NORTHWESTERN UNIVERSITY

A Generic Finite Element Solver By Meta-Expressions

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Mechanical Engineering

By

Jiaxi Xie

EVANSTON, ILLINOIS

June 2022

© Copyright by Jiaxi Xie 2022

All Rights Reserved

ABSTRACT

A Generic Finite Element Solver By Meta-Expressions

Jiaxi Xie

Finite Element Method (FEM) is a common simulation method in Computer Aided Engineering (CAE) or studying natural phenomenon by numerically solving Partial Differential Equations (PDEs). Programmable solvers for generic physics and geometry are thus highly desirable. However, the classical solvers are mostly complex systems in terms of both their mathematical complexity and lines of code, resulting in considerable entry cost for both new and experienced developers. Plus, a large portion of them do not support GPU-accelerated simulation.

This work makes two observations: 1) many finite element processes are actually about assembling similar mathematical structures, namely bilinear terms, in a repetitive manner, for which symbolic techniques such as rewriting are highly suitable for automation, and 2) a large portion of the actual computations take place at each integration point locally in a vectorized manner, for which GPU kernels can be conveniently generated. To exploit both ideas, we propose a skeleton generic FEM solver, named MetaFEM[**4**], in total about 6,000 lines of Julia code, which translates generic input PDE weak forms

into corresponding GPU-accelerated simulations with a grammar similar to FEniCS[**19**] or FreeFEM[**30**]. Two novel approaches differentiate MetaFEM from the common solvers: (1) the FEM kernel is based on an original theory/algorithm which explicitly processes meta-expressions, as the name suggests, and (2) the symbolic engine is a rule-based Computer Algebra System (CAS), i.e., the equations are rewritten/derived according to a set of rewriting rules instead of going through completely fixed routines, supporting easy customization by developers.

This article is a direct expansion of [**4**].

Acknowledgements

0, I need to thank my family, you are my motives.

1, I need to thank my advisor Prof. Jian Cao foremostly, who offered me the valuable/invaluable opportunity of a focused development for 5 years and supported me well-roundedly during it. As a result, I believe I indeed grew to be generally better and one can not expect more. I also need to equally thank my advisor Prof. Kornel Ehmann. One may say that to respect somebody is to learn from him, and I did learned a copious amount of skills and wisdoms. Finally, I need to thank my dissertation committee member Prof. Gregory Wagner, who is both technically sharp and generally helpful. We had many valuable discussions. I also need to specially thank Prof. Gianluca Cusatis, who supported me crucially at the final stage in my PhD career.

2, I need to thank several long-time personal friends of mine, I am fortunate to have your friendships.

3, I need to thank a not small number of colleagues. Firstly, I would like to thank Dr. Huaqing Ren, who guided me in my early study in many valuable and appreciable aspects. Secondly, I would like to thank Suman Bhandari, Dr. Yanming Zhang, Dr. Mojtaba Mozaffar, Shuheng Liao and Dr. Ashkan Gogoon, my helpful collaborators. Finally, I would also like to have a (incomplete) thank list for helping me furthering my study, i.e., Dr. Newell Moser, Dr. Qiang Zeng, Dr. Zixuan Zhang, Prof. Ping Guo, Dr. Fengchun Li, Dr. Yi Shi, Dr. Weizhao Zhang, Dr. Nicolas Prieto, Zilin Jiang, Dohyun

Leem, Samantha Webster, Ru Yang, Yaoke Wang, Marisa Bisram, Putong Kang and Sanjana Subramaniam.

4, I am generally proud of myself. Specifically for my PhD career, I did something new and usable.

Preface

The preface is supposed to be a personal record of the author's notable experience in his PhD career.

1. Everything is objective, such as the trueness of a statement, the causality, or the change/conservation of something.

2. We live in one specific world, so among all the parallel explanations/statements about something, one is true and the others are false.

3. Human creations are finite and purposeful, and usually not complex in the effective size, especially something without massive evolution.

4. Every human creation is thus practically reasonable, such as a decision, a design, or a result structure from iterative evolution.

TL;DR: Be honest; Make sense; Things are simple and learn them.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1. Motivation

Multi-physics FEM solvers are an important component of the fundamental infrastructure in a wide range of modern engineering and academic fields. Most common solvers, if not all, are complex. Their complexity can be roughly expressed in term of Lines Of Code (LOC), i.e., each solver typically has $> 10^5$ LOC, as seen by the examples in table 1.1. Naturally, it is tempting to have a compact, flat skeleton software that encompasses a large portion of major functions, i.e., solves a wide range of PDE systems.

Table 1.1. Codebase size of selected open-source multi-physics FEM solvers, more FEM package names (but not data) can be simply found, for example, in the list of Wikipedia [**3**].

Name	Module and Version	LOC (Comments Included)
FreeFEM [**30**]	4.9	166805 lines of C++ in /src
GOMA [**2**]	6.2	423285 lines of C in /src and /include
Elmer [**1**]	9.0	375613 lines of Fortran or C in /fem/src
FEniCS, [**19**] DOLFIN-toolchain	DOLFIN [**34**] 0.3.1	49895 lines of C++ in /cpp/dolfinx, 31004 lines of Python in /python/dolfinx
	FFC [**32**] 0.3.1	10035 lines of Python in /ffcx
	UFL [**20**] 2019.1.0	23255 lines of Python in /ufl
MOOSE [**36**]	0.9.0	175745 lines of C in /framework/src, (/modules uncounted)

Different multi-physics solvers have very different architectures and levels of generality. This paper focuses on a small specific type, namely generic FEM solvers, which process completely generic PDE weak forms. That is, the solver merely uses tensor symbols with (free or dumb) indices added, multiplied, or algebraically operated, without higher level concepts such as Navier-Stokes equation(s), Dirichlet boundary condition(s), or even advection and diffusion. More directly, the script of the physics should look just like the mathematical expressions someone would write on a blackboard, for example, in a continuum mechanics class.

Besides of the structure compactness and the symbolic support, GPU-acceleration is another suitable enhancement for performance, but has not been widely implemented in the traditional solvers. Exploiting the convenient abstraction in CUDA.jl [**23**], it will be nice to directly generate GPU kernels instead of CPU code. Now, having the above three demands in mind, we shall introduce the actual work.

1.2. Overview

In practice, physics takes place on geometry. From a software perspective, a geometric object is represented by a mesh, i.e., a list of control points and the corresponding connectivity information. A control point may traditionally be a sampled vertex, but it can also be an abstract value for geometry generation, such as a gradient value in the Hermite interpolations or a patch value in the Non-Uniform Rational B-Splines (NURBS) [**26**]. A physical state is represented by a mesh with each of its control points assigned some physical variables so that the actual physical field is implicitly represented via interpolation. A simulation is to update one physical state to another, i.e., update a vector of floating-point numbers representing all the physical variables. In FEM, the variable vector is updated by well established $\mathbf{Kx} = \mathbf{d}$ where $\mathbf{K}, \mathbf{d}$ are obtained via minimizing the residue in the Galerkin method. Therefore, a generic FEM solver is a software that takes

3 inputs, i.e., the physics, the mesh, and the physics-geometry assignment, and output the simulation, i.e., a list of the variable vectors. A typical workflow is provided in fig. 1.1.

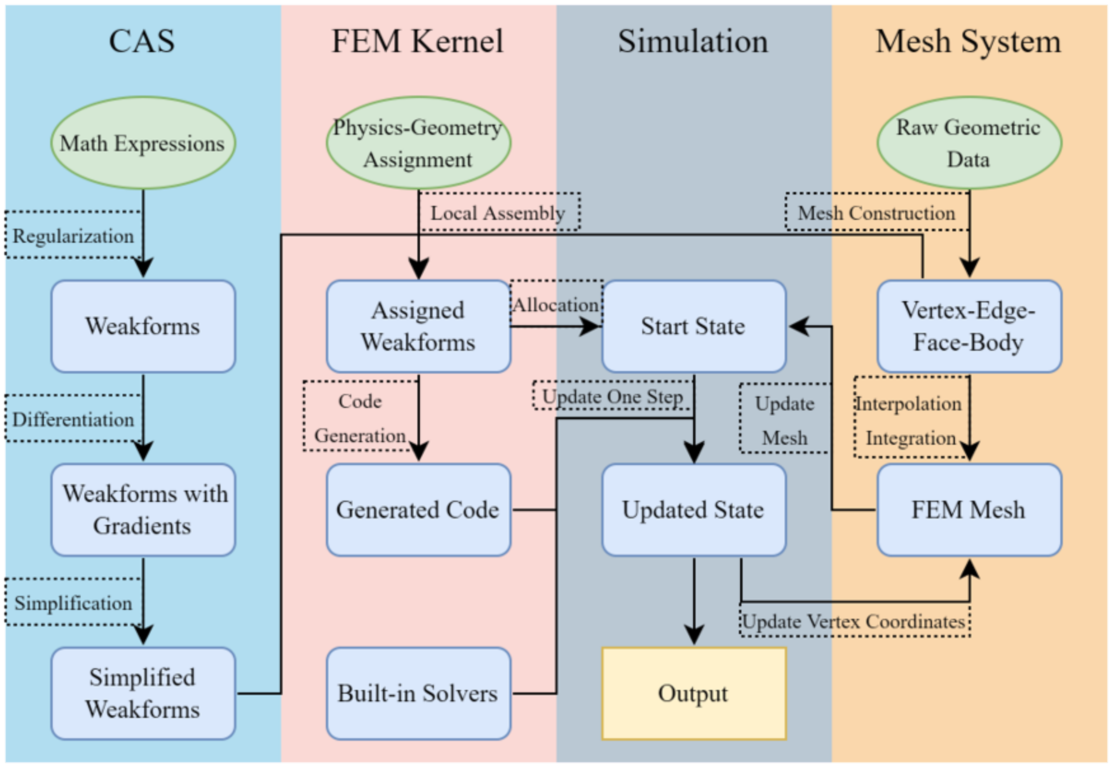


Figure 1.1. MetaFEM overview.

As in fig. 1.1 the solver consists of three components for the three inputs:

(1) A CAS collects and regularizes the mathematical expressions into weakforms, conducts any suitable differentiation and simplification, finally feeds the reorganized Intermediate Representations (IRs) to the FEM kernel for assembly. In a classical solver, such as FEniCS or FreeFEM, the CAS is usually completely hard-coded, which can be fundamentally improved by applying the theory of rewriting [**22**]. A general symbolic operation is a mapping between two symbolic

expressions and can be described by a third expression composed of the former two, denoted by a rewriting rule, which essentially establishes a mapping from data to operations. A symbolic processor is no more than a collection of elemental symbolic operations, and can be practically generated by parsing a set of simple rules, which is just a list of data. By interpreting the rewriting rules, the proposed formulation is flatter in structure, smaller in size and much more extensible for customization. The further discussion about CAS is provided in detail in chapter 3.

(2) A mesh system collects the raw geometric data, e.g., vertices and edges only, constructs the vertex-edge-face-body hierarchy and generate the FEM mesh, i.e., a mesh with high-order interpolations and integration points, then feed the FEM mesh into the FEM kernel for assembly. During a dynamic simulation the mesh system should adapt its mesh according to the current control point, including both the in-place update of interpolation values/integration weights or structural modification of the mesh connectivity, such as the adaptive refinement techniques.

(3) A FEM kernel assigns weakforms to meshes, assembles each mesh locally and then assembles every previously assigned weakform into a global system. Then, the kernel allocates and initializes the start state, and finally generates the corresponding code of the state updater in later simulation. The assembly process is inherently low-level and is discussed in detail in chapter 2. However, no matter what the actual formulation might be, the essential function is relatively fixed, i.e., the linear system $\mathbf{K}\mathbf{x} = \mathbf{d}$ is uniquely described by:

 (a) PDE weak forms (domain, boundary, stabilization);

(b) Linearization, e.g., complete gradient of nonlinear terms;

(c) Element type and order;

(d) Sufficient quadrature order or the same numerical integration scheme;

(e) Temporal discretization scheme; and

(f) The sequence (numbering) of variables.

which the kernel "simply" assembles.

Following the architecture in fig. 1.1, we have been developing MetaFEM, a compact open-source generic FEM solver of $\approx$ 6,000 lines of Julia [**24**] code, with the only dependence on CUDA.jl [**23**] for the GPU interface and some other Julia's intrinsic libraries. MetaFEM has both the rigorous mathematical formulation for the scalable future development and the already demonstrable practical utility. Since our formulation on the CAS and the FEW kernel are novel while the mesh system is mostly classical, we will focus only on the former two topics in the following chapters, i.e., we extensively discuss the FEM kernel in chapter 2, the CAS in chapter 3, the numerical examples in chapter 4 and the discussion in chapter 5.

1.3. State Of The Art (SOTA)

Before the discussion about the detailed implementation, it may be beneficial to provide an extended, practical comparison between MetaFEM and SOTA:

0. By 05/2022, there are 42 packages listed in Wikipedia [**3**];

1. 23 of them are proprietary, e.g., COMSOL, Abaqus, LS-DYNA, ANSYS, Siemens Nastran, Autodesk Simulation, MATLAB, Mathematica, etc.;

2. 3 out of the rest 19, Range3[**5**], Z88[**6**], Agros2D[**7**], do not have an accessible document by simple google search;

3. 9 out of the rest 16, OOFEM[**8**], CalculiX[**9**], GOMA, FEBio[**10**], Elmer, MOOSE, MoFEM JosePH[**11**], FreeCAD[**12**] and OpenSees[**13**] are fixed solvers or compositions of fixed solvers instead of generic solvers. Although we do not focus on the fixed solvers, we must emphasize that a fixed solver usually has better performance, while still being capable for a practically enough variety of physics;

4. 5 out of the rest 7, DUNE[**14**], deal.II[**15**], MFEM[**16**], Hermes[**17**], GetFEM++[**18**], are more toolboxes than solvers, which require relatively heavy user programming, e.g., through exposed cpp templates;

6. The final 2, FEniCS and FreeFEM, are the closest reference packages to MetaFEM, i.e., automatically generate FEM simulation (mainly) by the input weak form and a small collection of helper functions.

Specifically, MetaFEM is mostly similar to FEniCS in various aspects, where MetaFEM is much younger and lean while FEniCS is more mature and developed, and we make the detailed comparison as following:

1. The established benchmarks of physics are similar. The both solvers share a wide range of physics/mechanics and the tutorial examples in the both documents have a large content overlap. MetaFEM has plasticity while FEniCS has electromagnetics but they are only about the documentation development and essentially there is no reason one physical process can be solved in only one solver but not the other. In fact, the effective mechanisms of the both solvers are pretty similar, which can be observed by the hyper-elastic example where the both solvers utilize tensor differentiation for conciseness, despite

that the development histories are totally independent, i.e., the author of this article was foolishly unaware of the FEniCS project when he started MetaFEM.

2. The mesh system is the major undeveloped aspect of MetaFEM, where only (most of) the classical elements are supported, i.e., 2D/3D simplex/cubic Lagrange/serendipity elements while FEniCS can do DG, adaptive refinement, etc.

3. The symbolic processing is the major advantage of MetaFEM: A) MetaFEM allows easy customized symbolic processing by simply adding more rules in run time, e.g., doing a user-defined differentiation, while the symbolic derivation in FEniCS is fixed in compile time; and B) The MetaFEM has a more compact/smooth grammar and more flat structure in a variety of aspects, which can be observed by simply comparing the example scripts. Specifically, there is no helper functions like div, grad, Constant, dirichletbc, etc., no separation of the LHS/RHS and no VectorElement, since everything is in the component form, which is more aligned with the convention in the field of mechanics. Some practical examples can be observed from the 2 boundary condition cases in the FEniCSx tutorial: B-1, for the stacked/overlap different boundary conditions, one can simply write out different weak forms and assign them to the corresponding edges/faces in MetaFEM independently, and B-2, for a component-wise boundary condition where only some directions are constrained, the MetaFEM component form can do it straightforwardly.

4. Although we used LOC to approximate the system complexity and claiming MetaFEM is compact at beginning, one may argue that A) LOC can be a poor indicator and B) every software package starts in a compact structure but grows increasingly larger to accommodate more features, which are both true. A better description may be, MetaFEM utilizes a compact technique (rewriting) on a compact language (Julia),

resulting in 6000 LOC for all the above features, which may not be more compact than another solver, but is certainly a lightweight approach itself.

5. Finally, MetaFEM works on a single GPU (and requires CUDA) while FEniCS and most other solvers are classical (distributed) CPU solvers. We have not made a formal performance comparison because the current practical limit is more about the memory. A single GPU has very limited memory, e.g., 24 GB for NVIDIA GeForce RTX 3070, so distributed approaches are certainly needed but, have not been implemented yet.

CHAPTER 2

The FEM Kernel

To begin with, the notation for two different kinds of collections need to be clarified for the rest of the article. First, a physical vector is denoted by an arrow over the symbol, e.g., $\vec{u}$, and a component is subscripted in English $i, j, ...$ like u_i, with the Einstein summation convention, e.g., $u_i u_i \coloneqq \sum_{i=1}^{dim} u_i u_i$. Second, a general, variable-sized collection is enclosed in curly brackets like $\{\phi_\alpha\}$, and a component is subscripted in Greek $\alpha, \beta, ...$ like ϕ_α, without the Einstein summation convention. The collection size, which is also the maximum index, is denoted by a hat, e.g., $\hat{\alpha} \coloneqq \alpha_{max}$.

2.1. Theory

For illustration simplicity, we begin with a PDE system with one variable ϕ on one single compact manifold Ω with boundary $\partial\Omega$. From the engineering perspective, Ω is simply a workpiece assigned with some known physics.

Then, each PDE in Ω is limited to the following meta-expression:

$$\mathcal{L}(\phi) = \mathcal{L}^a(\partial_t^{\nu_1} D_1\phi, ..., \partial_t^{\nu_\lambda} D_\lambda\phi, ..., \partial_t^{\nu_{\hat{\lambda}}} D_{\hat{\lambda}}\phi) = 0 \tag{2.1}$$

where $\mathcal{L}$ is the overall operator of ϕ with the algebraic operator $\mathcal{L}^a$, which has the arity $\hat{\lambda}$, i.e., the number of operands. Note $\mathcal{L}^a$ can be nonlinear, where the letter L is used for a generic operator because we need the letter F in chapter 3. Each operand can be addressed by its index $\lambda = 1, 2, ..., \hat{\lambda}$. The λ^{th} operand has the $(\nu_\lambda)^{th}$ order temporal differential operator $\partial_t^{\nu_\lambda}$ and the spatial differential operator D_λ.

A weak solution ϕ^w (superscript w for weak) is a function which satisfies:

$$\int_\Omega \mathcal{L}(\phi^w)\delta\overline{\phi}^w = 0 \tag{2.2}$$

under an arbitrary smooth test function $\overline{\phi}^w$. Each specific $\overline{\phi}^w$ is independent of ϕ^w, but they will be always discussed in pairs, so the overline is used here and later to distinguish the symbols while emphasizing their duality.

The FEM, or more generally the minimum weighted residual method(s), is to find a discretized solution ϕ^h (superscript h following the customary notation in literature) which satisfies

$$\int_\Omega \mathcal{L}(\phi^h)\delta\overline{\phi}^h = 0 \tag{2.3}$$

under the arbitrary discretized test function $\overline{\phi}^h$. Each discretized function is a function which can be decomposed into the weighted sum of some predetermined interpolation functions:

$$\partial_t D\phi^h(\vec{x}, t) = \sum_{\alpha=1}^{\hat{\alpha}} (DN_\alpha(\vec{x}))(\partial_t \phi_\alpha(t)) \tag{2.4}$$

$$\delta(D\overline{\phi}^h(\vec{x}, t)) = \sum_{\overline{\alpha}=1}^{\hat{\alpha}} (D\overline{N}_{\overline{\alpha}}(\vec{x}))\delta\overline{\phi}_{\overline{\alpha}} \tag{2.5}$$

where N_α is the α^{th} shape function and $\overline{N}_{\overline{\alpha}}$ is the $\overline{\alpha}^{th}$ test function, with $\alpha, \overline{\alpha} = 1, 2, ..., \hat{\alpha}$. Shape functions and test functions are two predetermined collections of interpolation functions of size $\hat{\alpha}$, which span the base space and the dual space, respectively. The order of accuracy is determined by the complete polynomial order of the base space while the order of conservation is determined by the complete polynomial order of the dual space. When the dual space contains the base space, the scheme yields optimal convergence. Therefore, the most natural design is to choose $N_\alpha = \overline{N}_\alpha$ for each α, as in classical FEM, Discontinuous Galerkin (DG), etc., where the overline is usually simply omitted since no distinguishing is needed.

In classical FEM, each α is (the index of) a specific mesh node and ϕ_α is the value of the corresponding physical variable at that node. In general, each N_α represents a deformation mode with ϕ_α representing the corresponding deformation amplitude, named by a control point value, but may not be equal to any physical variable at a geometric point.

By directly extending eq. (2.1), a generic PDE system can be represented as a sum of bilinear forms, i.e., as:

$$d = \overbrace{(\cdot,\cdot)_\Omega + \ldots + (\cdot,\cdot)_\Omega}^{\text{domain physics}} + \overbrace{(\cdot,\cdot)_{\partial\Omega} + \ldots + (\cdot,\cdot)_{\partial\Omega}}^{\text{boundary conditions}} + \overbrace{(\cdot,\cdot)_\Omega + \ldots + (\cdot,\cdot)_\Omega}^{\text{numerical modifications}} \tag{2.6}$$

where the overall weak form d consists of the domain physics, the boundary conditions and the numerical modifications (in addition to the domain physics such as stabilizations), while each

$$(D_0\overline{\phi}, \mathcal{L}^a(\ldots, \partial_t^{\nu_\lambda} D_\lambda\phi, \ldots))_{\Omega'} \coloneqq \int_{\Omega'} \mathcal{L}^a(\ldots, \partial_t^{\nu_\lambda} D_\lambda\phi, \ldots)\delta D_0\overline{\phi} \tag{2.7}$$

is a single bilinear-form on $\Omega' = \Omega$ or $\partial\Omega$ with the dual word $D_0\overline{\phi}$ and the base term $\mathcal{L}^a(..., \partial_t^{\nu_\lambda} D_\lambda \phi, ...)$. A word is a symbol optionally with subscripts and a term is an expression tree formed by words. The details of words and terms will be discussed in chapter 3 and are skipped here.

Numerically, each bilinear form is explicitly approximated by numerical integration as:

$$(D_0\overline{\phi}, \mathcal{L}^a(..., \partial_t^{\nu_\lambda} D_\lambda \phi, ...))_\Omega \approx \tag{2.8}$$
$$\sum_\gamma w_\gamma^{\text{itg}} \mathcal{L}^a(..., \sum_{\alpha=1}^{\hat{\alpha}} (D_\lambda N_\alpha)|_{\vec{x}_\gamma^{\text{itg}}} (\partial_t^{\nu_\lambda} \phi_\alpha), ...) \sum_{\overline{\alpha}=1}^{\hat{\alpha}} (D_0 \overline{N}_{\overline{\alpha}})|_{\vec{x}_\gamma^{\text{itg}}} \delta\overline{\phi}_{\overline{\alpha}}$$

where $w_\gamma^{\text{itg}}, \vec{x}_\gamma^{\text{itg}}$ are the weight and position of the γ^{th} numerical integration point respectively.

With eq. (2.3), classical FEM minimizes each discretized component

$$d_{\overline{\alpha}}(\{\phi_\alpha\}) := \frac{\partial d}{\partial(\delta\overline{\phi}_{\overline{\alpha}})} \tag{2.9}$$

by the Newton-Raphson method. For a static problem, with an initial (guess or given) set of control point values $\{\phi_\alpha^0\}$, for each sub-step n (in the overall one single timestep) one has:

$$0 = d_{\overline{\alpha}}|_{\{\phi_\alpha^{n+1}\}} \approx \sum_{\alpha=1}^{\hat{\alpha}} \frac{\partial d_{\overline{\alpha}}}{\partial \phi_\alpha^n}|_{\{\phi_\alpha^n\}} (\phi_\alpha^{n+1} - \phi_\alpha^n) + d_{\overline{\alpha}}|_{\{\phi_\alpha^n\}} \tag{2.10}$$

where we denote

$$K_{\overline{\alpha}\alpha}^n := \frac{\partial d_{\overline{\alpha}}}{\partial \phi_\alpha^n} \tag{2.11}$$

to be the $(\overline{\alpha}, \alpha)^{th}$ component of the tangent stiffness of sub-step n, resulting in the linear system $\sum_{\alpha=1}^{\hat{\alpha}} K^n_{\overline{\alpha}\alpha}(\phi^{n+1}_\alpha - \phi^n_\alpha) = -d_{\overline{\alpha}}|_{\{\phi^n_\alpha\}}$, which is exactly the well established $\mathbf{Kx} = \mathbf{d}$ in classical literature. eq. (2.11) will be explicitly calculated in section 2.2.

For a dynamic problem, the above process needs to be extended according to the temporal discretization, and we choose the generalized-α scheme **[25]** with the maximum temporal derivative order $\hat{\nu} = 2$ for a simple but practical example.

