrix.

```python
def mass_matrix(space, qf, area):
    bcs, ws = qf.quadpts, qf.weights
    phi = space.basis(bcs)
    A = np.einsum('m, mj, mk, i->ijk', ws, phi, phi, area)

    cell2dof = space.dof.cell2dof
    ldof = space.number_of_local_dofs()
    I = np.einsum('k, ij->ijk', np.ones(ldof), cell2dof)
    J = I.swapaxes(-1, -2)

    gdof = space.number_of_global_dofs()
    A = csr_matrix((A.flat, (I.flat, J.flat)), shape=(gdof, gdof))
    return A
```

Python code 6 The assembly code of right-hand-side vector

```python
def source_vector(f, space, qf, area):
    bcs, ws = qf.quadpts, qf.weights
    pp = space.mesh.bc_to_point(bcs)
    fval = f(pp)
    phi = space.basis(bcs)
    b = np.einsum('i, ij, ik, k->kj', ws, phi, fval, area)

    cell2dof = space.dof.cell2dof
    gdof = space.number_of_global_dofs()
    b = np.bincount(cell2dof.flat, weights=b.flat, minlength=gdof)
    return b
```

4 The Numerical Examples

5 Conclusion and Future Work

References

[1] L. Chen. iFEM: an innovative finite element methods package in MATLAB. *Prepr. Maryland, 2008)*, pages 1–35, 2008.

[2] S. Funken, D. Praetorius, and P. Wissgott. Efficient implementation of adaptive P1-FEM in Matlab. *Comput. Methods Appl. Math.*, 11(4):460–490, 2011.

[3] R. H. Hui. An implementation of J, 1992.

[4] Kenneth E. Iverson. Notation as a Tool of Thought. *Commun. ACM*, 23(8):444–465, 1980.

[5] A. Logg, K.-A. Mardal, G. N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[6] MathWorks. MATLAB. https://www.mathworks.com/products/matlab.html.

[7] T. Rahman and J. Valdman. Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements. *Appl. Math. Comput.*, 219:7151–7158, 2013.

[8] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-t. Bercea, G. R. Markall, P. H. J. Kelly, and I. C. London. Firedrake : Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Softw.*, 43(3):24:1–24:27, 2016.

[9] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. J. Kelly. PyOP2 : A High-Level Framework for Performance- Portable Simulations on Unstructured Meshes. In *High Performance Computing, Networking Storage and Analysis, SC Companion*, pages 1116–1123, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[10] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Comput. Sci. Eng.*, 13(2):22–30, 2011.

[11] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Comput. Sci. Eng.*, 13(2):22–30, 2011.

[12] H. Wei. FEALPy: Finite Element Analysis Library in Python. https://github.com/weihuayi/fealpy.