At each timestep m, sub-step n, the collection of

$$\{\phi^{n+1}_{\alpha,m}, ...\} \coloneqq \{\phi^{n+1}_{\alpha,m}, u^{n+1}_{\alpha,m}, a^{n+1}_{\alpha,m}\} \tag{2.12}$$

needs to be determined as the basic variable, velocity and acceleration, where the term "basic" specifically emphasizes the symbol without time derivative, with the following constraints:

$$\Delta\phi^n_{\alpha,m} = \phi^{n+1}_{\alpha,m} - \phi^n_{\alpha,m}, \quad \Delta u^n_{\alpha,m} = u^{n+1}_{\alpha,m} - u^n_{\alpha,m}, \quad \Delta a^n_{\alpha,m} = a^{n+1}_{\alpha,m} - a^n_{\alpha,m} \tag{2.13}$$

$$\Delta\phi^n_{\alpha,m} = \Delta t(u^0_{\alpha,m} + b_1\Delta u^n_{\alpha,m}), \qquad \Delta u^n_{\alpha,m} = \Delta t(a^0_{\alpha,m} + b_2\Delta a^n_{\alpha,m}) \tag{2.14}$$

$$\widetilde{\phi}^n_{\alpha,m} = \phi^0_{\alpha,m} + c_1\Delta\phi^n_{\alpha,m}, \quad \widetilde{u}^n_{\alpha,m} = u^0_{\alpha,m} + c_2\Delta u^n_{\alpha,m}, \quad \widetilde{a}^n_{\alpha,m} = a^0_{\alpha,m} + c_3\Delta a^n_{\alpha,m} \tag{2.15}$$

where $\{\Delta\phi^n_{\alpha,m}, ...\}$ are the $(\hat{\nu}+1)\times\hat{\alpha}$ incremental values linked by $\hat{\nu}\times\hat{\alpha}$ constraints defined by the predefined constraint parameters b_1, b_2, so that the only $\hat{\alpha}$ degrees of freedom are exclusively the basic variables $\{\phi_\alpha\}$ but not its time derivatives, as in eq. (2.14). Meanwhile, $\{\widetilde{\phi}^n_{\alpha,m}, ...\}$ are the $(\hat{\nu}+1)\times\hat{\alpha}$ effective values at which the residues are evaluated with c_1, c_2 and c_3 being the relaxation parameters interpolating between the values at the last time step and the current (incremental) values, as in eq. (2.15).

By extending eq. (2.10), the corresponding linear system becomes:

$$\begin{aligned}
0 &= d_{\overline{\alpha}}|_{\{\widetilde{\phi}^{n+1}_{\alpha,m},\ldots\}} \qquad \text{for } \overline{\alpha} = 1, 2, \ldots, \hat{\alpha} \\
&\approx \sum_{\alpha=1}^{\hat{\alpha}} \left(\frac{\partial d_{\overline{\alpha}}}{\partial \widetilde{\phi}^n_\alpha}\frac{\partial \widetilde{\phi}^n_\alpha}{\partial \phi^n_\alpha} + \frac{\partial d_{\overline{\alpha}}}{\partial \widetilde{u}^n_\alpha}\frac{\partial \widetilde{u}^n_\alpha}{\partial \phi^n_\alpha} + \frac{\partial d_{\overline{\alpha}}}{\partial \widetilde{a}^n_\alpha}\frac{\partial \widetilde{a}^n_\alpha}{\partial \phi^n_\alpha}\right)|_{\{\widetilde{\phi}^n_{\alpha,m},\ldots\}}(\phi^{n+1}_{\alpha,m} - \phi^n_{\alpha,m}) + d_{\overline{\alpha}}|_{\{\widetilde{\phi}^n_{\alpha,m},\ldots\}} \\
&= \sum_{\alpha=1}^{\hat{\alpha}} \left(c_1\frac{\partial d_{\overline{\alpha}}}{\partial \widetilde{\phi}^n_\alpha} + \frac{c_2}{b_1\Delta t}\frac{\partial d_{\overline{\alpha}}}{\partial \widetilde{u}^n_\alpha} + \frac{c_3}{b_1 b_2(\Delta t)^2}\frac{\partial d_{\overline{\alpha}}}{\partial \widetilde{a}^n_\alpha}\right)|_{\{\widetilde{\phi}^n_{\alpha,m},\ldots\}}(\phi^{n+1}_{\alpha,m} - \phi^n_{\alpha,m}) + d_{\overline{\alpha}}|_{\{\widetilde{\phi}^n_{\alpha,m},\ldots\}}
\end{aligned} \tag{2.16}$$

and when $d_{\overline{\alpha}}|_{\{\widetilde{\phi}^n_{\alpha,m},\ldots\}}$ converges at $n = n_m$, one sets:

$$\{\phi^0_{\alpha,m+1}, \ldots\} := \{\phi^{n_m}_{\alpha,m}, \ldots\} \tag{2.17}$$

and the new timestep $m + 1$ starts.

2.2. Algorithm

This section demonstrates a more fine-grained algorithm that assembles multiple variables across different workpieces. For simplicity, we will still use the generalize-α temporal discretization and only show the process for the domain physics. In practice, the boundary conditions are just the domain physics one dimension lower and the numerical modifications are treated in the same way as the domain physics, so that the same process is essentially repeated another two times with different data.

The proposed FEM kernel consists of 4 Blocks, as shown in fig. 2.1, positioned from left to right and with abstraction levels from high to low as:

(1) Physics Assembly;

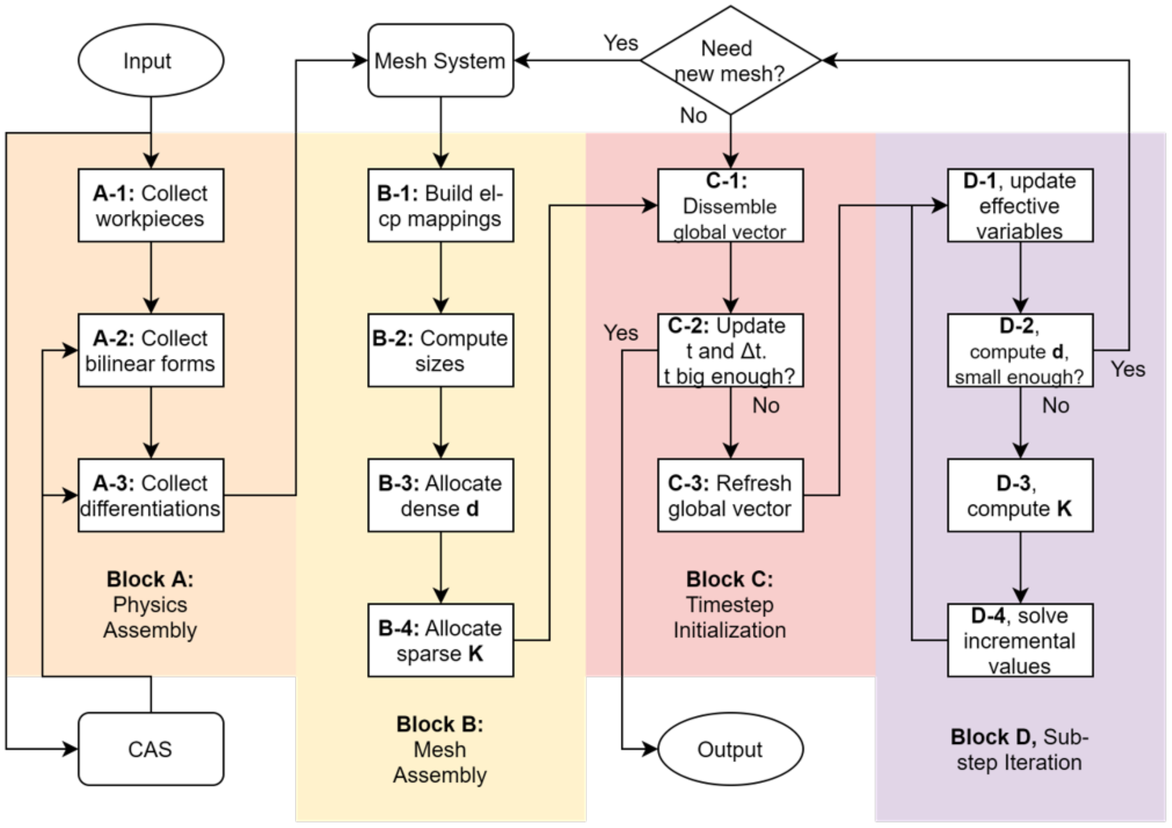

Input
Mesh System
Yes
Need new mesh?
No
A-1: Collect workpieces
A-2: Collect bilinear forms
A-3: Collect differentiations
Block A: Physics Assembly
CAS
B-1: Build el-cp mappings
B-2: Compute sizes
B-3: Allocate dense d
B-4: Allocate sparse K
Block B: Mesh Assembly
C-1: Dissemble global vector
Yes
C-2: Update t and Δt. t big enough?
No
C-3: Refresh global vector
Block C: Timestep Initialization
Output
D-1, update effective variables
D-2, compute d, small enough?
Yes
No
D-3, compute K
D-4, solve incremental values
Block D, Sub-step Iteration

Figure 2.1. FEM kernel flowchart

(2) Mesh Assembly;

(3) Timestep Initialization; and

(4) Sub-step Iteration;

which will be discussed one by one next.

Block A - *Physics Assembly* is completely symbolic and only runs right after the physics is assigned to the workpieces (with or without discretization to the mesh).

A-1, all workpieces are collected as $\{\Omega_{\beta^{wp}}\}$, $\beta^{wp} = 1, 2, ..., \hat{\beta}^{wp}$.

A-2, for each workpiece $\Omega_{\beta^{wp}}$, all the bilinear forms are collected as $\{B_{\beta^b}\}_{\beta^{wp}}$, where β^b is not tracked since the bilinear forms will be re-organized by the rewriting rules. Meanwhile, all the symbols of the basic variables are collected in $\{\phi^\kappa\}_{\beta^{wp}}$, with superscripted index $\kappa = 1, 2, ..., \hat{\kappa}_{\beta^{wp}}$ since the subscript is reserved for the control point index.

A-3, for each bilinear form in $\{B_{\beta^b}\}_{\beta^{wp}}$, i.e.,

$$(D_0\overline{\phi}^{\kappa_0}, \mathcal{L}^a(..., \partial_t^{\nu_\lambda} D_\lambda \phi^{\kappa_\lambda}, ...))_{\Omega_{\beta^{wp}}}$$

where the λ^{th} basic variable symbol index is specified by κ_λ, we collect the unique pairs $\{(\kappa_0, \kappa_\lambda)_{\beta^{sym\_pair}}\}_{\beta^{wp}}$ with index $\beta^{sym\_pair} = 1, 2, ..., \hat{\beta}^{sym\_pair}_{\beta^{wp}}$, for example, by integer hashing, to assemble the sparse matrix $\mathbf{K}$ later.

The operator $\mathcal{L}^a$ is also differentiated with respect to each operand (word, not symbol), denoted by $\frac{\partial \mathcal{L}^a}{\partial \lambda}$, resulting in the bilinear forms:

$$(D_0\overline{\phi}^{\kappa_0}, \frac{\partial \mathcal{L}^a}{\partial \lambda}(..., \partial_t^{\nu_\lambda} D_\lambda \phi^{\kappa_\lambda}, ...))_{\Omega_{\beta^{wp}}}$$

which are collected in $\{B^{diff}_{\beta^{b'}}\}_{\beta^{wp}}$ for computing the matrix value $\mathbf{K}$ later.

Block A ends above. Before **Block B**, the extension to multiple variables and multiple workpieces can be briefly summarized as:

(1) The residue d is modified to the sum of residues from all workpieces:

$$d = \sum_{\beta^{wp}=1}^{\hat{\beta}^{wp}} d_{\beta^{wp}} \tag{2.18}$$

(2) The global DOFs, denoted by α' and $\overline{\alpha}'$, are not only the workpiece control point indices $\alpha, \overline{\alpha}$, but are determined by a mapping from the workpiece, basic variable

and workpiece control point indices hierarchically as:

$$\alpha' := \alpha'(\beta^{wp}, \kappa, \alpha) \tag{2.19}$$

$$\overline{\alpha}' := \overline{\alpha}'(\beta^{wp}, \overline{\kappa}, \overline{\alpha}) \tag{2.20}$$

(3) The FEM is to minimize each discretized component in the global DOF, according to:

$$d_{\overline{\alpha}'(\beta^{wp},\overline{\kappa},\overline{\alpha})} := \frac{\partial d_{\beta^{wp}}}{\partial(\delta\phi^{\overline{\kappa}}_{\overline{\alpha}})} \tag{2.21}$$

Block B - *Mesh Assembly* runs after all the workpieces are discretized and whenever the mesh is changed.

B-1, for each workpiece $\Omega_{\beta^{wp}}$,

(1) The element indices are $\beta^{el} = 1, 2, ..., \hat{\beta}^{el}_{\beta^{wp}}$;

(2) The workpiece control point indices are $\alpha = 1, 2, ..., \hat{\alpha}_{\beta^{wp}}$;

(3) In each element β^{el}, the control points can be referred to by element to control point mapping $\alpha = \alpha(\beta^{el}, \beta^{el\_cp})$ where $\beta^{el\_cp} = 1, 2, ..., \hat{\beta}^{el\_cp}_{\beta^{el}}$ with $\hat{\beta}^{el\_cp}_{\beta^{el}}$ being the number of control points in this element;

(4) The workpiece control point index pairs $\{(\alpha_1, \alpha_2)_{\beta^{cp\_pair}}\}_{\beta^{wp}}$ are collected with $\beta^{cp\_pair} = 1, 2, ..., \hat{\beta}^{cp\_pair}_{\beta^{wp}}$ by variating the last input in each element to control point mapping, where each unique pair is only kept once, like the basic symbol pairs.

B-2, For each workpiece $\Omega_{\beta^{wp}}$, the workpiece-wise last dense ID $n^{dense}_{\beta^{wp}}$ is:

$$n^{dense}_{\beta^{wp}} := \sum_{\beta'=1}^{\beta^{wp}} \hat{\kappa}_{\beta'} \times \hat{\alpha}_{\beta'} \tag{2.22}$$

and the workpiece-wise last sparse ID $n^{sp}_{\beta^{wp}}$ is:

$$n^{sp}_{\beta^{wp}} := \sum_{\beta'=1}^{\beta^{wp}} \hat{\beta}^{sym\_pair}_{\beta'} \times \hat{\beta}^{cp\_pair}_{\beta'} \tag{2.23}$$

Note that dense ID is for the residue vector $\mathbf{d}$, while the sparse matrix is for the stiffness matrix $\mathbf{K}$.

B-3, for the residue vector, the size is $\hat{\alpha}' = n^{dense}_{\hat{\beta}^{wp}}$ and the global DOF mapping is explicitly:

$$\alpha'(\beta^{wp}, \kappa, \alpha) := n^{dense}_{\beta^{wp}-1} + (\kappa - 1)\hat{\alpha}_{\beta^{wp}} + \alpha \tag{2.24}$$

$$\alpha'(\beta^{wp}, \kappa, \beta^{el}, \beta^{el\_cp}) := \alpha'(\beta^{wp}, \kappa, \alpha(\beta^{el}, \beta^{el\_cp})) \tag{2.25}$$

where $n^{dense}_0 = 0$.

The global effective arrays $\{\partial^{\nu}_t \widetilde{\phi}_{\alpha'}\}$, the increment variable arrays $\{\Delta\partial^{\nu}_t \phi_{\alpha'}\}$ and the global residue array $\{d_{\overline{\alpha}'}\}$, are allocated with sizes $\hat{\nu} \times \hat{\alpha}'$, $\hat{\nu} \times \hat{\alpha}'$, and $\hat{\alpha}'$ respectively. Note that here both $\partial^{\nu}_t \phi$ and $\partial^{\nu}_t \widetilde{\phi}$ are regarded as numerical variables instead of symbolic differentiations, i.e., $\{\partial_t \phi_{\alpha'}\} = \{u_{\alpha'}\}$, so as in the Block C and D.

B-4, for the stiffness matrix, we:

(1) Define the total sparse vector size $\hat{\beta}^{sp} = n^{sp}_{\hat{\beta}^{wp}}$;

(2) Allocate the row ID, the column ID and the value arrays $\{I_{\beta^{sp}}\}$, $\{J_{\beta^{sp}}\}$, $\{K_{\beta^{sp}}\}$ with $\beta^{sp} = 1, 2, ..., \hat{\beta}^{sp}$ defining the COO-format sparse matrix;

(3) For each basic symbol pair and workpiece control point pair, i.e.,

$$\{(\kappa_0, \kappa_\lambda)_{\beta^{sym\_pair}}, \quad (\alpha_1, \alpha_2)_{\beta^{cp\_pair}}\}_{\beta^{wp}}$$

we define the sparse ID mapping

$$\beta^{sp}(\beta^{wp}, \beta^{sym\_pair}, \beta^{cp\_pair}) := n^{sp}_{(\beta^{wp}-1)} + (\beta^{sym\_pair} - 1)\hat{\beta}^{cp\_pair}_{\beta^{wp}} + \beta^{cp\_pair} \tag{2.26}$$

and fill the corresponding component of the row ID array and the column ID array with:

$$I_{\beta^{sp}} = \alpha'(\beta^{wp}, \kappa_0, \alpha_1) = n^{dense}_{(\beta^{wp}-1)} + (\kappa_0 - 1)\hat{\alpha}_{\beta^{wp}} + \alpha_1 \tag{2.27}$$

$$J_{\beta^{sp}} = \alpha'(\beta^{wp}, \kappa_\lambda, \alpha_2) = n^{dense}_{(\beta^{wp}-1)} + (\kappa_\lambda - 1)\hat{\alpha}_{\beta^{wp}} + \alpha_2 \tag{2.28}$$

Block C - *Timestep Initialization* runs at the beginning of each timestep to refresh the incremental data.

C-1, each workpiece control point variable value $\partial_t^\nu \phi_\alpha^\kappa$ in $\Omega_{\beta^{wp}}$ is updated by:

$$\partial_t^\nu \phi_\alpha^\kappa \mathrel{+}= \Delta \partial_t^\nu \phi_{\alpha'(\beta^{wp}, \kappa, \alpha)} \tag{2.29}$$

C-2, the new time t and timestep Δt are calculated. $\Delta \partial_t^{\hat{\nu}} \phi_{\alpha'}$ is cleared (to zeros).

C-3, new $\{\Delta\partial_t^{\nu}\phi_{\alpha'}\}$ is initialized by updating the sequence of $\nu = (\hat{\nu}-1), (\hat{\nu}-2), .., 0$, i.e.:

$$\Delta\partial_t^{\nu}\phi_{\alpha'} = (b_{(\nu+1)}\Delta t)\partial_t^{(\nu+1)}\phi_{\alpha}^{\kappa} + \Delta\partial_t^{(\nu+1)}\phi_{\alpha'} \tag{2.30}$$

Block D - *Sub-step Iteration* is executed at each sub-step.

D-1, the effective variable values $\{\partial_t^{\nu}\widetilde{\phi}_{\alpha'}\}$ are updated as:

$$\partial_t^{\nu}\widetilde{\phi}_{\alpha'} = c_{(\nu+1)}\Delta\partial_t^{\nu}\phi_{\alpha'} + \partial_t^{\nu}\phi_{\alpha}^{\kappa} \tag{2.31}$$

D-2, the residue array $\{d_{\overline{\alpha}'}\}$ is cleared first. Then for $\overline{\beta}^{el\_cp} = 1, 2, ..., \hat{\beta}_{\beta^{el}}^{el\_cp}$ in each element β^{el} of each bilinear form in $\{B_{\beta^b}\}_{\beta^{wp}}$ in each workpiece β^{wp}

$$(D_0\overline{\phi}^{\kappa_0}, \mathcal{L}^a(..., \partial_t^{\nu_\lambda}D_\lambda\phi^{\kappa_\lambda}, ...))_{\Omega_{\beta^{wp}}}$$

the residue vector $\{d_{\overline{\alpha}'}\}$ is updated by the (atomic) increment:

$$\begin{aligned} & d_{\alpha'(\beta^{wp},\kappa^0,\beta^{el},\overline{\beta}^{el\_cp}))} \mathrel{+}= \\ & \sum_{\gamma}\{w_{\gamma}^{\text{itg}}(D_0\overline{N}_{\alpha(\beta^{el},\overline{\beta}^{el\_cp})})|_{\vec{x}_{\gamma}^{\text{itg}}} \\ & \mathcal{L}^a(..., \sum_{\beta^{el\_cp}=1}^{\hat{\beta}_{\beta^{el}}^{el\_cp}}(D_\lambda N_{\alpha(\beta^{el},\beta^{el\_cp})})|_{\vec{x}_{\gamma}^{\text{itg}}}(\partial_t^{\nu_\lambda}\widetilde{\phi}_{\alpha'(\beta^{wp},\kappa_\lambda,\beta^{el},\beta^{el\_cp})}), ...)\} \end{aligned} \tag{2.32}$$

If the residue is small enough, break, else continue.

D-3, the sparse value array $\{K_{\beta^{sp}}\}$ is cleared first. Then for $\beta^{el\_cp}, \overline{\beta}^{el\_cp} = 1, 2, ..., \hat{\beta}_{\beta^{el}}^{el\_cp}$ in each element β^{el} of each differentiated bilinear form in $\{B_{\beta^{b'}}^{diff}\}_{\beta^{wp}}$ in each workpiece

β^{wp},

$$(D_0\overline{\phi}^{\kappa_0}, \frac{\partial\mathcal{L}^a}{\partial\lambda^{diff}}(..., \partial_t^{\nu_\lambda} D_\lambda \phi^{\kappa_\lambda}, ...))_{\Omega_{\beta^{wp}}}$$

the sparse value array component $K_{\beta^{sp}}$ is updated by the (atomic) increment:

$$K_{\beta^{sp}} \mathrel{+}= \tag{2.33}$$
$$\sum_\gamma \{ w_\gamma^{\text{itg}} (D_0 \overline{N}_{\alpha(\beta^{el}, \overline{\beta}^{el\_cp})} D_{\lambda^{diff}} N_{\alpha(\beta^{el}, \beta^{el\_cp})})|_{\vec{x}_\gamma^{\text{itg}}} \frac{c_{(\nu_{\lambda^{diff}}+1)}}{\prod_{\beta'=1}^{\nu_{\lambda^{diff}}} (b_{\beta'}\Delta t)} \times$$
$$\frac{\partial\mathcal{L}^a}{\partial\lambda^{diff}}(..., \sum_{\beta'=1}^{\hat{\beta}_{\beta^{el}}^{el\_cp}} (D_\lambda N_{\alpha(\beta^{el},\beta')})|_{\vec{x}_\gamma^{\text{itg}}} (\partial_t^{\nu_\lambda} \widetilde{\phi}_{\alpha'(\beta^{wp},\kappa_\lambda,\beta^{el},\beta')}), ...)\}$$

where the sparse matrix entry β^{sp} is computed from eq. (2.26) with the basic symbol pair index β^{sym_pair} from pair $(\kappa_0, \kappa_{\lambda^{diff}})$ and the local control point pair index β^{cp_pair} from pair $(\alpha(\beta^{el}, \overline{\beta}^{el\_cp}), \alpha(\beta^{el}, \beta^{el\_cp}))$.

D-4, the sub-step increment $\{\Delta_{sub}\phi_{\alpha'}\}$ is calculated by solving $\mathbf{K}/\mathbf{d}$ through a (direct or iterative) linear solver. The incremental variable array $\{\Delta\partial_t^\nu \phi_{\alpha'}\}$ is updated by:

$$\Delta\partial_t^\nu \phi_{\alpha'} \mathrel{+}= \Delta_{sub}\phi_{\alpha'} \prod_{\beta'=1}^{\nu} (b_{\beta'}\Delta t), \quad \nu = 0, 1, ..., \hat{\nu} \tag{2.34}$$

and the program returns to the Block start **D-1**.

With the four (4) blocks discussed above, we can have a brief review on the kernel workflow in fig. 2.1. **Block A** parses and rewrites (automatically derives) the input expressions of the physics into intermediate representations. **Block B** links the intermediate representations to the mesh and allocates all the data arrays. **Block C** updates the data arrays by one timestep by repeating **Block D** until the residue is reduced below the tolerance. Or more function-wise, firstly **Block A** processes symbols, then **Block B** assembles arrays, finally **Block C** and **Block D** actually run the FEM simulation. Now, we will discuss in detail in chapter 3 how exactly symbols are processed.

CHAPTER 3

The Rewriting System

The center of our CAS is a rewriting system for the generic automated symbolic derivation, whose main applications are simplification (in multiple ways) and symbolic differentiation. The rewriting system is designed primarily from a continuum mechanics perspective following the notations used in [**22**].

3.1. Theory

The fundamental elements of the rewriting systems are symbols with (or without) indices, denoted by words. A word can be represented in the meta form $w_{a_1 \ldots a_n, b_1 \ldots b_m}$, where w is the physical tensor symbol, $a_1 \ldots a_n$ are the component indices with n being the tensor order and $b_1 \ldots b_m$ are the derivatives. As the data structure implies, covariance and contravariance indices are not distinguished in MetaFEM for simplicity.

Next, the signature Σ is defined by the union of:

(1) The set of all words, denoted by $\mathcal{W}$ (W for word);
(2) The set of all real (floating-point) numbers, denoted by $\mathcal{N}$ (N for number); and
(3) The set of all function names, denoted by $\mathcal{F}$ (F for function), which contains $\mathcal{L}^a$ in chapter 2.

A ground term T is defined to be either an element in $\mathcal{W} \cup \mathcal{N}$, or a tree with a function name in $\mathcal{F}$ as the tree root and other ground terms being the leaves, which may result in a nested tree structure. The collection of ground terms is denoted as $\mathbb{T}(\Sigma, \emptyset)$.

An example ground term $T = a^2 + \sin(b)$ is shown in fig. 3.1. Each node of T can be referred to by a sequence of indices, denoted by a position. The set of all the possible positions is denoted by $Pos(T) = \{0, 1, 2, 10, 11, 12, 20, 21\}$, $T|_1 = a^2$, $T|_{20} = \sin$, $T|_{21} = b$, etc., where index 0 represents the function name while a positive integer index n represents the n^{th} subnode. Note, the separator in the sequence of indices is omitted when the maximum subnode number is under 10, following the classical literature, so that $10, 11, ...$ in the above $Pos(T)$ are actually "1-0, 1-1, ...".

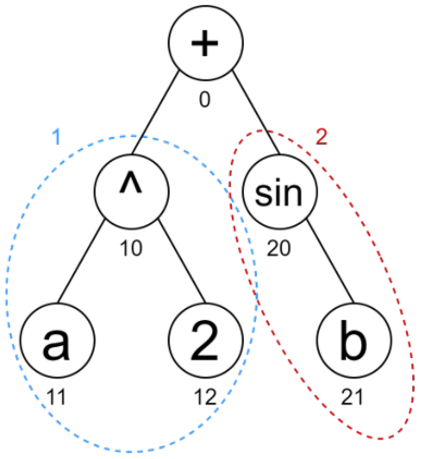


Figure 3.1. The example ground term $T = a^2 + \sin(b)$ with positions.

In practice, a ground term may represent an algebraic operation $\mathcal{L}^a(...)$, a bilinear form $(\cdot, \cdot)$, the overall weak form (but without the geometry part), or more generally, a term with some concrete physical meaning a program can evaluate. In contrast, a general term is a term optionally with some placeholders to provide structure information, but cannot be evaluated until all the placeholders are filled, in which case the general term becomes a ground term.

The placeholders are named as syntactical variables X, which can be further divided into placeholders for ground terms W and placeholders for operators F, $\mathsf{X} = \mathsf{W} \cup \mathsf{F}$ (new fonts are used for new definitions). With the syntactical variables, the general terms $\mathbb{T}(\Sigma, \mathsf{X})$ are defined to be the set where each general term $T \in \mathbb{T}(\Sigma, \mathsf{X})$ is either an element in $\mathcal{W} \cup \mathcal{N} \cup \mathsf{W}$ or a tree with an element in $\mathcal{F} \cup \mathsf{F}$ being the tree root and other general terms being the leaves.

A ground term is a general term with no variables, therefore, the general term T can also be referred to by the positions in the same way. With the positions $Pos(T)$, the set of all occurring variables of term is defined by:

$$Var(T) := \{v| \quad \exists p \in Pos(T), \quad v = T|_p, \quad v \in \mathsf{X}\} \tag{3.1}$$

On the other hand, a general term can be filled to generate a new ground term. The generalized filling process is denoted by substitution $\sigma = \sigma_{(\mathsf{X}^0, \mathcal{X}^0)}$ which is determined by a variable list $\mathsf{X}^0 = \{W_1, ..., W_m, F_1, ..., F_n\}$ and a result list $\mathcal{X}^0 = \{T_1, ..., T_m, G_1, ..., G_n\}$, where $W_1, ..., W_m \in \mathsf{W}$, $F_1, ..., F_n \in \mathsf{F}$, $T_1, ..., T_m \in \mathbb{T}(\Sigma, \emptyset)$ and $G_1, ..., G_n \in \mathcal{F}$ with m, n being non-negative integers.

The substitution can operate on a term by:

$$\begin{aligned}
\sigma(W_p) &= T_p && p = 1, ..., m \\
\sigma(F_q) &= G_q && q = 1, ..., n \\
\sigma(V) &= V && V \in (\Sigma \cup \mathsf{X}) - \mathsf{X}^0 \\
\sigma(F(V_1, V_2, ...)) &= \sigma(F)(\sigma(V_1), \sigma(V_2), ...) && (3.2)
\end{aligned}$$

In other words: (1) when σ acts on a term tree, it keeps the tree structure and acts on each node; and (2) when σ acts on a variable, if the variable is found in X^0, σ returns the element at the same index in $\mathcal{X}^0$, otherwise it does nothing.

Next, the match operator M can be defined to determine whether a general term T^M represents a ground term T, that is, $M(T^M, T) = (true, \mathcal{X}^M)$ if there exists a mapping list $\mathcal{X}^M$ so that:

$$\sigma_{(Var(T^M), \mathcal{X}^M)}(T^M) = T \tag{3.3}$$

otherwise $M(T^M, T) = (false, \emptyset)$. Note $\mathcal{X}^M$ can also be empty like in $M(T^M, T) = (true, \emptyset)$ when T^M is a specific ground term.

A classical rewrite rule R is determined by a suitable pair of general terms T^l, T^r, where the superscripts are named for the Left Hand Side (LHS) and Right Hand side (RHS) in an equation, although the considered rewrite rules are one-sided derivations (left to right only) instead of real equations, as shown in fig. 3.2,

where:

$$R \coloneqq (T^l, T^r) \in \mathbb{T}(\Sigma, \mathsf{X}) \times \mathbb{T}(\Sigma, \mathsf{X})$$

$$Var(T^r) \subseteq Var(T^l) \tag{3.4}$$

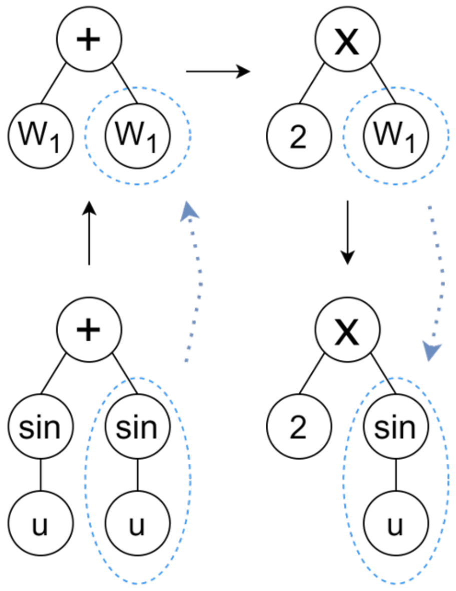


Figure 3.2. The example rule rewriting $sin(u) + sin(u) \rightarrow 2 * sin(u)$.

and R can operate on a ground term node T (but not its subnodes) by:

$$(Bool, \mathcal{X}^l) \coloneqq M(T^l, T)$$

$$R(T) = \begin{cases} \sigma_{(Var(T^l), \mathcal{X}^l)}(T^r), & Bool = true \\ T, & Bool = false \end{cases} \tag{3.5}$$

which is, if T matches T^l, R rewrites T by substitution of T^r with the mapping list from $M(T^l, T)$, otherwise it does nothing.

Further, a rewrite rule R can recursively apply to a whole ground term T, which is denoted by $\widehat{R}(T)$, e.g., a Leftmost-Innermost (LI) formulation is:

$$\widehat{R}(T) = \begin{cases} R(F(\widehat{R}(V_1), \widehat{R}(V_2), ...)), & T = F(V_1, V_2, ...) \\ R(T), & \text{else} \end{cases} \tag{3.6}$$

A practical rewriting is to repetitively apply a set of rewriting rules to the target term until no change occurs.

In practice, one may (plainly) extend the classical rewrite rule (T^l, T^r) in two ways:

(1) The match operator M is purely about structure. An operator can be defined to check the content (per rewrite rule), denoted by the checker $Ck := \{c_1, c_2, ..., c_{m+n}\}$, where each $c_\beta, \quad \beta = 1, 2, ..., m+n$ is simply a predefined set. The checker checks whether each entry in the (matched) LHS result list $\mathcal{X}^l = \{T^l_1, ..., T^l_m, G^l_1, ..., G^l_n\}$ belongs to the set at the same index of itself as follows:

$$Ck(\mathcal{X}^0) = \begin{cases} true, & T^l_1 \in c_1, ..., T^l_m \in c_m, G^l_1 \in c_{m+1}, ..., G^l_n \in c_{m+n} \\ false, & \text{else,} \end{cases} \tag{3.7}$$

(2) The restriction in eq. (3.4) keeps $R(T)$ a ground term, but is too strict. What is really needed is that $Var(T^r)$ is computable from $Var(T^l)$. Therefore, for a general $Var(T^r) = \{W^r_1, ..., W^r_{m^r}, F^r_1, ..., F^r_{n^r}\}$, eq. (3.4) can be replaced by a operator (per rewrite rule), denoted by the transcriber $Tb := \{H_1, ..., H_{m^r+n^r}\}$ with each entry a (programmable) function which takes in the LHS result list $\mathcal{X}^l$ and outputs the ground term at the same index to form the RHS result list $\mathcal{X}^r$, that is:

$$\mathcal{X}^r(\mathcal{X}^l) = Tb(\mathcal{X}^l) = \{H_1(\mathcal{X}^l), ..., H_{m^r+n^r}(\mathcal{X}^l)\} \tag{3.8}$$

Finally, the extended rewrite rule for the FEM CAS can be redefined by $R = (T^l, T^r, Ck, Tb)$ so that:

$$(Bool, \mathcal{X}^l) \coloneqq M(T^l, T)$$

$$R(T) = \begin{cases} \sigma_{(Var(T^r), Tb(\mathcal{X}^l))}(T^r), & Bool \wedge Ck(\mathcal{X}^l) = true \\ T, & \text{else} \end{cases} \tag{3.9}$$

which is ready for implementation with unchanged $\widehat{R}(T)$. A brief review of the proposed rewriting system is listed in table 3.1.

Table 3.1. Overview of the rewriting system.

Signature			Terms		Rules
Numbers	Words	Functions	Ground terms	General terms	
0.65, 1.0	$p, u_{i,t}$, $\sigma_{ij,j}$	$+, \times, (\cdot, \cdot)$, $myfunc$	$0.1 + p$, $a^2 + sin(b)$	$a^2 + sin(b)$, $W_1, W_1 + W_1$	$W_1 + 0 \to W_1$, $W_1 + W_1 \to 2 \times W_1$

3.2. Implementation

The code for a rewriting system can be very simple and straightforward, since applying a rewriting rule $R(\cdot)$ is essentially a pattern matching problem from the computing perspective. An example data structure, instead of the real code to make it simpler in few lines, is shown below.

There is no difference in data structure between placeholders for words or for functions, and only a general variable is needed. On the other hand, although the checker and the transcriber Ck, Tb are defined as rule-wise collections, each c and each H are more conveniently attached to the corresponding variables in practice. Therefore, a syntactic variable is defined by:

```
mutable struct Syntactic_Variable
    id::Symbol
    c::Function
    H::Function
    H_args::Vector{Syntactic_Variable}
end
```

where H_args contains the arguments of H.

A term is either a number, a word (definition [**4**] omitted here for simplicity), a variable or a tree, and can be formulated as:

```
struct Term_Tree
    head::Union{Function_Name, Syntactic_Variable}
    args::Vector{Union{Term_Tree, Number, Word, Syntactic_Variable}}
end
Term = Union{Term_Tree, Number, Word, Syntactic_Variable}
```

Since c and H are variable-wise, a rewrite rule is simply:

```
struct Rewrite_Rule
    LHS::Term
    RHS::Term
    LHS_vars::Vector{Syntactic_Variable}
    RHS_vars::Vector{Syntactic_Variable}
end
```

where LHS_vars, RHS_vars are explicitly collected for illustration convenience since they contain both the checkers and transcribers.

Locally applying a rewrite rule to a ground term node (but not its subnodes), i.e., $R(T)$, can be simply:

```
function apply(rule::Rewrite_Rule, src_term::Term)
    match_is_successful, LHS_result_list = match(rule.LHS, src_term)
    if match_is_successful
        checks_are_passed = check(LHS_result_list, rule.LHS_vars)
        if checks_are_passed
            RHS_result_list = transcribe(LHS_result_list, rule.RHS_vars)
            return substitute(rule.RHS, RHS_result_list)
        end
    end
    return src
end
```

which can be described in 4 steps:

(1) Matching the rewrite rule LHS to the ground term node-by-node through a Deterministic Finite Automaton (DFA);

(2) If the matching (of structures) is successful, check the LHS result list with checkers in LHS variables;

(3) If all the checks pass, transcribe the RHS result list by the transcriber in RHS variables and the LHS result list; and

(4) Generate the new ground term by substituting the rewrite rule RHS with the RHS result list.

With the local apply function, the global apply ($\widehat{R}(T)$) can be coded with multiple dispatch as:

```
apply_g(rule::Rewrite_Rule, src::Union{Word, Number}) = apply(rule, src)
apply_g(rule::Rewrite_Rule, src::Term_Tree) = apply(rule,
    Term_Tree(src.head, map(x -> apply_g(rule, x), src.args)))
```

In practice, an intuitive grammar is designed for inputting rules, and we show a real code that uses the current approach. Specifically:

(1) (Part of) the code for expression simplification is:

```
@Define_Semantic_Constraint N₁ ∈ (N₁ isa Number)
@Define_Semantic_Constraint N₂ ∈ (N₂ isa Number)

@Define_Aux_Semantics N₁₊₂(N₁,N₂) = N₁ + N₂
@Define_Aux_Semantics N₁₊₊(N₁) = N₁ + 1
@Define_Aux_Semantics N₁₂(N₁,N₂) = N₁ * N₂
@Define_Aux_Semantics N₁ₚ₂(N₁,N₂) = N₁ ^ N₂

@Define_Rewrite_Rule Unary_Sub ≔ -(a) => (-1) * a
@Define_Rewrite_Rule Binary_Sub ≔ (b - a => b + (-1) * a)
@Define_Rewrite_Rule Binary_Div ≔ (a / b => a * b ^ (-1))

@Define_Rewrite_Rule Unary_Add ≔ (+(a) => a)
```

```
@Define_Rewrite_Rule Add_Splat ≔ ((a...) + (+(b...)) + (c...) => a + b + c)
@Define_Rewrite_Rule Add_Numbers ≔ ((a...) + N₁ + (b...) + N₂ + (c...) =>
                                    N₁₊₂ + a + b + c)
@Define_Rewrite_Rule Add_Sort ≔ (a + (b...) + N₁ + (c...) => N₁ + a + b + c)
@Define_Rewrite_Rule Add_0 ≔ (0 + (a...) => +(a))

@Define_Rewrite_Rule Unary_Mul ≔ (*(a) => a)
@Define_Rewrite_Rule Mul_Splat ≔ ((a...) * (*(b...)) * (c...) => a * b * c)
@Define_Rewrite_Rule Mul_Numbers ≔ ((a...) * N₁ * (b...) * N₂ * (c...) =>
                                    N₁₂ * a * b * c)
@Define_Rewrite_Rule Mul_Sort ≔ (a * (b...) * N₁ * (c...) => N₁ * a * b * c)
@Define_Rewrite_Rule Mul_0 ≔ (0 * (a...) => 0)
@Define_Rewrite_Rule Mul_1 ≔ (1 * (a...) => *(a))

@Define_Rewrite_Rule Pow_a0 ≔ (a ^ 0 => 1)
@Define_Rewrite_Rule Pow_a1 ≔ (a ^ 1 => a)
@Define_Rewrite_Rule Pow_0a ≔ (0 ^ a => 0)
@Define_Rewrite_Rule Pow_1a ≔ (1 ^ a => 1)
@Define_Rewrite_Rule Pow_Numbers ≔ (N₁ ^ N₂ => N₁ₚ₂)
@Define_Rewrite_Rule Pow_Splat ≔ ((a ^ b) ^ c => a ^ (b * c))
```

(2) The code for symbolic differentiation is:

```
@Define_Semantic_Constraint S₁ ∈ (S₁ isa SymbolicWord)
@Define_Semantic_Constraint S₂ ∈ (S₂ isa SymbolicWord)

@Define_Semantic_Constraint f_undef ∈ (~(f_undef in [:+; :-; :*;
                                         :/; :^; :log; :inv]))

@Define_Rewrite_Rule Basic_Diff1 ≔ ∂(S₁, S₁) => 1
```

```
@Define_Rewrite_Rule Basic_Diff2 ≔ ∂(S₂, S₁) => 0
@Define_Rewrite_Rule Basic_Diff3 ≔ ∂(N₁, S₁) => 0


@Define_Rewrite_Rule Basic_Diff4 ≔ ∂({f_undef}(a...), S₁) => 0


@Define_Rewrite_Rule Add_Diff ≔ ∂(a + (b...), S₁) =>
    ∂(a, S₁) + ∂(+(b...), S₁)
@Define_Rewrite_Rule Mul_Diff ≔ ∂(a * (b...), S₁) =>
    ∂(a, S₁) * (b...) + a * ∂(*(b...), S₁)
@Define_Rewrite_Rule Pow_Diff ≔ ∂(a ^ b, S₁) =>
    ∂(a, S₁) * a ^ (b - 1) * b  + ∂(b, S₁) * log(a) * a ^ b
```

where:

(1) Each line with *Define Semantic Constraint* attaches the check function c to the variable;

(2) Each line with *Define Aux Semantics* attaches the transcribe function H and the function argument variable H_args to the variable; and

(3) Each line with *Define Rewrite Rule* defines a rule with the rule name before $:=$ while the LHS and the RHS are connected with $\Rightarrow$ after $:=$. The splatting operator (...) denotes a variable quantity (0 or arbitrarily more) of function inputs, similar to the Kleene star in a regular expression.

Finally, we provide a discussion about the practical utility in closing this chapter.

3.3. Discussion

Although the rewriting system plays a central role in the CAS, one should note that the rewriting process only makes up a minor portion in the whole translation process. On the one hand, the rewriting process is flexible, intuitive but often not very performant, e.g., merging common subterms is much more efficient with hashing, so in actual MetaFEM a large portion of common symbolic operations are hard-coded manually for speed. On the other hand, a CAS is essentially a static analysis from a compiler perspective, which faces a relative difficulty in the control flows. For example, in plasticity, the variable evolves according to the yielding criteria, which is an implicit, nonlinear constraint and usually solved by another manually designed Newton iteration with selective differentiation, such as the radial return algorithm. Such empirical process is obviously more easily and conveniently coded as an escaped external function instead of being derived. Despite of the limitations, the rewriting system is still a powerful and practical abstraction for symbolic derivation, especially for a quick implementation of some ad-hoc procedures.

CHAPTER 4

Numerical Examples

The code is open-sourced on GitHub [4] with the modules discussed above and an original generic mesh system implemented with 2D/3D Lagrange cube/simplex elements of arbitrary order and serendipity cube elements of order 2 and 3 (without mesh generation). All the example cases shown later are also provided with most relevant files, including:

(1) Mesh and script for MetaFEM (this work) simulations;
(2) Commercial software setup and result files for comparison cases; and
(3) VTK files and Paraview states for visualization

to be repeated in exact details. To be concise, we will only focus on the input code about physics in the ensuing discussions.

4.1. Thermal conduction in a solid

We begin with thermal conduction in a solid since it is among the simplest and most stable types of practical physics for FEM. The mathematical formulation is:

$$\text{variable} \quad T, \qquad \text{parameters} \quad C, k, h, h_{penalty}, s, e_m, T_{env}, T_{fix}$$

$$
\begin{aligned}
-C(T, T_{,t}) - k(T_{,i}, T_{,i}) + (T, s) &= 0, && in \quad \Omega \\
h(T, T_{env} - T) + e_m \sigma^b (T, T_{env}^4 - T^4) &= 0, && on \quad \partial(\Omega)_{convection\_radiation} \\
h_{penalty}(T, T_{fix} - T) + k(T, n_i T_{,i}) &= 0, && on \quad \partial(\Omega)_{fix}
\end{aligned}
\tag{4.1}
$$

where T is the temperature, C is the volumetric heat capacity, k is the thermal conductivity, h is the convective coefficient, $h_{penalty}$ is a large number to enforce the fixed temperature (Dirichlet) boundary condition, s is the heat source, e_m is the emissivity and $\sigma^b = 5.670 \times 10^{-8} W/m^2K^4$ is the Stefan–Boltzmann constant.

The code for the physics is:

```
@Def begin
    heat_dissipation = - C * Bilinear(T, T{;t}) - k * Bilinear(T{;i}, T{;i}) +
                         Bilinear(T, s)
    conv_rad_boundary = h * Bilinear(T, T_env - T) + em * σ^b * Bilinear(T, T_env^4 - T^4)
    fix_boundary = h_penalty * Bilinear(T, Tw - T) + k * Bilinear(T, n{i} * T{;i})
end
```

Apparently, the grammar is almost exactly the same with the corresponding math, where the function Bilinear denotes a bilinear form $(\cdot, \cdot)$, {} in the code denotes subscripts in the math with the comma "," as a separator of words and the semicolon ";" marks the start of derivative indices.

4.1.1. Various boundary conditions on a 2D stripe

A $2\,cm \times 1\,cm$ rectangular strip is simulated for equilibrium with the bottom side insulated, the left, right sides fixed at $(900 + 273.15)\,K$ and the top side under convection and radiation with $T_{env} = (50 + 273.15)\,K, \quad h = 50\,W/m^2K$ and emissivity $e_m = 0.7$.

The case is a tutorial example in the matlab package FEATool [**37**]. Solved in MetaFEM by 40×20 quadratic serendipity rectangular elements (dx $= 0.5\,mm$), the temperature contour is plotted in fig. 4.1 and the temperature distribution along the vertical mid-line (x $= 1\,cm$) is compared to the FEATool result in fig. 4.2. The good match in temperature comparison indicates that the domain physics and boundary conditions are correctly derived and assembled, including the nonlinear radiation term.

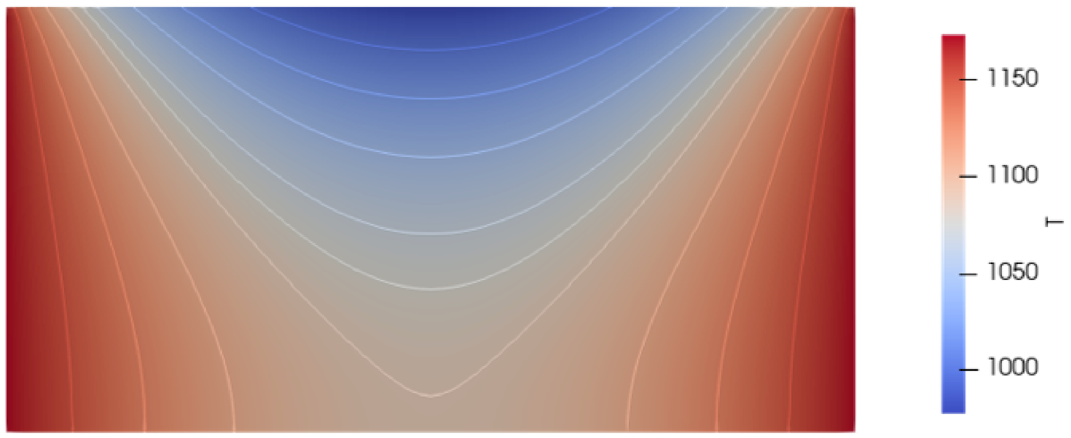


Figure 4.1. T contour of the stripe.

4.1.2. Irregular 3D geometry with heat source

To test the unstructured 3D mesh system, a pikachu is simulated where the body is considered as a uniform heat source $s = 1.6 \times 10^3\,W/m^3$, $k = 0.6\,W/(m \cdot K)$ from his

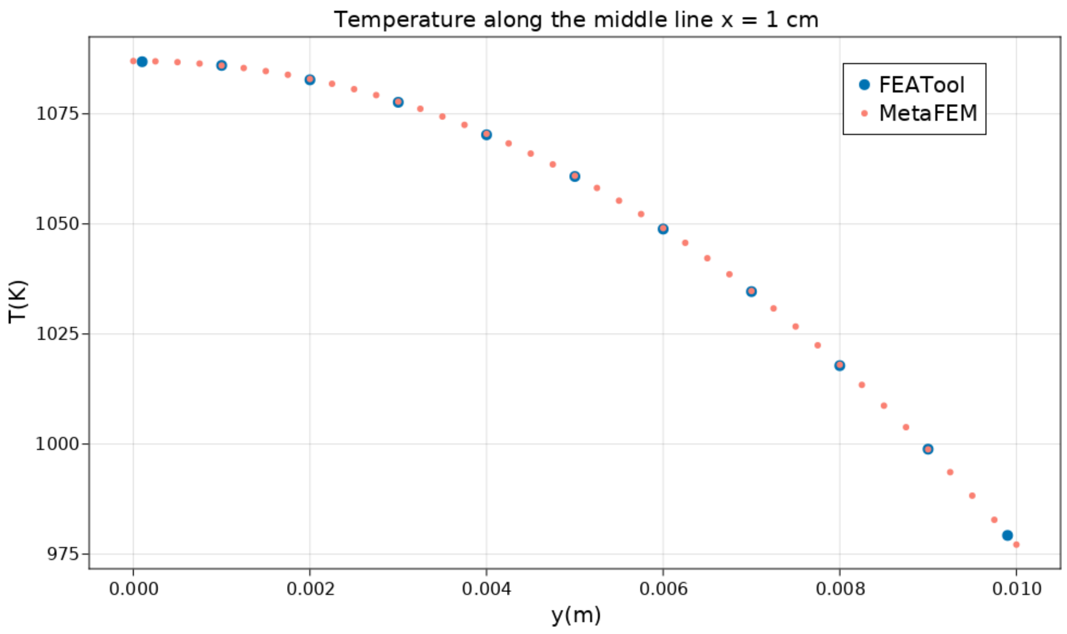


Figure 4.2. T distribution along the vertical mid-line of the stripe. FEATool data are at sample points while MetaFEM (this work) data are at mesh nodes.

own metabolism while the whole surface is under convection with $T_{env} = (20 + 273.15)\,K$, $h = 25\,W/m^2K$. The numbers of property parameters are chosen by reference to the property parameters of a human in air.

The original source is a low-poly pikachu CAD file [**27**] which is imported in COMSOL [**31**] for both mesh generation and simulation. Then, MetaFEM reads the same mesh and simulates with the same parameters. The mesh has 15,334 quadratic simplex elements and 23,703 nodes, which is shown in fig. 4.3 while the temperature distribution along sample lines a and b are compared across the two results in fig. 4.4. The good match indicates that the mesh system is able to handle complex, practical geometries equally well as common commercial solvers.

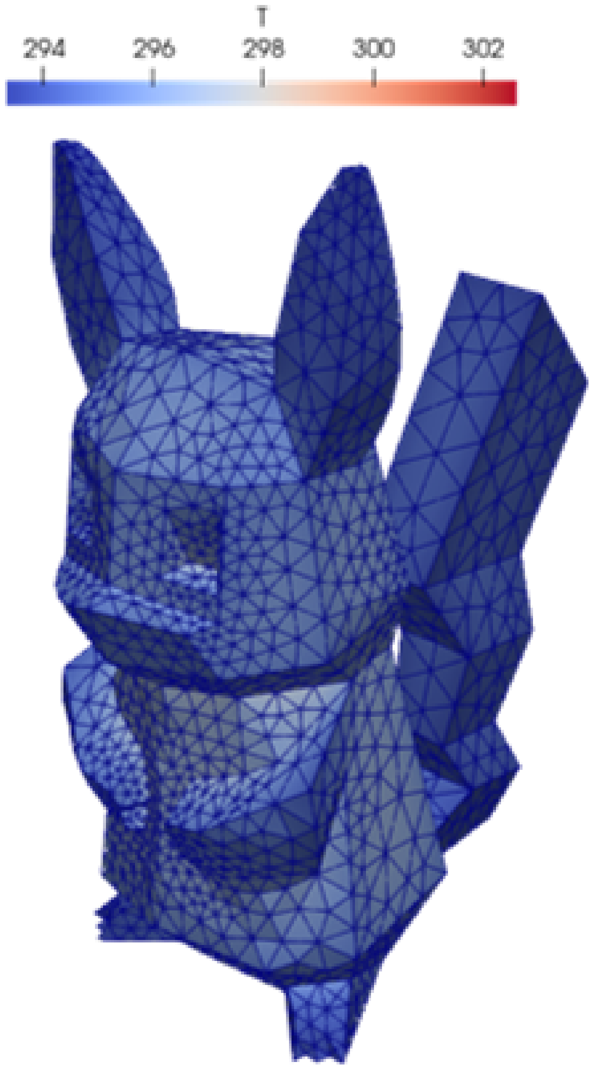


Figure 4.3. The mesh of pikachu, colored with temperature.

4.2. Linear elasticity

The mathematical formulation for the linear elastostatics in FEM is as follows:

$$\text{variables} \quad d_i, \qquad \text{parameters} \quad E, \nu, d_i^w, \sigma_{ij}^l, \tau^b$$

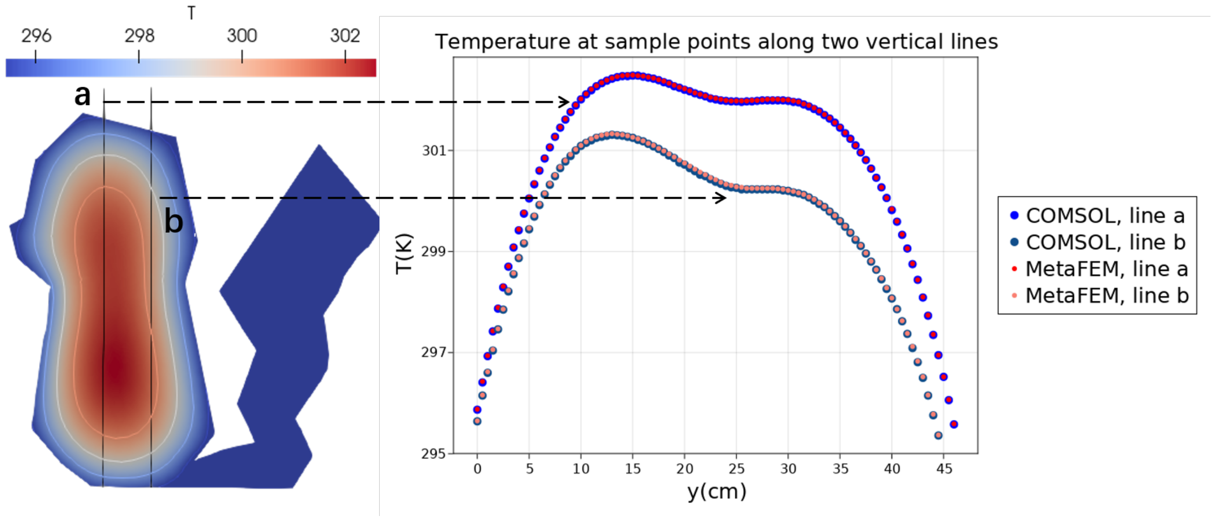


Figure 4.4. Pikachu temperature distribution. Left: Temperature contour on a slice; Right: T comparison at sample points along two vertical lines a and b in the pikachu.

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \qquad \mu = \frac{E}{2(1+\nu)}$$

$$\varepsilon_{ij} = \frac{d_{i,j}+d_{j,i}}{2}, \qquad \sigma_{ij} = \lambda\delta_{ij}\varepsilon_{mm} + 2\mu\varepsilon_{ij}$$

$$\begin{aligned} -(d_{i,j}, \sigma_{ij}) &= 0, \qquad in \quad \Omega \\ (d_i, \sigma^l_{ij} n_j) &= 0, \qquad on \quad (\partial\Omega)_{load} \\ (d_i, \tau^b(d^w_i - d_i)) &= 0, \qquad on \quad (\partial\Omega)_{fix} \end{aligned} \tag{4.2}$$

where $d_i, E, \nu, \lambda, \mu, \varepsilon_{ij}, \sigma_{ij}, \tau^b$ are the displacement, Young's modulus, Poisson ratio, Lame's first parameter, shear modulus, strain, stress and a big number to enforce the fixed displacement boundary, respectively.

The code for the physics is:

```
λ = E * ν / ((1 + ν) * (1 - 2 * ν))
μ = E / (2 * (1 + ν))

@Def ε{i,j} = (d{i;j} + d{j;i}) / 2.
@Def σ{i,j} = λ * δ{i,j} * ε{m,m} + 2. * μ * ε{i,j}

@Def begin
    ES_domain = - Bilinear(ε{i,j}, σ{i,j})
    ES_total_d_fixed_bdy = τᵇ * Bilinear(d{i}, (dʷ{i} - d{i}))
    ES_d1_fixed_bdy = τᵇ * Bilinear(d{1}, (dʷ{1} - d{1}))
    ES_loaded_bdy = Bilinear(d{i}, σˡ{i,j} * n{j})
end
```

4.2.1. 3D cantilever beam bending

A beam of normalized/dimensionless length $L = 10$, width and depth $h = 1$ with its left side (displacement) fixed is simulated for its deflection distribution along the mid-plane $y = z = 0.5$ with $10 \times 4 \times 4$ quadratic serendipity brick elements under 3 different loading conditions:

(1) Concentrated load P on the right side, where the analytical deflection distribution from beam theory [**38**] is:

$$d_2^{a1}(x) = -\frac{Px^2}{6EI}(3L - x), \qquad I := \frac{1}{12h^3} \tag{4.3}$$

(2) Uniform pressure p on the top, where the analytical deflection distribution is:

$$d_2^{a2}(x) = -\frac{px^2}{24EI}(6L^2 - 4Lx + x^2) \tag{4.4}$$

(3) Linear pressure $p'(x) = p_0(1 - \frac{x}{L})$ on the top, where the analytical deflection distribution is:

$$d_2^{a3}(x) = -\frac{p_0x^2}{120LEI}(10L^3 - 10L^2x + 5Lx^2 - x^3) \tag{4.5}$$

In all cases the Young's modulus is $E = 1$ and the Poisson ratio is $\nu = 0$, and the result is shown in fig. 4.5. A good match can be observed.

4.2.2. 2D/3D stress concentration

A square/cube of side length $L = 10\,m$ with a circular/spherical hole of radius $r = 1\,m$ under uniaxial tension σ_0 along the y axis is simulated by both MetaFEM and Abaqus with 440/3,375 quadratic serendipity quadrilateral/brick elements and 1,399/15,645 mesh nodes, in total $2 \times 2 = 4$ simulations. The material is chosen as the general steel, $E = 210 \times 10^9\,Pa, \nu = 0.3$. Only $\frac{1}{4}/\frac{1}{8}$ of the sample is simulated to take advantage of symmetry. σ_{22} distribution along the x and y axes are compared in fig. 4.6, where we recover the well-known 2D/3D stress concentration factors 3 and 2 on the intersection of the circular/spherical surface and the x axis, while the distributions calculated by MetaFEM match well with the Abaqus results.

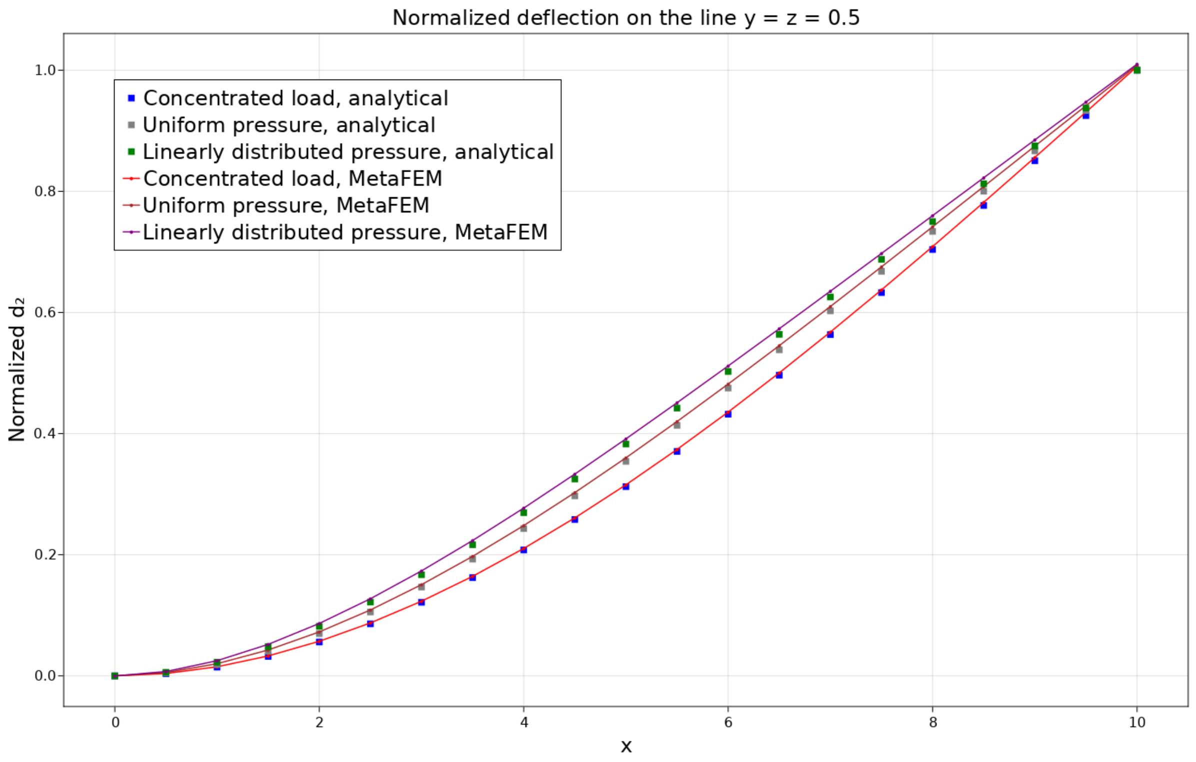


Figure 4.5. Cantilever bending. The analytical and numerical deflection distributions along the center line on the beam mid-plane after normalizing by the corresponding maximum analytical deflection (to fit different deflection scales in one graph).

4.3. Thermal elasticity

Combining heat conduction and linear elasticity, we obtain the coupling thermal elasticity problem to show a dynamic multi-physics example, where the mathematical formulation is:

$$\text{variables} \quad d_i, T, \qquad \text{parameters} \quad E, \nu, \rho, c, \alpha, C, k, \tau^b$$

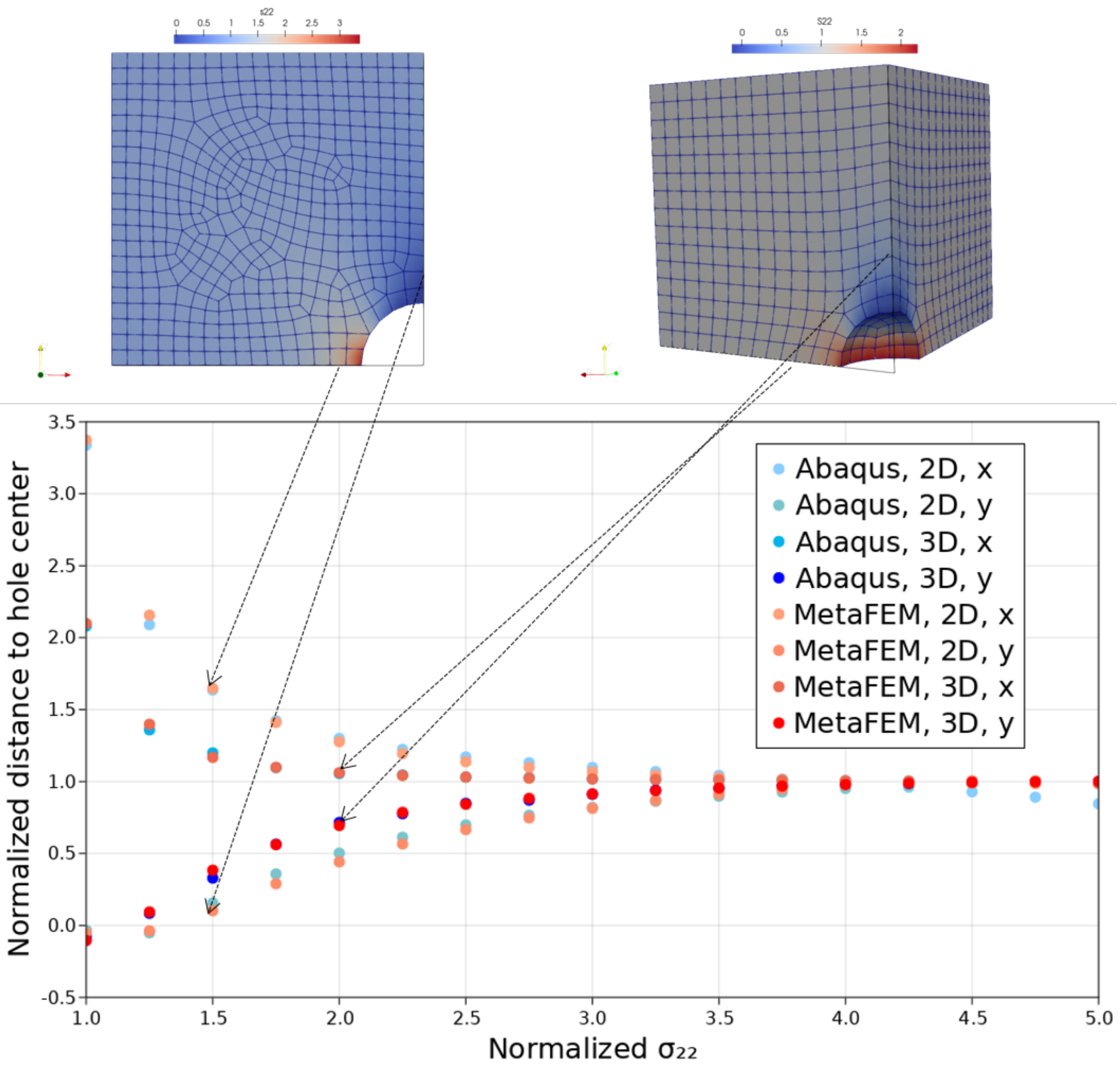


Figure 4.6. 2D/3D stress concentration. Top left: Mesh and σ_{22} distribution for 2D; Top right: Mesh and σ_{22} distribution for 3D; Bottom: Comparison of σ_{22} along two axes.

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \qquad \mu = \frac{E}{2(1+\nu)}$$

$$\varepsilon_{ij} = \frac{d_{i,j} + d_{j,i}}{2} - \alpha T \delta_{ij}, \qquad \sigma_{ij} = \lambda \delta_{ij} \varepsilon_{mm} + 2\mu\varepsilon_{ij}$$

$$(\varepsilon_{ij}, \sigma_{ij}) + \rho(d_i, cd_{i,t}) + C(T, T_{,t}) + k(T_{,i}, T_{,i}) = 0, \qquad in \quad \Omega$$

$$h(T, T - T_e) = 0, \qquad on \quad (\partial\Omega)_{convection}$$

$$\tau^b(d_i, d_i) = 0, \qquad on \quad (\partial\Omega)_{fix} \tag{4.6}$$

where we assumed the mechanical motion is viscosity-driven and α, ρ, c are the coefficient of thermal expansion, the density and the viscosity factor, respectively.

The code for the physics is:

```
λ = E * ν / ((1 + ν) * (1 - 2 * ν))
μ = E / (2 * (1 + ν))

@Def begin
    ε{i,j} = (d{i;j} + d{j;i}) / 2. - α * T * δ{i,j}
    σ{i,j} = λ * δ{i,j} * ε{m,m} + 2. * μ * ε{i,j}

    heat_dissipation = C * Bilinear(T, T{;t}) + k * Bilinear(T{;i}, T{;i})
    elasticity = Bilinear(ε{i,j}, σ{i,j}) + Bilinear(d{i}, ρ * (c * d{i;t}))

    domain = heat_dissipation + elasticity
    conv_bdy = h * Bilinear(T, T - Te)
    fixed_bdy = τᵇ * Bilinear(d{i}, d{i})
end
```

We consider a thermal bending problem, where the geometry is still the $10m \times 1m \times 1m$ block used in the cantilever case in section 4.2.1. Mechanically, the beam has its left face fixed and the other faces free. Thermally, the beam has its bottom face linked to a 300°C source and its top face linked to a 0°C source. The other faces remain adiabatic. Having the facial convection coefficient h numerically equal to the dimensioned conductivity $k \cdot 1m^{-1}$, the thermal distribution can be solved analytically to obtain an equilibrium linear temperature distribution along the beam height from 200°C(bot) to 100°C(top), as shown in fig. 4.7. Assuming a linear expansion factor $\alpha = 50 \times 10^{-6}/K$, the thermal strain is 0.05 and causes the beam bending as:

$$\frac{d\theta}{dx} = \frac{\alpha \Delta T}{L} = 0.005 \rightarrow y = 0.0025x^2$$

Resulting in a maximum deflection of $0.25m$ and an expansion of $0.01m$. The corresponding numerical solutions converged at equilibrium are $0.253m$ and $0.099m$, respectively.

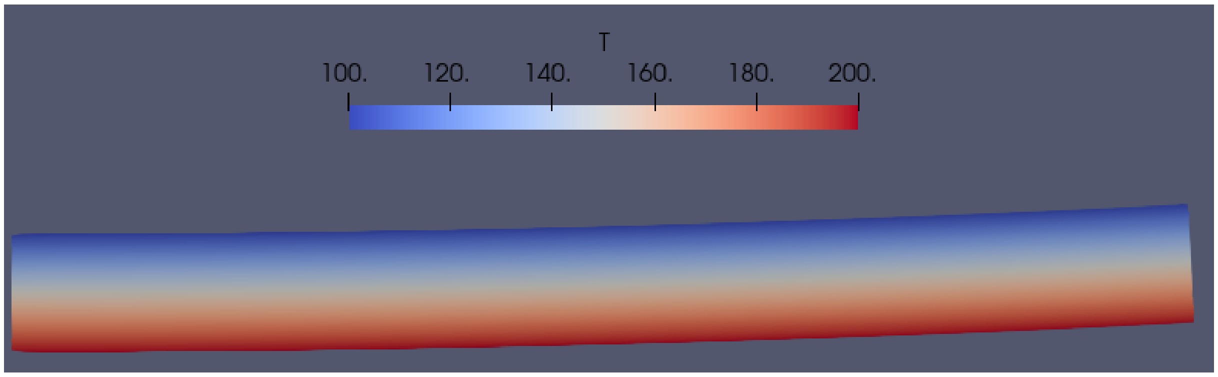


Figure 4.7. The temperature distribution of the thermal elastic bending beam at equilibrium.

Note, thermal-elasticity has at least two different characteristic time scales, i.e., in heat conduction and in mechanical motion, respectively. In the above code, we simply use the built-in time scheme by the subnote "{;t}", which is generalized-alpha [**25**] with implicitly the same time step length for the both time scales. Suppose we want to change the time scheme for one physic process, e.g., using fully implicit stepping on the heat conduction so that the temperature evolves by:

$$\frac{\partial T}{\partial t} = f(T) \rightarrow \frac{T^{n+1} - T^n}{\Delta t} = f(T^{n+1})$$

We can rewrite the definition code to be:

```
@Def begin
    T_eval = T + Tt * dt
    ε{i,j} = (d{i;j} + d{j;i}) / 2. - α * T_eval * δ{i,j}
    σ{i,j} = λ * δ{i,j} * ε{m,m} + 2. * μ * ε{i,j}

    heat_dissipation = C * Bilinear(Tt, Tt) + k * Bilinear(Tt{;i}, T_eval{;i})
    elasticity = Bilinear(ε{i,j}, σ{i,j}) + Bilinear(d{i}, ρ * (c * d{i;t}))

    domain = heat_dissipation + elasticity
    conv_bdy = h * Bilinear(Tt, T_eval - Te)
    fixed_bdy = τᵇ * Bilinear(d{i}, d{i})
end
```

also we need to update the states (denoted by the external variable) manually at every update step in the later iteration:

```
cpts = fem_domain.workpieces[1].mesh.controlpoints
cpts.T += cpts.Tt .* dt
```

Then the code will run as it should and converges to the same result.

Here the key point is that we can always program/interact with MetaFEM in the traditional way (FEniCS) where the package assembles the weak form spatially and the user scripts for the time scheme, while the reserved symbol t/dt provides the reference to the default generalized alpha t/dt.

4.4. Hyper-elasticity

Next, we consider a uniaxial tensile test for the Mooney-Rivlin materials, which is a finite deformation hyper-elastic problem, to demonstrate that MetaFEM is also practically capable of more sophisticated tensor symbolic computation, where the mathematical formulation in the total Lagrangian formulation is:

$$\text{variables} \quad d_i, \qquad \text{parameters} \quad C_{10}, C_{01}, d_i^w, P_{ij}^l, \tau^b$$

$$F_{ij} := \frac{x_i}{X_j} = \delta_{ij} + d_{i,j}, \quad B_{ij} = F_{ik}F_{jk}$$

$$I_1 = B_{ii}, \quad I_2 = 0.5(B_{ii}^2 - B_{ij}B_{ij}), \quad J = det(F) = F_{1i}F_{2j}F_{3k}\epsilon_{ijk}$$

$$W = C_{10}(I_1 - 3 - 2log(J)) + C_{01}(I_2 - 3 - 4log(J)) + 0.5\lambda(J-1)^2, \quad P_{ij} = \frac{dW}{dF_{ij}}$$

$$\begin{aligned} -(F_{ij}, P_{ij}) &= 0, && in \quad \Omega \\ \tau^b(d_i, d_i^w - d_i) &= 0, && on \quad (\partial\Omega)_{fix} \\ (d_i, P_{ij}^l n_j) &= 0, && on \quad (\partial\Omega)_{load} \end{aligned} \tag{4.7}$$

where $F_{ij}, B_{ij}, I_1, I_2, J, C_{10}, C_{01}, W, P_{ij}$ are the deformation gradient, the left Cauchy-Green tensor, the first and the second invariant of B, the determinant of F, the 10 and 01 terms of the Mooney-Rivlin parameters, the energy density and the first Piola-Kirchhoff stress tensor, respectively.

The code for the physics is:

```
@Def begin
    F{i,j} = δ{i,j} + d{i;j}
    B{i,j} = F{i,k} * F{j,k}
    I1 = B{i,i}
    I2 = 0.5 * (B{i,i} ^ 2 - B{i,j} * B{j,i})
    J = F{1,i} * F{2,j} * F{3,k} * ϵ{i,j,k}
    W = C10 * (I1 - 3 - 2 * log(J)) + C01 * (I2 - 3 - 4 * log(J)) + 0.5 * λ * (J - 1) ^ 2
    P{i,j} = d(W, F{i,j})
end

@Def begin
    WF_domain = - Bilinear(F{i,j}, P{i,j})
    WF_fixed_bdy = τ^b * Bilinear(d{i}, (d^w{i} - d{i}))
    WF_right_bdy = Bilinear(d{i}, P^l{i,j} * n{j})
end
```

where the built-in operator d differentiates the energy density W with respect to the intermediate tensor F.

For simplicity we still use the $10m \times 1m \times 1m$ block in the cantilever case in section 4.2.1, with its left face fixed, right face loaded and the other faces free. The deformation gradient F can be explicitly approximated as:

$$F_{ij} = l_1 \delta_{i1} \delta_{j1} + l_2(\delta_{i2}\delta_{j2} + \delta_{i3}\delta_{j3})$$

Then we plug F into P_{22}, which has to be 0 in a uniaxial tension, thus obtain the relation as follows:

$$P_{22} = ... = 2C_{10}l_2 + 2C_{01}(l_2(l_1^2 + l_2^2)) + (\lambda(J-1) - 2C_{10} - 4C_{01})l_2^{-1} = 0$$

which is just a second-order algebraic equation of l_2^2, thus we define $L \coloneqq l_2^2$ and represent it by l_1:

$$L(l_1) = \frac{-(2C_{10} + 2C_{01}l_1^2 + \lambda l_1) + \sqrt{(2C_{10} + 2C_{01}l_1^2 + \lambda l_1)^2 + 8C_{01}(\lambda + 2C_{10} + 4C_{01})}}{4C_{01}}$$

then we obtain P_{11} explicitly:

$$P_{11} = ... = 2C_{10}l_1 + 2C_{01}(2l_1L) + (\lambda(l_1L - 1) - 2C_{10} - 4C_{01})l_1^{-1}$$

finally we compare the above analytical formula to the numerical result in fig. 4.8. Neo-Hookean materials are also checked with the same procedure in fig. 4.9. For the both cases, good matches are observed.

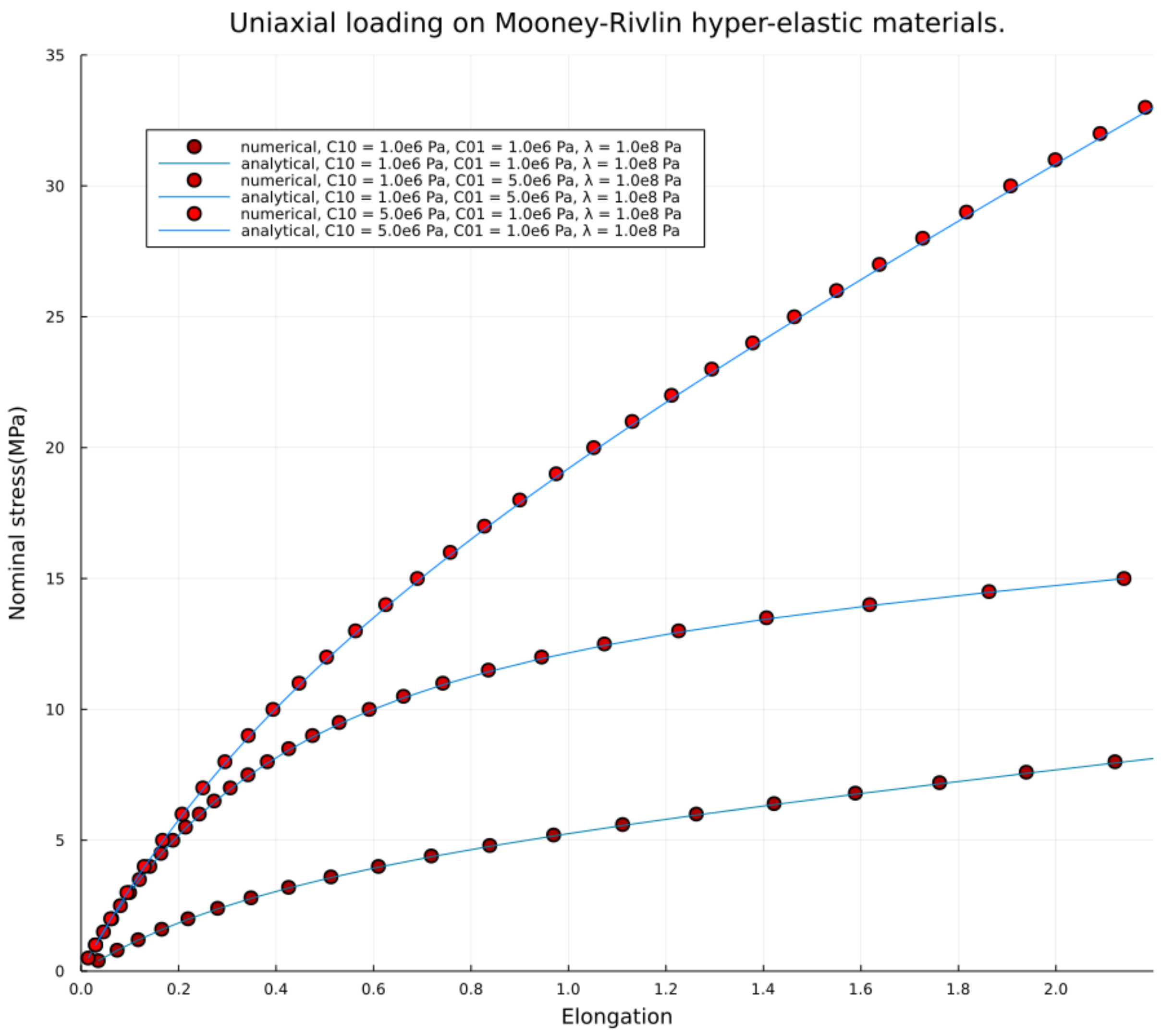


Figure 4.8. The uniaxial tensile test of the Mooney-Rivlin materials.

4.5. Hypo-elastic-plasticity

Then, we consider the same uniaxial loading for hypo-elastic-plastic materials with the J2 flow law and the mixed hardening law, which is inherently a dynamic, stateful

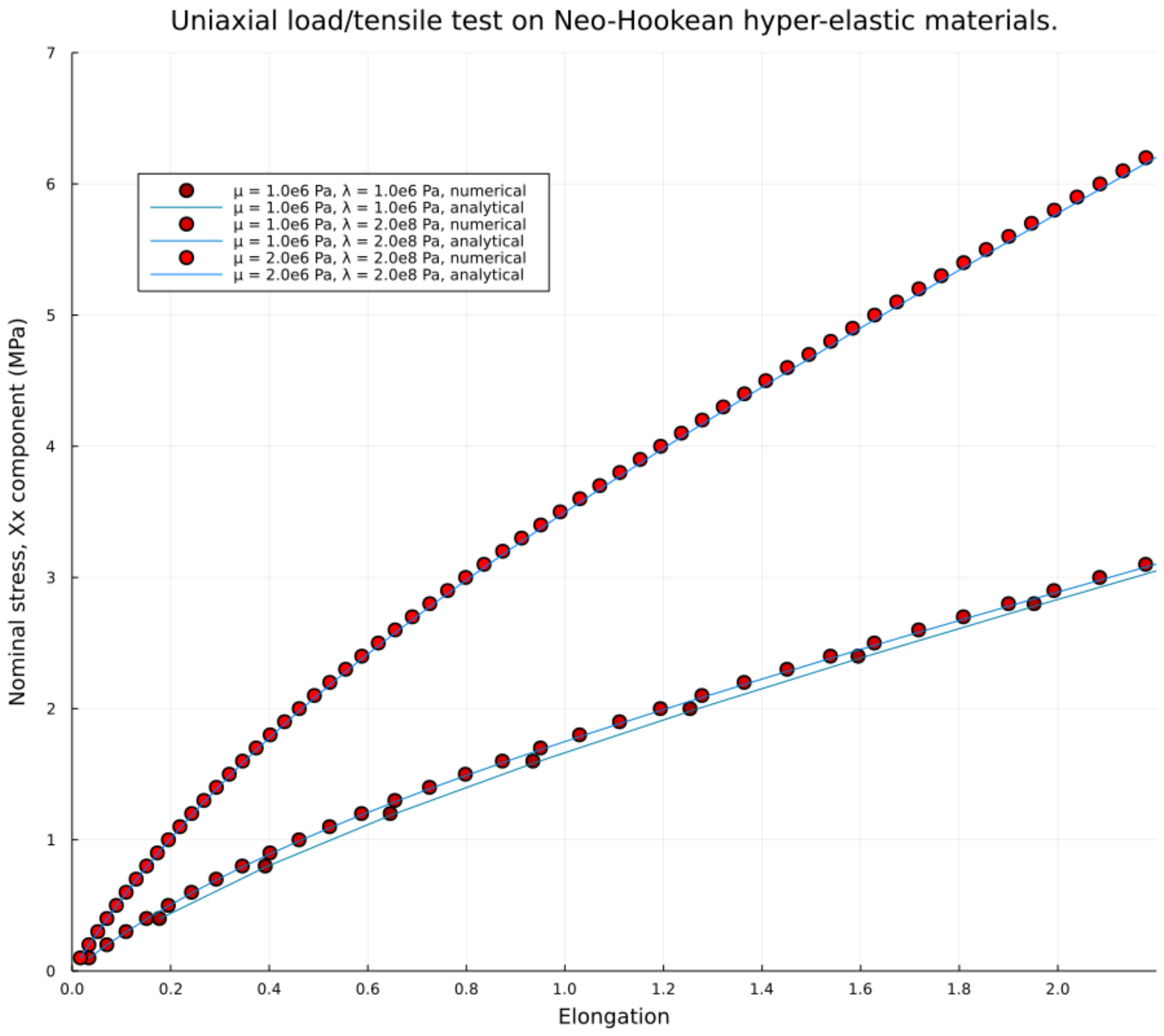


Figure 4.9. The uniaxial tensile test of the Neo-Hookean materials.

process and can be mathematically formulated by:

$$\text{variables} \quad d_i, \qquad \text{states} \quad \varepsilon^p_{ij}, \qquad \text{parameters} \quad E, \nu, E_p, E_b, Y, d^w_i, \sigma^l_{ij}, \tau^b$$

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \qquad \mu = \frac{E}{2(1+\nu)}$$
$$\varepsilon_{ij} = \frac{d_{i,j} + d_{j,i}}{2}, \qquad \sigma_{ij} = \lambda\delta_{ij}(\varepsilon_{mm} - \varepsilon^p_{mm}) + 2\mu(\varepsilon_{ij} - \varepsilon^p_{ij})$$

(4.8)
$$s_{ij} = \sigma_{ij} - b_{ij}, \quad s'_{ij} = s_{ij} - \frac{1}{3}s_{kk}\delta_{ij}, \quad J_2(s) = |s'|^2 = s'_{ij}s'_{ij}, \quad f(\sigma) = \frac{3}{2}J_2(s) - \sigma_y^2 \le 0$$

$$\varepsilon^p_{ij} = 0, \quad \sigma_y = Y, \quad b_{ij} = 0, \qquad when \quad t = 0$$

$$\frac{d\varepsilon^p_{ij}}{dt} = \lambda\frac{s'_{ij}}{|s'|}, \quad \frac{d\sigma_y}{dt} = \sqrt{\frac{2}{3}}E_p\lambda, \quad \frac{db_{ij}}{dt} = \frac{2}{3}E_b\lambda\frac{s'_{ij}}{|s'|} \tag{4.9}$$

$$\begin{aligned}(d_{i,j}, \sigma_{ij}) + \rho(d_i, cd_{i,t} + d_{i,tt}) &= 0, && in \quad \Omega\\ -(d_i, \sigma^l_{ij}n_j) &= 0, && on \quad (\partial\Omega)_{load}\\ (d_i, \tau^b(d_i - d^w_i)) &= 0, && on \quad (\partial\Omega)_{fix}\end{aligned} \tag{4.10}$$

where E_p, E_b, Y are the isotropic hardening modulus, the kinematic hardening modulus and the initial yield stress, respectively. In words, the system evolves initially by the linear elasticity and the yield criteria f in eq. (4.8) is checked every step. If $f > 0$ in a test step, the material yields with the plastic flow rate λ as the only independent internal variable. The plastic strain ε^p, the current yield stress σ_y and the back stress b evolve according to the flow rule in eq. (4.9) with the solved λ to let $f = 0$ exactly holds, representing that physically the material cannot hold shear so it deforms permanently.

The code for the physics is:

```
@Def begin
    et{i,j} = (d{i;j} + d{j;i}) / 2
    ep{i,j} = strain_updater(et{1,1}, et{1,2}, et{1,3}, et{2,2}, et{2,3}, et{3,3})
    e_eval{i,j} = et{i,j} - ep{i,j}

    σ{i,j} = 2 * μ * e_eval{i,j} + λ * e_eval{m,m} * δ{i,j}
    WF_domain = Bilinear(d{i;j}, σ{i,j}) + Bilinear(d{i}, ρ * (c * d{i;t} + d{i;t,t}))

    WF_fixed_bdy = τᵇ * Bilinear(d{i}, (d{i} - dʷ{i}))
    WF_right_bdy = Bilinear(d{i}, - σˡ{i,j} * n{j})
end
```

where the user-defined external function "strain_updater" records the states. We compare the numerical and analytical results in fig. 4.10 and find good match.

4.6. Incompressible flow

Finally, we provide fluid mechanics examples to demonstrate that MetaFEM is capable of the flow-dominated problems with stabilization. The mathematical formulation of the steady incompressible Navier-Stokes (NS) equation in FEM with the Streamline Upwind/Petrov-Galerkin (SUPG) stabilization is given by:

$$\text{variables} \quad u_i, p, \qquad \text{parameters} \quad \rho, \mu, u_i^W, \tau^b, \tau^c, \tau^m$$

$$Rc := u_{k,k}, \qquad Rm_i := \rho u_k u_{i,k} + p_{,i} - \mu u_{i,kk}$$

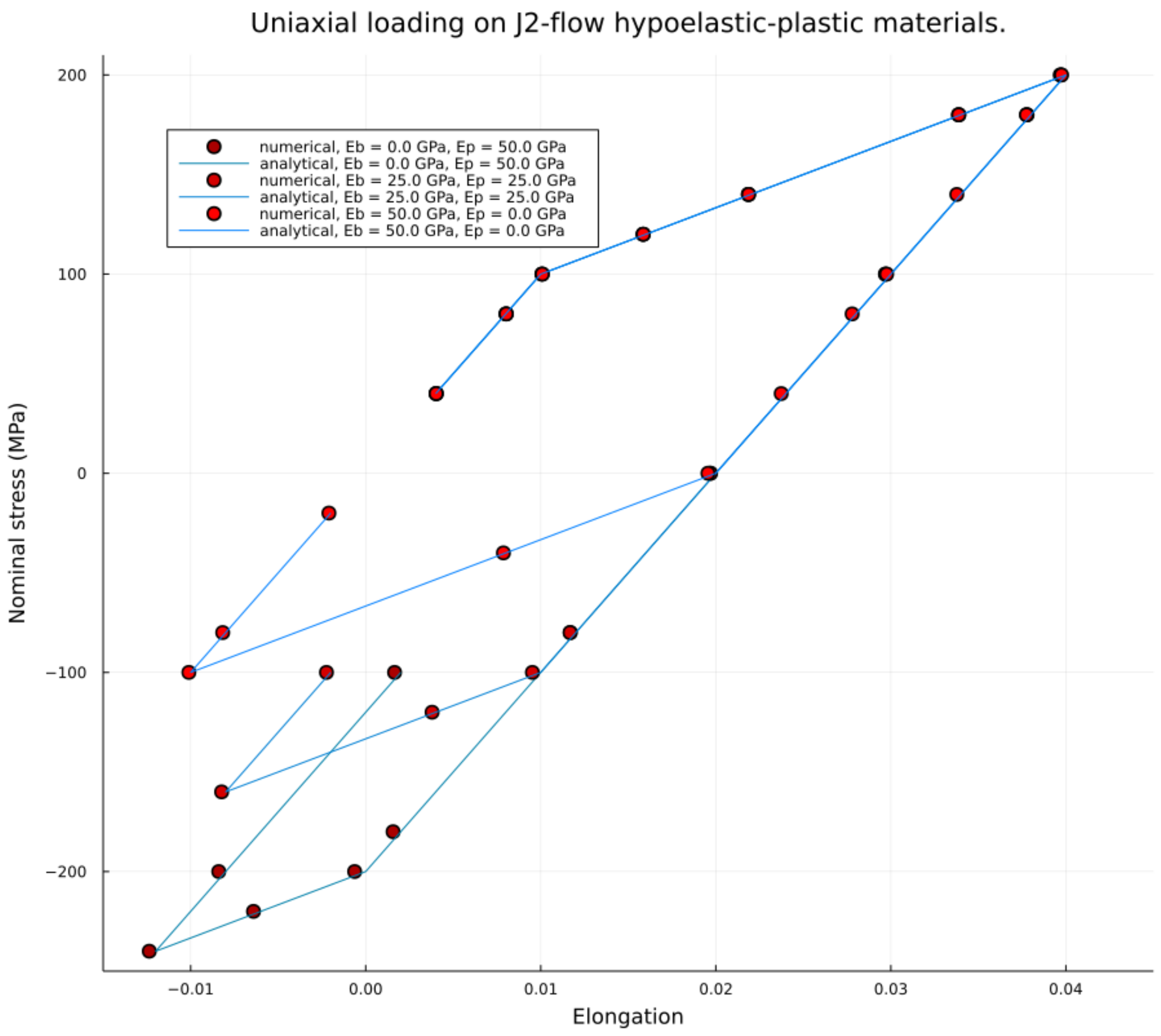


Figure 4.10. The uniaxial tensile test of the J2 hypo-elastic-plastic materials with different hardening parameters.

$$\begin{aligned}&\overbrace{-\rho(u_{i,j}, u_i u_j) - (u_{i,i}, p) + (p, u_{i,i}) + \mu(u_{i,j}, u_{i,j})}^{NS} +\\ &\overbrace{\tau^m \rho(u_{i,j}, Rm_i u_j) + \tau^m (p_{,i}, Rm_i) + \tau^c (u_{i,i}, Rc)}^{SUPG}\\ &= 0, \qquad in \quad \Omega\end{aligned} \tag{4.11}$$

$$(u_i, pn_i) - \mu(u_i, u_{i,j}n_j) = 0, \qquad on \quad \partial\Omega$$

$$\rho(u_i, u_i^w u_j^w n_j) + (p, (u_i^w - u_i)n_i)$$

(4.12)

$$+\mu(u_{i,j}, (u_i^w - u_i)n_j) + \tau^b \rho(u_i, u_i - u_i^w) = 0, \qquad on \quad (\partial\Omega)_{inflow}$$

$$\rho(u_i, u_i u_j n_j) = 0, \qquad on \quad (\partial\Omega)_{outflow}$$

(4.13) $$(p, -u_i n_i) + \mu(u_{i,j}, -u_i n_j) + \tau^b \rho(u_i, u_i) = 0, \qquad on \quad (\partial\Omega)_{fix}$$

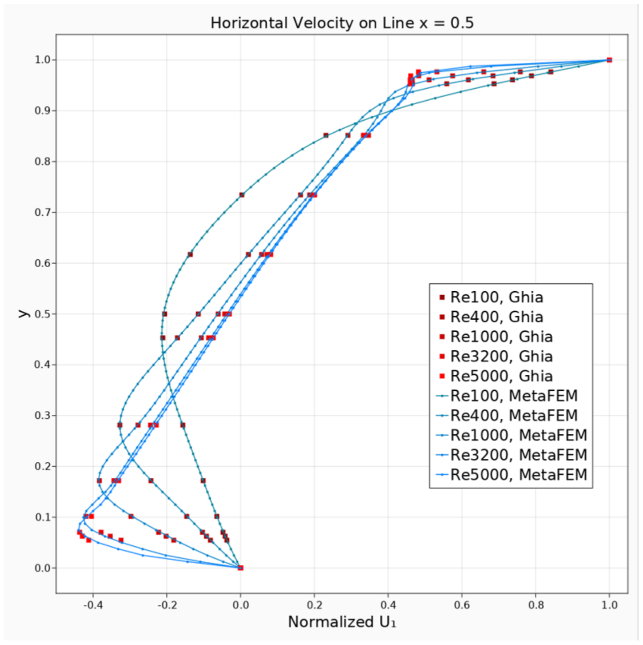
where u_i, p, ρ, μ are the velocity, pressure, density and the dynamic viscosity respectively. The code for physics is given by:

```
@Def begin
    Rc = u{k;k}
    Rm{i} = ρ * u{k} * u{i;k} + p{;i} - μ * u{i;k,k}
end

@Def begin
   NS_domain_BASE = - ρ * Bilinear(u{i;j}, u{i} * u{j}) - Bilinear(u{i;i}, p) +
   Bilinear(p, u{i;i}) + μ * Bilinear(u{i;j}, u{i;j})

   NS_domain_SUPG = τ^m * ρ * Bilinear(u{i;j}, Rm{i} * u{j}) +
   τ^m * Bilinear(p{;i}, Rm{i}) + τ^c * Bilinear(u{i;i}, Rc)
   NS_boundary_BASE = Bilinear(u{i}, p * n{i}) - μ * Bilinear(u{i}, u{i;j} * n{j})


   NS_boundary_INFLOW = ρ * Bilinear(u{i}, uʷ{i} * uʷ{j} * n{j}) +
   Bilinear(p, (uʷ{i} - u{i}) * n{i}) + μ * Bilinear(u{i;j}, (uʷ{i} - u{i}) * n{j}) +
   τᵇ * ρ * Bilinear(u{i}, u{i} - uʷ{i})


   NS_boundary_OUTFLOW = ρ * Bilinear(u{i}, u{i} * u{j} * n{j})


   NS_boundary_FIX = Bilinear(p, - u{i} * n{i}) + μ * Bilinear(u{i;j}, - u{i} * n{j}) +
   τᵇ * ρ * Bilinear(u{i}, u{i})
end


@Def begin
   NS_domain = NS_domain_BASE + NS_domain_SUPG
   NS_boundary_inflow = NS_boundary_BASE + NS_boundary_INFLOW
   NS_boundary_outflow = NS_boundary_BASE + NS_boundary_OUTFLOW
   NS_boundary_fix = NS_boundary_BASE + NS_boundary_FIX
end
```

4.6.1. 2D lid-driven cavity

A cavity of side length $L = 1$ full of fluid $\rho = 1, \mu = 1$ has its left, bottom and right wall fixed and the top wall moving rightward with a velocity u^w determined by the Reynold number, $u^w = Re\frac{\mu}{L\rho}$, is simulated for cases with Re = 100, 400, 1,000, 3,200 and 5,000 with 40×40 quadratic serendipity square elements (dx = 0.025). The horizontal velocity on the middle vertical line $x = 0.5$ is compared between MetaFEM and the result of Ghia

[**28**], as shown in fig. 4.11. The Paraview streamline plot for the example case *Re* =1,000 is also provided.



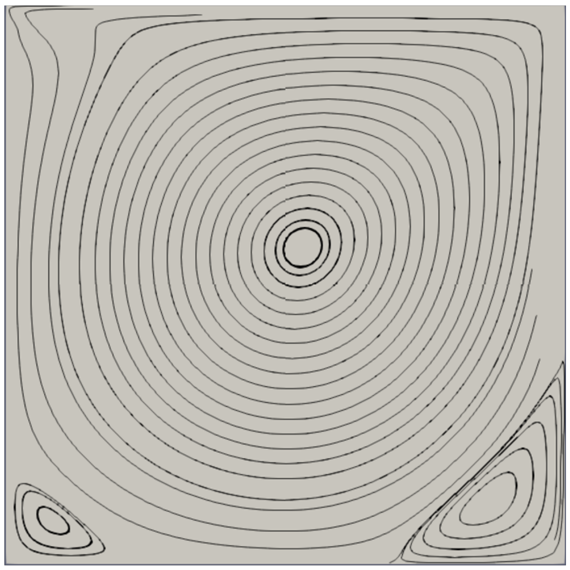


Figure 4.11. The lid-driven cavity flow. Left: Horizontal velocity comparison along the middle vertical line $x = 0.5$; Right: Streamline plot for $Re =$ 1,000

4.6.2. 3D cylinder flow

A cuboid channel of length $L = 2.5$, width and depth $h = 0.41$ with a fixed z-axis oriented cylinder obstacle of radius $r = 0.05$ is positioned with its center at $(x, y) = (0.2, 0.2)$. The 4 channel sides $(y = 0, z = 0, y = 0.41, z = 0.41)$ are fixed while a steady flow ($\rho =$ 1,000, $\mu = 1$) with a fully developed parabolic inflow profile at the left entry $x = 0$ with maximum velocity $U = 0.45$ ($Re = 20$) flows out to right outlet $x = 2.5$. The detailed inflow profile is:

$$u^w(0,y,z) = (\frac{16U(h-y)(h-z)yz}{h^4}, 0, 0) \tag{4.14}$$

The case is simulated in both COMSOL and MetaFEM with 28,468 quadratic simplex elements and with 41,202 mesh nodes. The mesh is shown fig. 4.12.

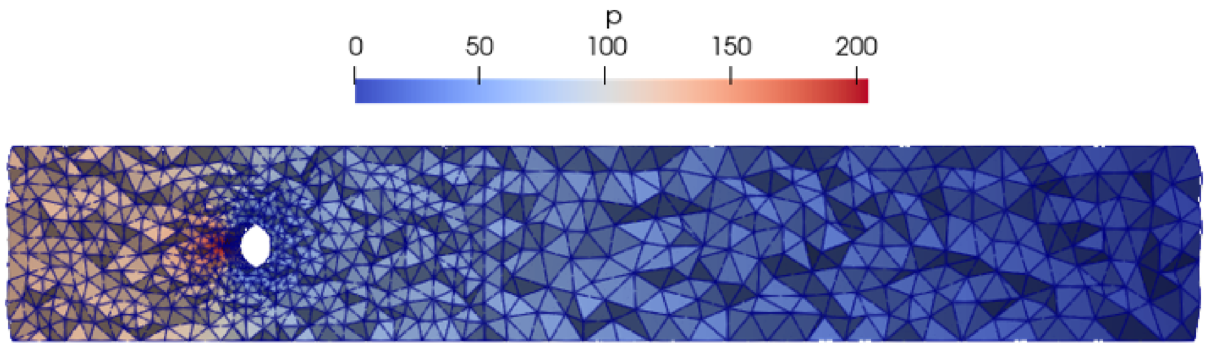
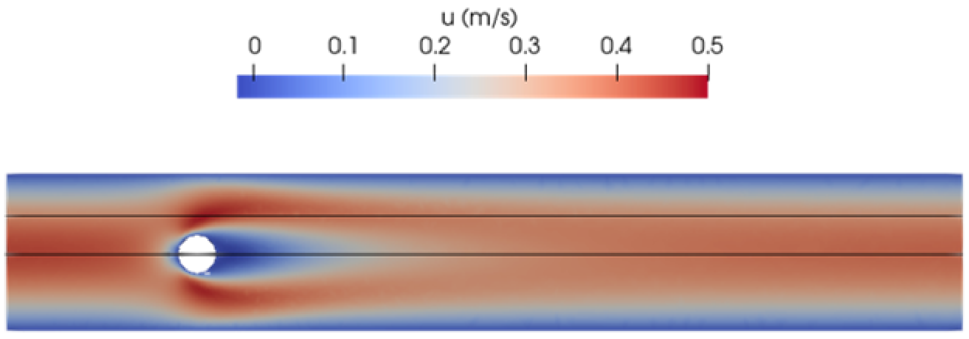
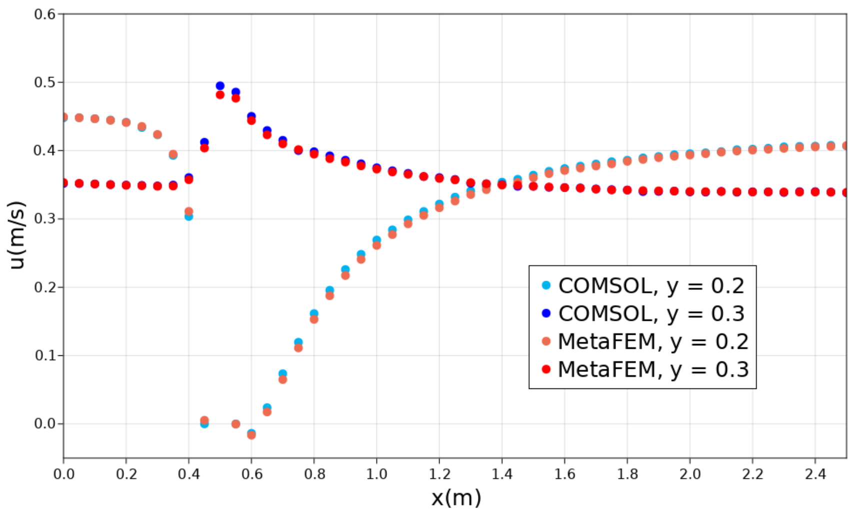


Figure 4.12. The mesh for the cylinder flow, colored with pressure distribution

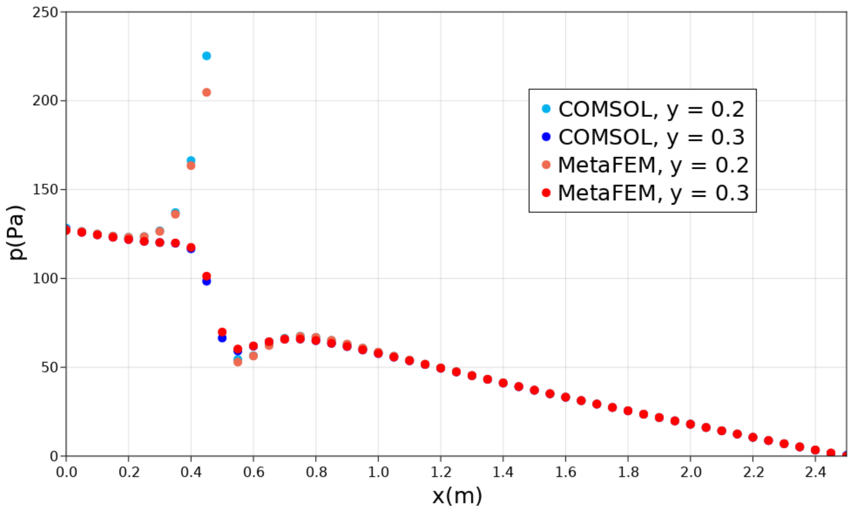
For further verification, we compare the horizontal velocity u_1 and pressure p along two horizontal lines $y = 0.2, z = 0.2$ and $y = 0.3, z = 0.2$, as in fig. 4.13, where the good match indicates that MetaFEM is able to correctly derive and assemble the physics of incompressible laminar flow.



(a) The plan $z = 0.2$ for the sample lines, colored with horizontal velocity distribution



(b) Horizontal velocity distribution comparison.



(c) Pressure comparison.

Figure 4.13. The 3D cylinder flow

CHAPTER 5

Conclusion

This work proposes MetaFEM, a generic finite element solver with original and uncompromising formulations of the theory, the algorithm, and a demonstratable software package. The solver is designed to generate GPU-accelerated FEM simulation of generic physics on generic geometry, and it has already been capable to solve a broad range of problems in thermal conduction, solid mechanics and fluid mechanics. The rewriting-based symbolic engine not only provides a more compact and more mathematical-aligned user grammar, but also enhances both the precision and the flexibility. For example, one mentionable but less obvious advantage is that boundary conditions can be set in a more flexible way than in many other high-level solvers, e.g., a user is able to freely choose a plain penalty method or Nitsche's method [**29**] for a fixed wall instead of using a default "Dirichlet boundary" by simply changing a few bilinear terms.

Still, many areas await future development on the source level, where the foremost topic is the mesh system. The current mesh system only supports a traditional, static mesh, which suffers from mesh distortion under large deformations. Dynamic approaches, like a cut-cell mesh system or mesh-free methods [**33**], are certainly necessary. Also, currently MetaFEM assumes that each workpiece has, for simplicity, only one set of elements of the same order of integration and interpolation, which can be further developed so that the user can independently assign different bilinear forms to a workpiece with an optional element order, thus, for example, enabling selective reduced integration for the volumetric term in the bare displacement formulation or the reduced interpolation order for the pressure term in the mixed formulation for the incompressible problems. Other non-traditional elements are certainly desirable as well, like Discontinuous Galerkin, the IsoGeometric Analysis (IGA) [**26**] for more flexible interpolation and the eXtended Finite Element Method (XFEM) [**35**] for cracks. Contact mechanism is another gap to be filled. Currently all interactions between workpieces have to be user programmed, which is obviously inconvenient, e.g., when friction needs to be considered. Also, the scalability issue needs to be addressed by adapting a distributed framework.

Finally, we conclude the article with our motivation. The ambition of MetaFEM is to be usable both for geometrically complex manufacturing processes and for physically complex academical problems.

References

[1] *Elmer*, https://www.csc.fi/web/elmer/.

[2] *GOMA*, https://www.gomafem.com/.

[3] *List of finite element software packages*, https://en.wikipedia.org/wiki/List_of_finite_element_software_packages.

[4] *MetaFEM*, https://github.com/jxx2/MetaFEM.jl.

[5] *Range3*, https://github.com/Range-Software/range3#readme.

[6] *Z88*, https://github.com/LSCAD/Z88OS.

[7] *Agros2D*, http://www.agros2d.org/.

[8] *OOFEM*, http://oofem.orSg/en/manual.

[9] *CalculiX*, http://www.calculix.de/.

[10] *FEBio*, https://febio.org/.

[11] *MoFEM JosePH*, https://mofem.eng.gla.ac.uk/mofem/html/.

[12] *FreeCAD*, https://www.freecadweb.org/.

[13] *OpenSees*, https://opensees.berkeley.edu/index.php.

[14] *DUNE*, https://www.dune-project.org/modules/dune-fem/.

[15] *deal.II*, https://www.dealii.org/.

[16] *MFEM*, https://mfem.org/.

[17] *Hermes*, https://hpfem.org/wp-content/uploads/doc-web/doc-tutorial/index.html#.

[18] *GetFEM++*, `https://getfem.org/`.

[19] *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*, Lecture Notes in Computational Science and Engineering, Springer, 2012.

[20] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, *Unified form language: A domain-specific language for weak formulations of partial differential equations*, ACM Transactions on Mathematical Software, 40 (2014), p. 1–37, `https://doi.org/10.1145/2566630`, `https://github.com/FEniCS/ufl`.

[21] S. Badia and F. Verdugo, *Gridap: An extensible finite element toolbox in julia*, Journal of Open Source Software, 5 (2020), p. 2520, `https://doi.org/10.21105/joss.02520`.

[22] F. Baader and T. Nipkow, *Term rewriting and all that*, Cambridge Univ. Press, 1. paperback ed ed., 1999.

[23] T. Besard, C. Foket, and B. De Sutter, *Effective extensible programming: Unleashing julia on gpus*, IEEE Transactions on Parallel and Distributed Systems, 30 (2019), p. 827–841, `https://doi.org/10.1109/TPDS.2018.2872064`.

[24] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, *Julia: A fresh approach to numerical computing*, SIAM Review, 59 (2017), p. 65–98, `https://doi.org/10.1137/141000671`.

[25] J. Chung and G. M. Hulbert, *A time integration algorithm for structural dynamics with improved numerical dissipation: The generalized-alpha method*, Journal of Applied Mechanics, 60 (1993), p. 371–375, `https://doi.org/10.1115/1.2900803`.

[26] Cottrell, A., Hughes, T., and Bazilevs, Y. *Isogeometric Analysis: Toward Integration of CAD and FEA*, 1 ed. Wiley, 2009.

[27] FLOWALISTIK, *Low-poly pikachu*, `https://www.thingiverse.com/thing:376601`.

[28] U. Ghia, K. Ghia, and C. Shin, *High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method*, Journal of Computational Physics, 48 (1982), p. 387–411, `https://doi.org/10.1016/0021-9991(82)90058-4`.

[29] HANSBO, A., AND HANSBO, P. An unfitted finite element method, based on Nitsche's method, for elliptic interface problems. *Computer Methods in Applied Mechanics and Engineering 191*, 47-48 (2002), 5537–5552.

[30] F. HECHT, *New development in freefem++*, J. Numer. Math., 20 (2012), pp. 251–265, `https://freefem.org/`.

[31] C. INC., *Simulate real-world designs, devices, and processes with multiphysics software from comsol*, `https://www.comsol.com`.

[32] R. C. KIRBY AND A. LOGG, *A compiler for variational forms*, ACM Transactions on Mathematical Software, 32 (2006), p. 417–444, `https://doi.org/10.1145/1163641.1163644`, `https://github.com/FEniCS/ffcx`.

[33] LIU, W. K., JUN, S., AND ZHANG, Y. F. Reproducing kernel particle methods. *International Journal for Numerical Methods in Fluids 20*, 8-9 (1995), 1081–1106

[34] A. LOGG AND G. N. WELLS, *Dolfin*, ACM Transactions on Mathematical Software, 37 (2010), p. 1–28, `https://doi.org/10.1145/1731022.1731030`, `https://github.com/FEniCS/dolfinx`.

[35] MOËS, N., AND BELYTSCHKO, T. Extended finite element method for cohesive crack growth. *Engineering Fracture Mechanics 69*, 7 (2002), 813–833.

[36] C. J. PERMANN, D. R. GASTON, D. ANDRŠ, R. W. CARLSEN, F. KONG, A. D. LINDSAY, J. M. MILLER, J. W. PETERSON, A. E. SLAUGHTER, R. H. STOGNER, AND R. C. MARTINEAU, *MOOSE: Enabling massively parallel multiphysics simulation*, SoftwareX, 11 (2020), p. 100430, `https://doi.org/https://doi.org/10.1016/j.softx.2020.100430`, `http://www.sciencedirect.com/science/article/pii/S2352711019302973`.

[37] P. SIMULATION, *Heat transfer in a ceramic strip*, `https://www.featool.com/doc/heat_transfer_02_heat_transfer2`.

[38] S. P. TIMOŠENKO AND S. WOINOWSKY-KRIEGER, *Theory of plates and shells*, Engineering societies monographs, McGraw-Hill, 2. ed., internat. ed ed., 1976.

APPENDIX A

MetaFEM v0.1.4

We attached a selected collection of the core source code of MetaFEM v0.1.4 as below, i.e., without helper functions.

A.1. Infrastructure

The code inside the folder /src/misc defines the generic functions, macros, and data structures.

Firstly are some macros.

```
# Adapted from MacroTool.jl
walk(x, inner, outer) = outer(x)
walk(x::Expr, inner, outer) = outer(Expr(x.head, map(inner, x.args)...))

postwalk(f, x) = walk(x, x -> postwalk(f, x), f)
prewalk(f, x)  = walk(f(x), x -> prewalk(f, x), identity)

isexpr(x::Expr) = true
isexpr(x) = false
isexpr(x::Expr, ts...) = x.head in ts
isexpr(x, ts...) = any(T->isa(T, Type) && isa(x, T), ts)

isline(ex) = isexpr(ex, :line) || isa(ex, LineNumberNode)
iscall(ex, f) = isexpr(ex, :call) && ex.args[1] == f
```

```
rmlines(x) = x
function rmlines(x::Expr)
  # Do not strip the first argument to a macrocall, which is required.
  if x.head == :macrocall && length(x.args) >= 2
    Expr(x.head, x.args[1], nothing, filter(x->!isline(x), x.args[3:end])...)
  else
    Expr(x.head, filter(x->!isline(x), x.args)...)
  end
end
striplines(ex) = prewalk(rmlines, ex)

#Personal defination
is_block(ex) = isexpr(ex, :block)
is_collection(ex) = isexpr(ex, :tuple, :vect, :vcat)
rmblock(x) = x
rmblock(x::Expr) = x.head == :block ?
Expr(:block, vcat(map(x -> x.head == :block ? x.args : x, x.args)...)...) : x
stripblocks(ex) = postwalk(rmblock, ex)

function vectorize_Args(arg)
    if is_block(arg)
        return stripblocks(striplines(arg)).args[:]
    elseif is_collection(arg) #$arg isa Expr && $arg.head in [:tuple; :vect; :vcat]
        return arg.args[:]
    elseif arg isa Tuple
        return collect(arg)
    else
        return [arg]
    end
end

macro If_Match(sentence, syntax_structure...)
```

```
    sentence_length = length(syntax_structure) - 1 #the last one is the command

    keys = Pair[]
    term_conditions = Expr[]
    for (index, arg) in enumerate(syntax_structure)
        if arg isa Expr && arg.head == :vect
            push!(keys, index => arg.args[1])
        elseif arg isa Symbol
            push!(term_conditions, :($sentence[$index] == $(Meta.quot(arg))))
        end
    end

    condition = :(length($sentence) == $sentence_length)
    for this_condition in term_conditions
        condition = :($condition && $this_condition)
    end
    content = Expr(:block)
    for (index, name) in keys
        push!(content.args, :($name = $sentence[$index]))
    end
    push!(content.args, syntax_structure[end])
    ex =
    :(if $condition
         $content
      end)
    return esc(ex)
end

"""
    @Takeout a, b FROM c

equals to
```

```
    a = c.a
    b = c.b

while

    @Takeout a, b FROM c WITH PREFIX d

equals to

    da = c.a
    db = c.b
"""
macro Takeout(sentence...)
    ex = Expr(:block)
    arg_tuple = vectorize_Args(sentence[1])
    @If_Match sentence[2:3] FROM [host] for arg in arg_tuple
        argchain = Symbol[]
        this_arg = arg
        while this_arg isa Expr && arg.head == :.
            push!(argchain, this_arg.args[2].value)
            this_arg = this_arg.args[1]
        end
        push!(argchain, this_arg)
        reverse!(argchain)
        target_arg = argchain[end]
        source_arg = host
        for this_arg in argchain
            source_arg = Expr(:., source_arg, QuoteNode(this_arg))
        end
        push!(ex.args, :($target_arg = $source_arg))
    end
    @If_Match sentence[4:end] WITH PREFIX [prefix] for arg in ex.args
        arg.args[1] = Symbol(string(prefix, arg.args[1]))
    end

    return esc(ex)
end

Base.:|>(x1, x2, f) = f(x1, x2) # (a, b)... |> f = f(a, b)
Base.:|>(x1, x2, x3, f) = f(x1, x2, x3)
Base.:|>(x1, x2, x3, x4, f) = f(x1, x2, x3, x4)

reload_Pipe(x) = x
function reload_Pipe(x::Expr)
    (isexpr(x, :call) && x.args[1] == :|>) || return x
    xs, (node_func, extra_args...) = vectorize_Args.(x.args[2:3])
    is_splated = isexpr(node_func, :...)

    if is_splated
        node_func = node_func.args[1]
    end

    res = isexpr(node_func, :call) ? Expr(node_func.head, node_func.args[1], xs...,
    node_func.args[2:end]...) : Expr(:call, node_func, xs...)

    if is_splated
        res = Expr(:..., res)
    end
    return Expr(:tuple, res, extra_args...)
end

macro Pipe(exs)
```

```
    esc(postwalk(reload_Pipe, exs).args[1])
end


macro Construct(typename)
    if typename isa Symbol
        target_type = eval(typename)
    elseif typename.head == :curly
        target_type = eval(typename.args[1])
    end
    fields = fieldnames(target_type)
    return esc(:($typename($(fields...))))
end


const FEM_Int = Int32 # Note, hashing some GPU FEM_Int like indices may assume last
# 30 bits in 2D, 20 bits in 3D, so not really needed anything above Int32
const FEM_Float = Float64 # Double digit make iterative solvers much more robust,
# since we are still simply Jacobi conditioned
VTuple(T...) = Tuple{Vararg{T...}}


const _UNITS = Dict{String, Int}(["B", "KB", "MB", "GB", "TB"] .=> [Int(1 << (10 * i)) for i =
0:4])
mutable struct MemUnit
    u_name::String
    u_size::Int
end


function (_obj::MemUnit)(unit_name::String)
    if unit_name in keys(_UNITS)
        _obj.u_name = unit_name
        _obj.u_size = _UNITS[unit_name]
    end
end
```

```julia
"""
    MEM_UNIT(unit_name::String)

Set the displayed memory unit where `unit_name` can be "B", "KB", "MB", "GB", "TB".
"""
const MEM_UNIT = MemUnit("MB", _UNITS["MB"])

function estimate_msize(x::T, budget::Integer, counter_func::Function) where T
    msize = counter_func(x, budget)
    if isa(x, Dict)
        for dict_val in values(x)
            msize += estimate_msize(dict_val, budget, counter_func)
        end
    elseif isa(x, AbstractArray)

    elseif isstructtype(T)
        for fname in fieldnames(T)
            msize += estimate_msize(getfield(x, fname), budget, counter_func)
        end
    end
    return msize
end # recursively count the total size of an object

CPU_sizeof(x::T, budget::Integer = 0) where T <: AbstractArray =
sizeof(x) + Base.elsize(T) * budget
CPU_sizeof(x, budget::Integer = 0) = sizeof(x)
GPU_sizeof(x::T, budget::Integer = 0) where T <: CuArray = sizeof(x) + Base.elsize(T) * budget
GPU_sizeof(x, budget::Integer = 0) = 0

estimate_memory_CPU(x, budget::Integer = 0) = estimate_msize(x, budget, CPU_sizeof)
estimate_memory_GPU(x, budget::Integer = 0) = estimate_msize(x, budget, GPU_sizeof)
```

```
report_memory(x, _obj::MemUnit) where N =
string("$(estimate_memory_CPU(x)/ _obj.u_size) $(_obj.u_name)
CPU memory and $(estimate_memory_GPU(x)/ _obj.u_size) $(_obj.u_name) GPU memory")
report_memory(x) = report_memory(x, MEM_UNIT)
# Slightly prettier gensym
mutable struct PrefixGenerator
    prefix::String
    ID::Int
    PrefixGenerator(prefix::String) = new(prefix, 0)
end
(_obj::PrefixGenerator)(x = "") = Symbol("$(_obj.prefix)_$(_obj.ID+=1)_$x")
```

The polynomial type:

```
mutable struct Polynomial{dim}
    factors::Vector{FEM_Float}
    orders::Vector{Tuple{Vararg{FEM_Int, dim}}}

    Polynomial(dim::Integer) = new{dim}([FEM_Float(0.)], [const_Tup(FEM_Int(0), dim)])
    Polynomial(factor::AbstractFloat, order::Tuple) = new{length(order)}(
        FEM_Float[factor], [FEM_Int.(order)])
    Polynomial(factors::Vector, orders::Vector) = new{length(orders[1])}(
        FEM_Float.(factors), map(x -> FEM_Int.(x), orders))
    Polynomial(example::Polynomial{dim}) where dim = new{dim}(
        copy(example.factors), copy(example.orders))
end

const_Tup(w, dim_num) = Tuple(fill(w, dim_num))
function basis_Tup(dim_id, dim_num, wi = 1, wo = 0)
```

```
    temp = fill(wo, dim_num)
    temp[dim_id] = wi
    return Tuple(temp)
end
collect_Basis(dim::Integer) = [Polynomial(1., basis_Tup(i, dim)) for i = 1:dim]

function Base.:+(p1::Polynomial{dim}, num::Number) where dim
    p_ans = Polynomial(p1)
    num == 0. && return p_ans

    this_order = const_Tup(FEM_Int(0), dim)
    matched_id = findfirst(x -> x == this_order, p1.orders)
    if isnothing(matched_id)
        push!(p_ans.factors, num)
        push!(p_ans.orders, this_order)
    else
        p_ans.factors[matched_id] += num
        p_ans = check_Clear(p_ans)
    end
    return p_ans
end

Base.:+(num::Number, p1::Polynomial{dim}) where dim = p1 + num

function Base.:+(p1::Polynomial{dim}, p2::Polynomial{dim}) where dim
    p_ans = Polynomial(p1)
    for (this_factor, this_order) in zip(p2.factors, p2.orders)
        matched_id = findfirst(x -> x == this_order, p1.orders)
        if isnothing(matched_id)
            push!(p_ans.factors, this_factor)
            push!(p_ans.orders, this_order)
        else
```

```
            p_ans.factors[matched_id] += this_factor
        end
    end
    return check_Clear(p_ans)
end

function Base.:-(p1::Polynomial{dim}) where dim
    p_ans = Polynomial(p1)
    p_ans.factors .*= -1
    return p_ans
end
Base.:-(p1::Polynomial{dim}, num::Number) where {dim} = p1 + (-num)
Base.:-(num::Number, p1::Polynomial{dim}) where {dim} = -p1 + num
Base.:-(p1::Polynomial{dim}, p2::Polynomial{dim}) where {dim} = p1 + (-p2)

function check_Clear(p1::Polynomial{dim}) where dim
    id_to_remove = findall(x -> x == 0., p1.factors)
    # not_empty = p1.factors .!= 0
    err = 1e-8
    not_empty = abs.(p1.factors) .>= err
    if sum(not_empty) == 0
        p1.factors = [FEM_Float(0.)]
        p1.orders = [const_Tup(FEM_Int(0), dim)]
    else
        p1.factors = p1.factors[not_empty]
        p1.orders = p1.orders[not_empty]
    end
    return p1
end

function Base.:*(p1::Polynomial{dim}, num::Number) where dim
    p_ans = Polynomial(p1)
    if num == 0
        p_ans.factors = [FEM_Float(0.)]
        p_ans.orders = [const_Tup(FEM_Int(0), dim)]
    else
        p_ans.factors .*= num
    end
    return p_ans
end

Base.:*(num::Number, p1::Polynomial{dim}) where dim = p1 * num
Base.:/(p1::Polynomial{dim}, num::Number) where dim = p1 * (1/num)

function Base.:*(p1::Polynomial{dim}, p2::Polynomial{dim}) where dim
    p_ans = Polynomial(dim)
    for (first_factor, first_order) in zip(p1.factors, p1.orders)
        for (second_factor, second_order) in zip(p2.factors, p2.orders)
            this_factor = first_factor * second_factor
            this_order = first_order .+ second_order

            matched_id = findfirst(x -> x == this_order, p_ans.orders)
            if isnothing(matched_id)
                push!(p_ans.factors, this_factor)
                push!(p_ans.orders, this_order)
            else
                p_ans.factors[matched_id] += this_factor
            end
        end
    end
    return check_Clear(p_ans)
end

function Base.:^(p1::Polynomial{dim}, num::Integer) where dim
```

```
    p_ans = Polynomial(FEM_Float(1.), const_Tup(FEM_Int(0), dim))
    for i = 1:num
        p_ans *= p1
    end
    return p_ans
end

function substitute_Polynomial(p_src::Polynomial{dim1},
src_dim::Integer, p_template::Polynomial{dim2}) where {dim1, dim2}
    p_ans = Polynomial(dim2)
    for (this_factor, p_src_order) in zip(p_src.factors, p_src.orders)
        p_core = p_template ^ p_src_order[src_dim]
        p_base = Polynomial(this_factor, ntuple(x -> ((x == src_dim) ||
        (x > dim1)) ? 0 : p_src_order[x], dim2))
        p_ans += p_core * p_base
    end
    return p_ans
end

function derivative(p1::Polynomial{dim}, orders::Tuple) where dim
    p2 = Polynomial(p1)
    for i = 1:length(p2.factors)
        this_order = p2.orders[i]
        if minimum(this_order.- orders) < 0
            p2.factors[i] = FEM_Int(0)
            continue
        end
        for this_dim = 1:dim
            p2.factors[i] *= factorial(this_order[this_dim])/
            factorial(this_order[this_dim] - orders[this_dim])
        end
        p2.orders[i] = this_order .- orders
    end
    return check_Clear(p2)
end

function evaluate_Polynomial(p1::Polynomial{dim}, pos::Tuple) where dim
    sum = 0
    for (this_factor, this_order) in zip(p1.factors, p1.orders)
        this_term = prod(pos .^ this_order)
        sum += this_term * this_factor
    end
    return sum
end
```

Then is the memory management.

```
GPU_DeviceArray{T, N} = CuArray{T, N, CUDA.Mem.DeviceBuffer} where {T, N}
GPU_UnifiedArray{T, N} = CuArray{T, N, CUDA.Mem.UnifiedBuffer} where {T, N}

mutable struct ArrayDescriptor
    _type::Type
end
# const DEFAULT_ARRAYINFO = ArrayDescriptor(GPU_DeviceArray)
const DEFAULT_ARRAYINFO = ArrayDescriptor(GPU_UnifiedArray)

for src_func in (:zeros, :ones, :rand)
    FEM_func = Symbol("FEM_$(src_func)")
    @eval begin
        $FEM_func(element_type, dims...) =
        $FEM_func(DEFAULT_ARRAYINFO._type, element_type, dims...)
        $FEM_func(::Type{Array}, ::Type{ElementType}, dims::Number...) where
        ElementType = $src_func(ElementType, dims...)
        $FEM_func(::Type{GPU_DeviceArray}, ::Type{ElementType}, dims::Number...) where
        ElementType = (CUDA.$src_func)(ElementType, dims...)
    end
end
FEM_zeros(::Type{GPU_UnifiedArray}, ::Type{ElementType}, dims::Number...) where ElementType =
fill!(CuArray{ElementType, length(dims), CUDA.Mem.UnifiedBuffer}(undef, dims...),
zero(ElementType))
FEM_ones(::Type{GPU_UnifiedArray}, ::Type{ElementType}, dims::Number...) where ElementType =
fill!(CuArray{ElementType, length(dims), CUDA.Mem.UnifiedBuffer}(undef, dims...),
one(ElementType))
FEM_rand(::Type{GPU_UnifiedArray}, ::Type{ElementType}, dims::Number...) where ElementType = C
UDA.Random.rand!(CuArray{ElementType, length(dims), CUDA.Mem.UnifiedBuffer}(undef, dims...))

FEM_buffer(::Type{GPU_UnifiedArray}, ::Type{ElementType}, dims::Number...) where
ElementType = CUDA.zeros(ElementType, dims...)
FEM_buffer(::Type{ArrayType}, ::Type{ElementType}, dims::Number...) where {ArrayType,
ElementType} = FEM_zeros(ArrayType, ElementType, dims...)

FEM_ArrayTypes = (:Array, :GPU_DeviceArray, :GPU_UnifiedArray)

FEM_convert(src) = FEM_convert(DEFAULT_ARRAYINFO._type, src)
FEM_convert(::Type{Array}, src::Array) = src
FEM_convert(::Type{Array}, src::AbstractArray{T,N}) where {T, N} = collect(src)
FEM_convert(::Type{GPU_DeviceArray}, src::CuArray) = src
FEM_convert(::Type{GPU_DeviceArray}, src::AbstractArray{T,N}) where {T, N} =
CuArray{T, N, CUDA.Mem.DeviceBuffer}(src)

FEM_convert(::Type{GPU_UnifiedArray}, src::CuArray) = src
FEM_convert(::Type{GPU_UnifiedArray}, src::AbstractArray{T,N}) where {T, N} =
CuArray{T, N, CUDA.Mem.UnifiedBuffer}(src)
```

```
function rewrite_Title_ArgTypes(ex::Expr, t_mapping::Dict = Dict())
    for i = 2:length(ex.args)
        if (ex.args[i] isa Expr) && (ex.args[i].head == :(::))
            original_type_sym = ex.args[i].args[2]
            if original_type_sym in keys(t_mapping)
                ex.args[i].args[2] = t_mapping[original_type_sym]
            else
                ex.args[i] = ex.args[i].args[1]
            end
        end
    end
    ex
end

const GPU_Kernel_Generator = PrefixGenerator("GPU")
const GPU_BLOCK_SIZE = 256
function GPU_Single_Kernel(title, atom_content)
    typed_func_title = rewrite_Title_ArgTypes(deepcopy(title), Dict(:Array => :CuArray))
    untyped_func_title = rewrite_Title_ArgTypes(deepcopy(title))

    untyped_func_title.args[1] = GPU_Kernel_Generator(untyped_func_title.args[1])
    major_arr_sym = untyped_func_title.args[end]

    func_ex = Expr(:function, typed_func_title, :(begin
        (@cuda blocks = ceil(Int, size($major_arr_sym)[end] / $GPU_BLOCK_SIZE)
        threads = $GPU_BLOCK_SIZE $untyped_func_title)
    end))

    kernel_ex = Expr(:function, untyped_func_title, :(begin
        thread_idx = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        if thread_idx <= size($major_arr_sym)[end]
            $atom_content
        end
        return
    end))
    return func_ex, kernel_ex
end

macro Dumb_GPU_Kernel(title, atom_content)
    func_ex, kernel_ex = GPU_Single_Kernel(title, atom_content)
    esc(:(begin
        $kernel_ex
        $func_ex
    end))
end

# FEM_sparse COO -> CSR
# lacks CPU COO, distributed COD (bitonic sorting)
function sort_CUSPARSE_COO!(n::Integer, K_I, K_J) # need to rewrite by distributed bitonic ones
    K_length = length(K_I)
    P = Int32.(findall(CUDA.ones(Bool, K_length)) .- 1) #!!!may need to change
    typeof(P)
    function bufferSize()
        out = Ref{Csize_t}(1)
        CUDA.CUSPARSE.cusparseXcoosort_bufferSizeExt(CUDA.CUSPARSE.handle(), n, n,
        K_length, K_I, K_J, out)
        return out[]
    end

    CUDA.with_workspace(bufferSize) do buffer
        CUDA.CUSPARSE.cusparseXcoosortByRow(CUDA.CUSPARSE.handle(), n, n, K_length,
        K_I, K_J, P, buffer)
    end
    return P .+ 1
```

```
end
# lacs CPU & distributed
for ArrayType in FEM_ArrayTypes
    @eval begin
        function generate_J_ptr(Is::$ArrayType{FEM_Int}, m::Integer)
            J_ptr = FEM_zeros($ArrayType, FEM_Int, m + 1)
            J_ptr .= length(Is) + 1

            compress_CSR!(J_ptr, Is)
            return J_ptr
        end
    end
end

shifted_Pointer(arr, offset) = pointer(arr) + sizeof(eltype(arr)) * (offset - 1)
@Dumb_GPU_Kernel compress_CSR!(J_ptr::Array, Is::Array) begin
    CUDA.atomic_min!(shifted_Pointer(J_ptr, Is[thread_idx]), eltype(J_ptr)(thread_idx))
end

FEM_SpMat_CSR(J_ptr::CuArray, Js::CuArray, Ks::CuArray, dim::Tuple) =
CuSparseMatrixCSR(J_ptr, Js, Ks, Int64.(dim))

@Dumb_GPU_Kernel inc_Num!(num::Array, vals::Array, IDs::Array) begin
    CUDA.@atomic num[IDs[thread_idx]] += vals[thread_idx]
end

@Dumb_GPU_Kernel inc_Num!(num::Array, this_val, IDs::Array) begin
    CUDA.@atomic num[IDs[thread_idx]] += this_val
end

# LinearAlgebras
mul!(b::CuVector{T}, A::CuSparseMatrixCSR{T}, x::CuVector{T}, alpha::Number = 1.,
beta::Number = 0.) where T = CUDA.CUSPARSE.mv!('N', T(alpha), A, x, T(beta), b, 'O')
tmul!(b::CuVector{T}, A::CuSparseMatrixCSR{T}, x::CuVector{T}, alpha::Number = 1.,
beta::Number = 0.)  where T = CUDA.CUSPARSE.mv!('T', T(alpha), A, x, T(beta), b, 'O')

function Base.:*(A::CuSparseMatrixCSR{T}, x::CuVector{T}) where T #!!!may need to change
    b = CUDA.zeros(T, size(A, 1))
    mul!(b, A, x)
    return b
end
```

Then is the GPU Table, i.e., struct of arrays.

```
const TABLE_INITIAL_SIZE = 1024

abstract type Abstract_Table{ArrayType} end
get_Data(this_table::Abstract_Table) = getfield(this_table, :data)
Base.getproperty(this_table::Abstract_Table, key::Symbol) = get_Data(this_table)[key]
Base.setproperty!(this_table::Abstract_Table, key::Symbol, value) =
setindex!(get_Data(this_table), value, key)

struct FEM_Table{ArrayType} <: Abstract_Table{ArrayType}
    data::Dict{Symbol, ArrayType}
end

construct_FEM_Table(example_typed, example_direct::Vector = []) =
construct_FEM_Table(DEFAULT_ARRAYINFO._type,
example_typed, example_direct)
function construct_FEM_Table(::Type{ArrayType}, example_typed::ExampleType,
example_direct::Vector = []) where {ArrayType, ExampleType}
    names_direct, values_direct = getindex.(example_direct, 1), getindex.(example_direct, 2)
    names_typed = Symbol[name for name in fieldnames(ExampleType) if ~(name in names_direct)]
    values_typed = Any[getfield(example_typed, name) for name in names_typed]

    if ExampleType <: AbstractArray
        push!(names_typed, :arr)
        push!(values_typed, example_typed)
    end

    data_names = Symbol[names_direct..., names_typed..., :is_occupied]
    data_values = Any[values_direct..., values_typed..., false]

    data_vectors = [dvalue isa AbstractArray ? FEM_zeros(ArrayType, eltype(dvalue),
    size(dvalue)..., TABLE_INITIAL_SIZE) :
    FEM_zeros(ArrayType, typeof(dvalue), TABLE_INITIAL_SIZE) for dvalue in data_values]
    return FEM_Table{ArrayType}(Dict(data_names .=> data_vectors))
end

function extend!(this_table::FEM_Table{ArrayType}, final_size::Integer) where {ArrayType}
    _data = get_Data(this_table)
    allocated_IDs = allocated(this_table)

    for (name, arr) in _data
        if ndims(arr) == 1
            new_arr = FEM_zeros(ArrayType, eltype(arr), final_size)
            new_arr[allocated_IDs] .= arr[allocated_IDs]
        else
            effective_dims = size(arr)[1:(end - 1)]
            effective_ranges = Colon().(1, effective_dims)
            new_arr = FEM_zeros(ArrayType, eltype(arr), effective_dims..., final_size)
            new_arr[effective_ranges..., allocated_IDs] .= arr[effective_ranges...,
            allocated_IDs]
        end
        _data[name] = new_arr
    end
end

available(this_table::FEM_Table) = findall(.~(this_table.is_occupied))
allocated(this_table::FEM_Table) = findall(this_table.is_occupied)
tablelength(this_table::FEM_Table) = length(this_table.is_occupied)
Base.deleteat!(this_table::FEM_Table, IDs) = (this_table.is_occupied[IDs] .= false;
return this_table)

function allocate_by_length!(this_table::FEM_Table, allocating_number::Integer)
    available_length = sum(.~(this_table.is_occupied))
    allocating_number > available_length && extend!(this_table,
    Int(ceil((allocating_number - available_length) /
    tablelength(this_table) + 1) * tablelength(this_table)))

    available_IDs = available(this_table)
    allocating_IDs = available_IDs[1:allocating_number]
    this_table.is_occupied[allocating_IDs] .= true
    return allocating_IDs
end
```

Finally is the GPU hashtable.

```
#Thomas Wang'this_table hashing, the same as julia base
function GPU_hash_64_64(x::UInt64)
    a = x
    a = ~a + a << 21
    a =  a ⊻ a >> 24
    a =  a + a << 3 + a << 8
    a =  a ⊻ a >> 14
    a =  a + a << 2 + a << 4
    a =  a ⊻ a >> 28
    a =  a + a << 31
    return a
end
_DictSize(x::Integer) = x < (16 * 2 / 3) ? 16 : 1 << (64 - leading_zeros(Int(ceil(x * 1.5) - 1)))
_DictSize(arr::AbstractVector) = _DictSize(length(arr))

struct FEM_Dict{ArrayType} <: Abstract_Table{ArrayType}
    data::Dict{Symbol, ArrayType}
    var_names::Vector{Symbol}
end

get_VarNames(this_table::FEM_Dict) = getfield(this_table, :var_names)
get_Total_IDs(this_table::FEM_Dict) = findall(this_table.keys .!= 0) #Note dict key cannot be 0
tablelength(this_table::FEM_Dict) = length(this_table.keys)

construct_FEM_Dict(data_external::Vector = [], size_hint::Integer = 16) =
construct_FEM_Dict(DEFAULT_ARRAYINFO._type, data_external, size_hint)
function construct_FEM_Dict(::Type{ArrayType}, data_external::Vector = [],
size_hint::Integer = 16) where ArrayType
    names_external, values_external = getindex.(data_external, 1), getindex.(data_external, 2)
    data_names = Symbol[names_external..., :keys, :hashs, :hash_init, :hash_prev, :hash_next]
    data_values = Any[values_external..., UInt64(0), UInt64(0), Int32(0), Int32(0), Int32(0)]
    dict_size = _DictSize(size_hint)

    data_vectors = ArrayType[dvalue isa AbstractArray ? FEM_zeros(ArrayType, eltype(dvalue),
    size(dvalue)..., dict_size) :
    FEM_zeros(ArrayType, typeof(dvalue), dict_size) for dvalue in data_values]
    return FEM_Dict(Dict{Symbol, ArrayType}(data_names .=> data_vectors), names_external)
end
```

```
dumb_FEM_Dict_Init(::Type{ElementType}) where ElementType =
dumb_FEM_Dict_Init(DEFAULT_ARRAYINFO._type, ElementType)
dumb_FEM_Dict_Init(::Type{ArrayType}, ::Type{ElementType}) where {ArrayType, ElementType} =
construct_FEM_Dict(ArrayType, [:vals => zero(ElementType)], 16)
function dumb_FEM_Dict_Init(::Type{ArrayType}, new_keys::AbstractArray{UInt64, 1},
new_vals::AbstractArray{ElementType, 1}) where {ArrayType, ElementType}
    new_dict = dumb_FEM_Dict_Init(ArrayType, ElementType)
    new_IDs = FEM_Dict_SetID!(new_dict, new_keys)
    new_dict.vals[new_IDs] .= FEM_convert(ArrayType, new_vals)
    return new_dict
end

function FEM_Dict_SetID!(source_dict::FEM_Dict{ArrayType},
new_keys::AbstractArray{UInt64, 1}) where ArrayType
    if isempty(new_keys)
        return FEM_zeros(ArrayType, Int32, 0)
    end

    @Takeout (keys, hashs, hash_init, hash_prev, hash_next) FROM source_dict
    typed_new_keys = FEM_convert(ArrayType, new_keys)
    raw_size = length(typed_new_keys)
    dict_size = length(keys)
    source_IDs = get_Total_IDs(source_dict)
    source_dict_size = length(source_IDs)
    estimate_dict_size = _DictSize(raw_size + source_dict_size)
    new_IDs = FEM_zeros(ArrayType, Int32, raw_size)

    if estimate_dict_size == dict_size
        dict_SetID!(keys, hashs, hash_init, hash_prev, hash_next, typed_new_keys, new_IDs)
    else
        expanded_keys = FEM_zeros(ArrayType, UInt64, estimate_dict_size)
        expanded_hashs = FEM_zeros(ArrayType, UInt64, estimate_dict_size)

        expanded_hash_init = FEM_zeros(ArrayType, Int32, estimate_dict_size)

        expanded_hash_prev = FEM_zeros(ArrayType, Int32, estimate_dict_size)

        expanded_hash_next = FEM_zeros(ArrayType, Int32, estimate_dict_size)


        if source_dict_size > 0

            source_keys = keys[source_IDs]

            mapped_IDs = FEM_zeros(ArrayType, Int32, source_dict_size)

            dict_SetID!(expanded_keys, expanded_hashs, expanded_hash_init, expanded_hash_prev,

            expanded_hash_next, source_keys, mapped_IDs)

        end


        source_data = get_Data(source_dict)

        for var_name in get_VarNames(source_dict)

            var_data = source_data[var_name]

            mapped_data = FEM_zeros(ArrayType, eltype(var_data), estimate_dict_size)

            if source_dict_size > 0

                mapped_data[mapped_IDs] .= var_data[source_IDs]

            end

            source_data[var_name] = mapped_data

        end


        source_dict.keys = expanded_keys

        source_dict.hashs = expanded_hashs

        source_dict.hash_init = expanded_hash_init

        source_dict.hash_prev = expanded_hash_prev

        source_dict.hash_next = expanded_hash_next


        dict_SetID!(expanded_keys, expanded_hashs, expanded_hash_init, expanded_hash_prev,

        expanded_hash_next, typed_new_keys, new_IDs)

    end

    return new_IDs
```

```
end


function FEM_Dict_GetID(source_dict::FEM_Dict{ArrayType},
target_keys::AbstractArray{UInt64, 1}) where ArrayType
    @Takeout (keys, hashs, hash_init, hash_next) FROM source_dict
    target_IDs = FEM_zeros(ArrayType, Int32, length(target_keys))
    dict_GetID(keys, hashs, hash_init, hash_next, FEM_convert(ArrayType, target_keys), target_IDs)
    return target_IDs
end


dumb_FEM_Dict_Get(source_dict::FEM_Dict, target_keys::AbstractArray{UInt64, 1}) =
source_dict.vals[FEM_Dict_GetID(source_dict, target_keys)]


function FEM_Dict_DelID!(source_dict::FEM_Dict{ArrayType},
del_keys::AbstractArray{UInt64, 1}) where ArrayType
    @Takeout (keys, hashs, hash_init, hash_prev, hash_next) FROM source_dict
    del_infos = FEM_Dict_GetID(source_dict, del_keys)
    del_IDs = del_infos[del_infos .> 0]
    is_deleted = FEM_zeros(ArrayType, Bool, length(keys))
    is_deleted[del_IDs] .= true

    dict_DelID_Reconnect!(keys, hashs, hash_init, hash_prev, hash_next, is_deleted)
    keys[del_IDs] .= UInt64(0)
    hashs[del_IDs] .= UInt64(0)
    hash_prev[del_IDs] .= Int32(0)
    hash_next[del_IDs] .= Int32(0)
    return del_IDs
end


function dumb_FEM_Dict_Del!(source_dict::FEM_Dict, del_keys::AbstractArray{UInt64, 1})
    source_dict.vals[FEM_Dict_DelID!(source_dict, del_keys)] .= 0
    return
```

```
end


update_TruncID(prev_ID, current_size) = ((prev_ID % Int32) & Int32(current_size - 1)) + Int32(1)
@Dumb_GPU_Kernel dict_SetID!(dict_keys::Array, hashs::Array, hash_init::Array, hash_prev::Array,
hash_next::Array, new_keys::Array, new_IDs::Array) begin
    this_key = new_keys[thread_idx]
    this_hash = GPU_hash_64_64(this_key)

    current_size = length(dict_keys)
    start_ID = update_TruncID(this_hash, current_size)
    if hash_init[start_ID] == 0
        this_ID = start_ID
    else
        this_ID = hash_init[start_ID]
    end
    last_front_ID = Int32(0)
    while true
        local_key = CUDA.atomic_cas!(pointer(dict_keys) + sizeof(eltype(dict_keys)) *
        (this_ID - 1), UInt64(0), this_key) #Note dict key cannot be 0
        if local_key == 0 #write to new slot
            hashs[this_ID] = this_hash
            if (last_front_ID != 0) # && (last_front_ID != this_ID) #Not necessary
                hash_prev[this_ID] = last_front_ID
                hash_next[last_front_ID] = this_ID
            else
                hash_init[start_ID] = this_ID
            end
            break
        elseif local_key == this_key #overwrite
            break
        elseif update_TruncID(GPU_hash_64_64(local_key), current_size) == start_ID
            if hash_next[this_ID] == Int32(0)
```

```
                last_front_ID = this_ID
                this_ID = update_TruncID(this_ID, current_size)
            else
                this_ID = hash_next[this_ID]
            end
        else
            this_ID = update_TruncID(this_ID, current_size)
        end
    end
    new_IDs[thread_idx] = this_ID
end

@Dumb_GPU_Kernel dict_GetID(dict_keys::Array, hashs::Array, hash_init::Array, hash_next::Array,
target_keys::Array, target_IDs::Array) begin
    target_key = target_keys[thread_idx]
    current_size = length(dict_keys)
    start_ID = update_TruncID(GPU_hash_64_64(target_key), current_size)
    this_ID = hash_init[start_ID]

    if this_ID == 0
        target_IDs[thread_idx] = Int32(0)
        return
    end

    while true
        if dict_keys[this_ID] == target_key # Found
            target_IDs[thread_idx] = this_ID
            return
        elseif ~(update_TruncID(hashs[this_ID], current_size) == start_ID) #error
            target_IDs[thread_idx] = Int32(-1)
            return
        elseif hash_next[this_ID] == 0 # Not exist
```

```
            target_IDs[thread_idx] = Int32(0)
            return
        end
        this_ID = hash_next[this_ID]
    end
end

@Dumb_GPU_Kernel dict_DelID_Reconnect!(dict_keys::Array, hashs::Array, hash_init::Array,
hash_prev::Array, hash_next::Array, is_deleted::Array) begin
    if is_deleted[thread_idx]
        this_hash = hashs[thread_idx]
        start_ID = update_TruncID(this_hash, length(dict_keys))

        source_prev_ID = hash_prev[thread_idx]
        source_next_ID = hash_next[thread_idx]
        exist_prev_ID = Int32(0)
        local_prev_ID = source_prev_ID
        while true
            if (local_prev_ID == Int32(0)) || (~is_deleted[local_prev_ID])
                exist_prev_ID = local_prev_ID
                break
            end
            local_prev_ID = hash_prev[local_prev_ID]
        end
        exist_next_ID = Int32(0)
        local_next_ID = source_next_ID
        while true
            if (local_next_ID == Int32(0)) || (~is_deleted[local_next_ID])
                exist_next_ID = local_next_ID
                break
            end
            local_next_ID = hash_next[local_next_ID]
        end


        if source_prev_ID == Int32(0)
            hash_init[start_ID] = exist_next_ID
        end
        if (exist_prev_ID != Int32(0)) && (source_prev_ID == exist_prev_ID)
            hash_next[exist_prev_ID] = exist_next_ID
        end
        if (exist_next_ID != Int32(0)) && (source_next_ID == exist_next_ID)
            hash_prev[exist_next_ID] = exist_prev_ID
        end
    end
end
to_UInt64(x) = Int(x) % UInt64
to_UInt64(x::Integer) = x % UInt64
to_UInt64_32(x) = to_UInt64(x) & (UInt32(0) - UInt32(1))
to_UInt64_30(x) = to_UInt64(x) & ((UInt32(0) - UInt32(1)) >> 2)
to_UInt64_20(x) = to_UInt64(x) & ((UInt32(0) - UInt32(1)) >> 12)

I32I32_To_UI64(x, y) = (to_UInt64_32(x) << 32) + to_UInt64_32(y)
UI64_To_UpperHalf(x::UInt64) = ((x >> 32) & (UInt32(0) - UInt32(1))) % Int32
UI64_To_LowerHalf(x::UInt64) = (x & (UInt32(0) - UInt32(1))) % Int32

I4I30I30_To_UI64(x, y, z) = (to_UInt64(x + 1) << 60) + (to_UInt64_30(y) << 30) + to_UInt64_30(z)
I4I20I20I20_To_UI64(x, y, z, w) = (to_UInt64(x + 1) << 60) + (to_UInt64_20(y) << 40) +
(to_UInt64_20(z) << 20) + to_UInt64_20(w) # when 0, 0, 0 key should not be 0

UI64_To_64_60(x::UInt64) = ((x >> 60) & (UInt32(0) - UInt32(1))) % Int32 - 1
UI64_To_60_30(x::UInt64) = Int32(((x << 4) % Int64) >> 34)
UI64_To_30_00(x::UInt64) = Int32(((x << 34) % Int64) >> 34)
UI64_To_60_40(x::UInt64) = Int32(((x << 4) % Int64) >> 44)
UI64_To_40_20(x::UInt64) = Int32(((x << 24) % Int64) >> 44)
```

```
UI64_To_20_00(x::UInt64) = Int32(((x << 44) % Int64) >> 44)
```

A.2. CAS

The code inside the folder /src/symbolics defines the CAS.

```
VARIABLE_ATTRIBUTES = Dict{Symbol, Vector{Symbol}}()

function enum_Local(variable_set, attr::Vector{Symbol} = Symbol[])

    ex = Expr(:block)

    for sub_arg in variable_set

        arg_tuple = vectorize_Args(sub_arg)

        sym = arg_tuple[1]

        param = arg_tuple[2:end]


        push!(ex.args, :(

        begin

            VARIABLE_ATTRIBUTES[$(Meta.quot(sym))] = $(Symbol[param..., attr...])

            println($(Meta.quot(sym)), " is declared")

        end))

    end

    return ex

end


"""

    @Sym (name, attribute_1, attribute_2,...)

    @External_Sym (name, attribute_1, attribute_2,...)


`@Sym` declares a variable symbol `name` with the attributes `attribute_1`, `attribute_2`, ...,

which is simply stored as the pair `name` => (`:INTERNAL_VAR`, `attribute_1`, `attribute_2`, ...)
```

```
in the exposed global dictionary `VARIABLE_ATTRIBUTES`.

`@External_Sym` is just the `@Sym` with the default attribute:
`:EXTERNAL_VAR` instead of `:INTERNAL_VAR`.
"""
macro Sym(variable_set...)
    return esc(enum_Local(variable_set, [:INTERNAL_VAR]))
end

macro External_Sym(variable_set...)
    return esc(enum_Local(variable_set, [:EXTERNAL_VAR]))
end

const IndexSym = Union{FEM_Int, Symbol}
mutable struct SymbolicWord
    base_variable::Symbol
    td_order::FEM_Int
    c_ids::Vector{IndexSym}
    sd_ids::Vector{IndexSym}
    SymbolicWord(base_variable::Symbol) = new(base_variable, FEM_Int(0), IndexSym[], IndexSym[])
    function SymbolicWord(base_variable, td_order, c_ids, sd_ids)
        if length(c_ids) == 2
            (:SYMMETRIC_TENSOR in get_VarAttribute(base_variable)) && sort!(c_ids)
        end
        new(base_variable, td_order, c_ids, sd_ids)
    end
end

mutable struct SymbolicTerm # each term only belongs to one parent
    operation::Symbol
    subterms::Vector #Union{Number, SymbolicWord, SymbolicTerm}
    free_index::Vector{Symbol} #The free index this SymbolicTerm holds
```

```julia
    dumb_index::Vector{Symbol} #The dumb index this SymbolicTerm holds

    SymbolicTerm(operation, subterms, free_index, dumb_index) = new(operation,

    convert(Vector{GroundTerm}, subterms), free_index, dumb_index)

end


const GroundTerm = Union{FEM_Float, SymbolicWord, SymbolicTerm}

mutable struct FunctionVariable

    operation::Symbol

    tag::Val

    subterms::Vector

end


mutable struct SubtermVariable

    sym::Symbol

    tag::Val

end

GeneralTerm = Union{FunctionVariable, SubtermVariable, GroundTerm}


# op-node: fixed operation, idenpendent operation, dependent operation

# fixed single term (no tagged),

# subterm-node: independent_single, dependent_single term,

# independent_free term, dependent_free term,

# independent_inferrable term dependent_inferrable term

const FIXED_OP, IDPDT_OP, DPDT_OP, FIXED_SINGLE, IDPDT_SINGLE,

DPDT_SINGLE, IDPDT_FREE, DPDT_FREE, IDPDT_INFER = [Val{i} for i = 0:8]


struct Matcher # just a simple DFS (with subnodes eagerly expanded)

    matcher_nodes::VTuple(GeneralTerm)


    node_sym_IDs::VTuple(Int8)

    node_parent_IDs::VTuple(Int8)

    node_tail_length::VTuple(Int8) # if 0, then the end of local subterms
    syms::VTuple(Symbol)
    sym_constraints::VTuple(Function)
end

mutable struct MatchingInfo
    matched_parent_nodes::Vector{SymbolicTerm}
    matched_sym_nodes::Vector{Union{Symbol, GroundTerm, Vector}}
    current_branch::FEM_Int
    branching_infos::Array{FEM_Int, 2} # node ID, start pos of target, size, 3xn
end

struct RewritingRule
    structure_to_match::GeneralTerm
    structure_to_produce::GeneralTerm

    matcher::Matcher
    matchinginfo::MatchingInfo
end

DEFINITION_TABLE = Dict{Symbol, Tuple{Vector{Symbol}, GroundTerm}}()

TensorInfo = Tuple{Symbol, FEM_Int, FEM_Int, FEM_Int} # base_sym, td_order, sd_order
mutable struct PhysicalTensor
    tensor_info::TensorInfo
    definition::GroundTerm
    free_index::Vector{Symbol}

    indexed_instances::Dict{Vector, GroundTerm}
end

mutable struct TensorTable
```

```
    dim::FEM_Int
    tensors::Dict{TensorInfo, PhysicalTensor}
    diff_tensors::Dict{Tuple{TensorInfo, TensorInfo}, TensorInfo}
    TensorTable(dim::Integer) = new(FEM_Int(dim), Dict{TensorInfo, PhysicalTensor}(),
    Dict{Tuple{TensorInfo, TensorInfo}, TensorInfo}())
end

struct Symbolic_BilinearForm #inner product
    dual_word::SymbolicWord
    base_term::GroundTerm
end

function initialize_Definitions!()
    empty!(VARIABLE_ATTRIBUTES)
    empty!(DEFINITION_TABLE)

    (@External_Sym (x, CONTROLPOINT_VAR) (y, CONTROLPOINT_VAR)
    (z, CONTROLPOINT_VAR) (t, GLOBAL_VAR) (dt, GLOBAL_VAR))
    @External_Sym (n, INTEGRATION_POINT_VAR) (δ, SYMMETRIC_TENSOR) ϵ
end
initialize_Definitions!()

visualize(x::Union{Number, Symbol}) = "$x"

const SUBSCRIPT_MAPPING = Dict(:i => "ᵢ", :j => "ⱼ", :k => "ₖ", :l => "ₗ", :m => "ₘ", :n => "ₙ",
:o => "ₒ", :p => "ₚ", :t => "ₜ", 1 => "₁", 2 => "₂", 3 => "₃")
decorate_Subscript(x) = get(SUBSCRIPT_MAPPING, x, "ₓ")
function update_Symbolic_Word_Name(base_variable::Symbol, td_order::Integer, c_ids, sd_ids)
    full_name = "$(base_variable)$(join(decorate_Subscript.(c_ids)))"
    full_name = td_order > 0 ? "$(full_name),$(repeat("ₜ", td_order))" : full_name
    full_name =
    isempty(sd_ids) ? full_name : "$(full_name),$(join(decorate_Subscript.(sd_ids)))"
```

```julia
    return full_name
end


"""
    visualize(x::Union{SymbolicWord, SymbolicTerm, SubtermVariable, FunctionVariable,
    RewritingRule, Symbolic_BilinearForm, Vector{Symbolic_BilinearForm}})

Print the expressions.
"""
visualize(x::SymbolicWord) = update_Symbolic_Word_Name(x.base_variable,
x.td_order, x.c_ids, x.sd_ids)

function update_Term_Name(operation::Symbol, substrings)
    if operation in (:(+), :(-), :(*), :(/), :(^)) && length(substrings) > 1
        head, sep, tail = "(", " $operation ", ")"
    else
        head, sep, tail = "$operation(", ", ", ")"
    end
    body = join(substrings, sep)
    return string(head, body, tail)
end
visualize(x::SymbolicTerm) = update_Term_Name(x.operation, visualize.(x.subterms))


visualize(x::SubtermVariable) = "$(x.sym)"
visualize(x::FunctionVariable) =  update_Term_Name(x.operation, visualize.(x.subterms))
visualize(x::RewritingRule) =
"$(visualize(x.structure_to_match)) => $(visualize(x.structure_to_produce))"


function get_TensorName(x::TensorInfo)
    base_sym, _, td_order, sd_order = x
    full_name = "$base_sym"
    if td_order > 0
```

```
        full_name = "$(full_name)_$(repeat("t", td_order))"
    end
    if sd_order > 0
        full_name = "$(full_name)_$(repeat("x", sd_order))"
    end
    return full_name
end
get_DiffSym(x::TensorInfo, y::TensorInfo) =
Symbol("d($(get_TensorName(x)), $(get_TensorName(y)))")
visualize(x::PhysicalTensor) = "$(get_TensorName(x.tensor_info)) = $(visualize(x.definition))"

visualize(x::Symbolic_BilinearForm) = "($(visualize(x.dual_word)), $(visualize(x.base_term)))"

Base.show(io::IO, x::Union{SymbolicWord, SymbolicTerm, SubtermVariable, FunctionVariable,
RewritingRule, PhysicalTensor, Symbolic_BilinearForm}) = print(io, visualize(x))
Base.hash(x::SymbolicWord, h::UInt) = hash([x.base_variable, x.td_order, x.c_ids, x.sd_ids], h)
Base.:(==)(x::SymbolicWord, y::SymbolicWord) = (x.base_variable == y.base_variable) &&
(x.td_order == y.td_order) && (x.c_ids == y.c_ids) && (x.sd_ids == y.sd_ids)

Base.hash(x::SymbolicTerm, h::UInt) = hash(x.subterms, hash(x.operation, h))
Base.:(==)(x::SymbolicTerm, y::SymbolicTerm) =
(x.operation == y.operation) && (x.subterms == y.subterms)

Base.isless(::Integer, ::Symbol) = true
Base.isless(::Symbol, ::Integer) = false
# Base.isless(::FEM_Int, ::Symbol) = true
# Base.isless(::Symbol, ::FEM_Int) = false
function construct_Word(base_variable::Symbol, c_ids::Vector, d_ids::Vector)
    if isempty(d_ids)
        return SymbolicWord(base_variable, FEM_Int(0), c_ids, IndexSym[])
    else
        total_d_ids_num = length(d_ids)
        filter!(x -> x != :t, d_ids)
        return SymbolicWord(base_variable, total_d_ids_num - length(d_ids), c_ids, sort!(d_ids))
    end
end

function parse_Word_Index(this_word::SymbolicWord)
    @Takeout (c_ids, sd_ids) FROM this_word
    total_ids = [c_ids; sd_ids]
    free_index, dumb_index = Symbol[], Symbol[]
    for arg in total_ids
        arg isa Number && continue
        :t in total_ids && error("t is not allowed in component index, or there
        is a bug and derivative index t is not processed")
        if arg in dumb_index
            error("$arg appears 3 times")
        elseif arg in free_index
            push!(dumb_index, arg)
        else
            push!(free_index, arg)
        end
    end
    setdiff!(free_index, dumb_index)
    return free_index, dumb_index
end

const VOIGT_INDEX_2D = FEM_Int[1 3; 3 2]
const VOIGT_INDEX_3D = FEM_Int[1 6 5; 6 2 4; 5 4 3]
const INVERSE_VOIGT_INDEX_2D = Tuple{FEM_Int, FEM_Int}[(1, 1), (2, 2), (1, 2)]
const INVERSE_VOIGT_INDEX_3D =
Tuple{FEM_Int, FEM_Int}[(1, 1), (2, 2), (3, 3), (2, 3), (1, 3), (1, 2)]
function Voigt_ID(i, j, dim)
    if dim == 2
```

```
        return VOIGT_INDEX_2D[i,j]
    elseif dim == 3
        return VOIGT_INDEX_3D[i,j]
    else
        error("Undefined symmetry")
    end
end
function inverse_Voigt_ID(i, dim)
    if dim == 2
        return INVERSE_VOIGT_INDEX_2D[i]
    elseif dim == 3
        return INVERSE_VOIGT_INDEX_3D[i]
    else
        error("Undefined symmetry")
    end
end

function word_To_Sym(dim::Integer, base_variable::Symbol,
td_order::Integer, c_ids::Vector, sd_ids::Vector)
    full_name ="$base_variable"
    if length(c_ids) == 0
    elseif length(c_ids) == 1
        full_name = "$(full_name)$(c_ids[1])"
    else
        if :SYMMETRIC_TENSOR in get_VarAttribute(base_variable)
            full_name = "$(full_name)$(Voigt_ID(c_ids..., dim))"
        else
            ID = 1 + sum([(c_id - 1) * dim ^ (i - 1) for (i, c_id) in enumerate(c_ids)])
            full_name = "$(full_name)$(ID)"
        end
    end
    if td_order > 0
        full_name = "$(full_name)_$(repeat("t", td_order))"
    end

    if ~(isempty(sd_ids))
        full_name = "$(full_name)_$(sd_ids...)"
    end
    return Symbol(full_name)
end

get_VarAttribute(sym::Symbol) = get(VARIABLE_ATTRIBUTES, sym, Symbol[])
get_VarAttribute(word::SymbolicWord) = get_VarAttribute(word.base_variable)
# word_To_SymType(symbolic_word::SymbolicWord) = symbolic_word.base_variable
word_To_BaseSym(dim::Integer, symbolic_word::SymbolicWord) = word_To_Sym(dim,
symbolic_word.base_variable, FEM_Int(0), symbolic_word.c_ids, IndexSym[])
word_To_LocalSym(dim::Integer, symbolic_word::SymbolicWord) = word_To_Sym(dim,
symbolic_word.base_variable, symbolic_word.td_order, symbolic_word.c_ids, IndexSym[])
word_To_TotalSym(dim::Integer, symbolic_word::SymbolicWord) = word_To_Sym(dim,
symbolic_word.base_variable, symbolic_word.td_order, symbolic_word.c_ids, symbolic_word.sd_ids)

get_FreeIndex(this_num::Union{Number, Symbol}) = Symbol[]
get_FreeIndex(this_word::SymbolicWord) = parse_Word_Index(this_word)[1]
get_FreeIndex(this_term::SymbolicTerm) = copy(this_term.free_index)

get_DumbIndex(this_num::Union{Number, Symbol}) = Symbol[]
get_DumbIndex(this_word::SymbolicWord) = parse_Word_Index(this_word)[2]
get_DumbIndex(this_term::SymbolicTerm) = copy(this_term.dumb_index)

construct_Term(this_num::Number) = FEM_Float(this_num)
construct_Term(arg::Symbol) = SymbolicWord(arg)

function convert_And_Append!(arr, source)
```

```
    for id in source
        if id isa Number
            push!(arr, FEM_Int(id))
        elseif id isa Symbol
            push!(arr, id)
        else
            error("Wrong index")
        end
    end
    return arr
end

function construct_Term(arg::Expr)
    if arg.head == :curly
        sym = arg.args[1]
        length(arg.args) == 1 && error("Wrong grammar: {indices}
        cannot be empty, not beautiful")
        c_ids, d_ids = IndexSym[], IndexSym[]
        if typeof(arg.args[2]) == Expr && arg.args[2].head == :parameters
            convert_And_Append!(d_ids, arg.args[2].args[:])
            convert_And_Append!(c_ids, arg.args[3:end])
        else
            convert_And_Append!(c_ids, arg.args[2:end])
        end
        return construct_Word(sym, c_ids, d_ids)
    elseif arg.head == :call
        operation, arguments = arg.args[1], arg.args[2:end]
        return construct_Term(operation, construct_Term.(arguments))
    else
        error("Unknown grammar")
    end
end
```

```
#Note current no shift
function construct_Term(operation::Symbol, subterms::Vector)
    # shortcuts
    if operation == :+
        nums = filter(x -> x isa Number, subterms)
        num = isempty(nums) ? 0. : sum(nums)
        filter!(x -> ~(x isa Number), subterms)
        isempty(subterms) && return num
        if num != 0
            subterms = [num; subterms]
        else
            (length(subterms) == 1) && return subterms[1]
        end
        subterms = subterms[sortperm(hash.(subterms))]

        free_index = get_FreeIndex(subterms[1])
        for this_subterm in subterms[2:end]
            new_free_index = get_FreeIndex(this_subterm)
            (sort(free_index) != sort(new_free_index)) && error("$(visualize(this_subterm)),
            $(visualize(subterms[1])) should have the same free index but not,
            $new_free_index, $free_index")
        end
        return SymbolicTerm(operation, convert(Vector{Union{Number, Symbol,
        SymbolicWord, SymbolicTerm}}, subterms), free_index, Symbol[])
    elseif operation == :*
        nums = filter(x -> x isa Number, subterms)
        num = isempty(nums) ? 1. : prod(nums)
        filter!(x -> ~(x isa Number), subterms)
        isempty(subterms) && return num
        if num == 0
            return num
        elseif num == 1
```

```
            (length(subterms) == 1) && return subterms[1]
        else
            subterms = [num; subterms]
        end
        subterms = subterms[sortperm(hash.(subterms))]
    elseif operation == :^
        (subterms[1] isa Number) && (subterms[2] isa Number) && return FEM_Float(subterms[1] ^
subterms[2])
        (subterms[2] == 0.) && return FEM_Float(1.)
        (subterms[2] == 1.) && return subterms[1]
        (subterms[1] == 0.) && return FEM_Float(0.)
        (subterms[1] == 1.) && return FEM_Float(1.)
        if isempty(get_FreeIndex(subterms[1])) && isempty(get_FreeIndex(subterms[2]))
            return SymbolicTerm(:^, subterms, Symbol[], Symbol[])
        else
            error("Free index in power: $subterms, is ambiguous")
        end
    elseif operation == :Bilinear
        (length(subterms) == 2) ||
        error("$subterms does not have 2 elements as the subterms of a bilinear term")
        (subterms[1] isa Number) && return FEM_Float(0.)
        (subterms[2] == 0.) && return FEM_Float(0.)
    elseif isempty(subterms)
        return SymbolicTerm(operation, Any[], Symbol[], Symbol[])
    elseif operation == :-
        if length(subterms) == 1
            return mult([FEM_Float(-1), subterms[1]])
        elseif length(subterms) == 2
            return plus([subterms[1], mult([FEM_Float(-1), subterms[2]])])
        else
            error("Minus can at most only have 2 subterms")
        end
    elseif operation == :/
        return mult([subterms[1], construct_Term(:^, [subterms[2], FEM_Float(-1)])])
    end

    free_index = get_FreeIndex(subterms[1])
    dumb_index = Symbol[]
    for this_subterm in subterms[2:end]
        for this_index in get_FreeIndex(this_subterm)
            if this_index in dumb_index
                error("$this_index appears 3 times")
            elseif this_index in free_index
                push!(dumb_index, this_index)
                filter!(x -> x != this_index, free_index)
            else
                push!(free_index, this_index)
            end
        end
    end
    total_indices = Symbol[free_index; dumb_index]
    for i in 1:length(subterms)
        for ID in total_indices
            if ID in get_DumbIndex(subterms[i])
                subterms[i] = _substitute_Term!(subterms[i], ID => gensym())
            end
        end
    end
    return SymbolicTerm(operation, subterms, free_index, dumb_index)
end

refresh_Term(this_term, term_changed::Bool) =
term_changed ? refresh_Term(this_term) : this_term
refresh_Term(this_term) = this_term
refresh_Term(this_term::SymbolicTerm) = construct_Term(this_term.operation, this_term.subterms)


_substitute_Term!(this_number::Number, mapping::Pair) =

this_number #substitute index, may change free/dumb index relations but not term structures

function _substitute_Term!(this_word::SymbolicWord, mapping::Pair)

    source_id, target_id = mapping

    if (source_id in get_FreeIndex(this_word)) &&

    (target_id isa Symbol) && (target_id in get_DumbIndex(this_word))

        this_word = _substitute_Term!(this_word, target_id => gensym())

    end


    if source_id in this_word.c_ids

        replace!(this_word.c_ids, mapping)

        if length(this_word.c_ids) == 2

            (:SYMMETRIC_TENSOR in get_VarAttribute(this_word)) &&

            sort!(this_word.c_ids) #More symmetry, i.e., Voigt ones can be added here

        end

    end # note c_ids and sd_ids are two independent parts


    if source_id in this_word.sd_ids

        sort!(replace!(this_word.sd_ids, mapping))

    end

    this_word

end


function _substitute_Term!(this_term::SymbolicTerm, mapping::Pair)

    source_id, target_id = mapping

    if (source_id in this_term.free_index) || (source_id in this_term.dumb_index)

        if (target_id isa Symbol) && (target_id in this_term.dumb_index)

            this_term = _substitute_Term!(this_term, target_id => gensym())

        end
        for i = 1:length(this_term.subterms)
            this_term.subterms[i] = _substitute_Term!(this_term.subterms[i], mapping)
        end
        return refresh_Term(this_term)
    else
        return this_term
    end
end

function substitute_Term!(this_term, source_ids::Vector, target_ids::Vector)
    mappings = Pair[]
    for (src_id, target_id) in zip(source_ids, target_ids)
        (src_id == target_id) && continue
        if target_id isa Number
            this_term = _substitute_Term!(this_term, src_id => target_id)
        else
            placeholder = gensym()
            this_term = _substitute_Term!(this_term, src_id => placeholder)
            push!(mappings, placeholder => target_id)
        end
    end

    for mapping in mappings
        this_term = _substitute_Term!(this_term, mapping)
    end
    this_term
end

const DEFALUT_INDEX_POOL = [:i, :j, :k, :l, :m, :n, :o, :p, :q, :r, :s, :t]
generate_Index(n::Int) = n < length(DEFALUT_INDEX_POOL) ? DEFALUT_INDEX_POOL[1:n] :
[DEFALUT_INDEX_POOL[1:n]; [Symbol("i$j") for j in 1:(length(DEFALUT_INDEX_POOL) - n)]]
function reindex_Term!(this_term, source_ids::Vector)
```

```
    target_ids = generate_Index(length(source_ids))
    return target_ids, substitute_Term!(this_term, source_ids, target_ids)
end


plus(subterms::Vector) = construct_Term(:+, subterms)
mult(subterms::Vector) = construct_Term(:*, subterms)
#Note this always returns a new single term
unroll_Dumb_Indices(this_number::Number, dim::Integer) = this_number
function unroll_Dumb_Indices(this_word::SymbolicWord, dim::Integer)
    dumb_index = get_DumbIndex(this_word)
    isempty(dumb_index) && return this_word

    total_iterator = Iterators.product([1:dim for i in dumb_index]...)
    result_subterms = [deepcopy(this_word) for i in total_iterator]
    for ids in total_iterator
        result_subterms[ids...] = substitute_Term!(
            result_subterms[ids...], dumb_index, [FEM_Int(id) for id in ids])
    end
    return plus(vec(result_subterms))
end
function unroll_Dumb_Indices(this_term::SymbolicTerm, dim::Integer)
    @Takeout (subterms, dumb_index) FROM this_term
    this_term.subterms .= unroll_Dumb_Indices.(subterms, dim)
    isempty(dumb_index) && return this_term
    total_iterator = Iterators.product([1:dim for i in dumb_index]...)
    result_subterms = [deepcopy(this_term) for i in total_iterator]
    for ids in total_iterator
        result_subterms[ids...] = substitute_Term!(
            result_subterms[ids...], dumb_index, [FEM_Int(id) for id in ids])
    end
    return plus(vec(result_subterms))
end
```

```julia
parse_GeneralTerm(arg::Number) = FEM_Float(arg)

parse_GeneralTerm(arg::GroundTerm) = arg

parse_GeneralTerm(arg::Symbol) = SubtermVariable(arg, IDPDT_SINGLE())

function parse_GeneralTerm(arg::Expr)

    if arg.head == :call

        if arg.args[1] isa Symbol

            operation = arg.args[1]

            tag = FIXED_OP()

        elseif (arg.args[1] isa Expr) && (arg.args[1].head == :braces)

            operation = arg.args[1].args[1]

            tag = IDPDT_OP()

        else

            error("Wrong rewriting rule syntax")

        end

        subterms = parse_GeneralTerm.(arg.args[2:end])

        return FunctionVariable(operation, tag, subterms)

    elseif arg.head == :...

        return SubtermVariable(arg.args[1], IDPDT_FREE())

    elseif arg.head == :vect

        return parse_GeneralTerm(construct_Term(arg.args[1]))

    else

        error("Wrong rewriting rule syntax")

    end

end


get_Sym(this_node::FunctionVariable) = this_node.operation

get_Sym(this_node::SubtermVariable) = this_node.sym


get_Tag(this_node::GroundTerm) = FIXED_SINGLE()

get_Tag(this_node::Union{FunctionVariable, SubtermVariable}) = this_node.tag
```

```
function assemble_Matcher(head_node::GeneralTerm)
    # build matcher structure
    matcher_nodes, node_parent_IDs, node_tail_length, match_batches =
    GeneralTerm[head_node], Int8[0], Int8[0], UnitRange[]
    build_MatcherStructure!(head_node, 1, matcher_nodes,
    node_parent_IDs, node_tail_length, match_batches)
    # collect the symbols and retag
    node_sym_IDs = zeros(Int8, length(matcher_nodes))
    syms, sym_constraints = Symbol[], Function[]
    collect_MatcherSyms!(head_node, 1, node_sym_IDs, syms, sym_constraints)

    for subterm_IDs in match_batches
        for this_subterm_ID in subterm_IDs
            collect_MatcherSyms!(matcher_nodes[this_subterm_ID],
            this_subterm_ID, node_sym_IDs, syms, sym_constraints)
        end
        batch_tags = get_Tag.(matcher_nodes[subterm_IDs])

        inferrable_independent_node_id = findlast(batch_tags .== IDPDT_FREE())
        if ~isnothing(inferrable_independent_node_id)
            infer_ID = subterm_IDs[inferrable_independent_node_id]
            matcher_nodes[infer_ID].tag = IDPDT_INFER()
        end
    end
    return Matcher(VTuple(GeneralTerm)(matcher_nodes), VTuple(Int8)(node_sym_IDs),
    VTuple(Int8)(node_parent_IDs), VTuple(Int8)(node_tail_length),
    VTuple(Symbol)(syms), VTuple(Function)(sym_constraints))
end

build_MatcherStructure!(args...) = nothing
function build_MatcherStructure!(this_node::FunctionVariable, node_ID::Integer,
matcher_nodes::Vector{GeneralTerm}, node_parent_IDs::Vector{Int8},
node_tail_length::Vector{Int8}, match_batches::Vector{UnitRange})
    subterm_length = length(this_node.subterms)
    subterm_range = (1:subterm_length) .+ length(matcher_nodes)
    push!(match_batches, subterm_range)
    append!(matcher_nodes, this_node.subterms)
    append!(node_parent_IDs, [node_ID for i = 1:subterm_length])
    append!(node_tail_length, [subterm_length - i for i = 1:subterm_length])
    for this_subterm_ID in subterm_range #sequential
        build_MatcherStructure!(matcher_nodes[this_subterm_ID], this_subterm_ID,
        matcher_nodes, node_parent_IDs, node_tail_length, match_batches)
    end
    return nothing
end

collect_MatcherSyms!(args...) = nothing
function collect_MatcherSyms!(this_node::Union{FunctionVariable,
SubtermVariable}, node_ID::Integer,
node_sym_IDs::Vector{Int8}, syms::Vector{Symbol}, sym_constraints::Vector{Function})
    this_node.tag isa FIXED_OP && return nothing
    is_independent, sym_ID = check_Sym_Independent!(get_Sym(this_node), syms, sym_constraints)
    is_independent || retag_Dependent!(this_node.tag, this_node)
    node_sym_IDs[node_ID] = sym_ID
    nothing
end

function check_Sym_Independent!(this_sym::Symbol,
syms::Vector{Symbol}, sym_constraints::Vector{Function})
    ID = findfirst(x -> x == this_sym, syms)
    if isnothing(ID) # new (independent) sym
        push!(syms, this_sym)
        push!(sym_constraints, get(SEMANTIC_CONSTRAINT, this_sym, always_True))
        return true, length(syms)
    else  # old (dependent) sym
        return false, ID
    end
end
retag_Dependent!(args...) = nothing
retag_Dependent!(::IDPDT_OP, this_node) = (this_node.tag = DPDT_OP()); nothing
retag_Dependent!(::IDPDT_SINGLE, this_node) = (this_node.tag = DPDT_SINGLE()); nothing
retag_Dependent!(::IDPDT_FREE, this_node) = (this_node.tag = DPDT_FREE()); nothing

function allocate_MatchingInfo(matcher::Matcher)
    @Takeout (matcher_nodes, syms) FROM matcher
    matched_parent_nodes = Vector{SymbolicTerm}(undef, length(matcher_nodes))
    matched_sym_nodes = Vector{Union{Symbol, GroundTerm, Vector}}(undef, length(syms))
    current_branch = 1
    idp_free_node_num = sum(get_Tag.(matcher_nodes) .== IDPDT_FREE())
    branching_infos = zeros(FEM_Int, 3, idp_free_node_num)
    return @Construct MatchingInfo
end

function define_Rewriting_Structure(this_definition::Expr)
    if this_definition.head == :call && this_definition.args[1] == :(=>)
        structure_to_match = parse_GeneralTerm(this_definition.args[2])
        structure_to_produce = parse_GeneralTerm(this_definition.args[3])
        matcher = assemble_Matcher(structure_to_match)
        matchinginfo = allocate_MatchingInfo(matcher)
        return @Construct RewritingRule
    else
        error("Parse error, $this_definition")
    end
end

macro Define_Rewrite_Rule(input_ex::Expr)
```

```
    @assert input_ex.head == :=
    lhs, rhs = input_ex.args[1], input_ex.args[2]
    return esc(:($lhs = define_Rewriting_Structure($(Meta.quot(rhs)))))
end

always_True(x) = true
SEMANTIC_CONSTRAINT = Dict{Symbol, Function}()
AUX_SYM_DEFINITION = Dict{Symbol, Tuple{Function, Tuple{Vararg{Symbol}}}}()
macro Define_Semantic_Constraint(input_ex::Expr)
    if input_ex.head == :call && input_ex.args[1] == :(∈)
        lhs, rhs = input_ex.args[2:3]
        (lhs isa Symbol) || error("Wrong syntax")
        func_name = gensym()

        output_ex = :(begin
            $func_name($lhs) = $rhs
            SEMANTIC_CONSTRAINT[$(Meta.quot(lhs))] = $func_name
        end)
        return esc(output_ex)
    else
        error("Wrong syntax")
    end
end

macro Define_Aux_Semantics(input_ex::Expr)
    if input_ex.head == :(=)
        lhs, _ = input_ex.args
        lhs.head == :call || error("Wrong syntax")
        aux_symbol, syntactic_symbols = lhs.args[1], lhs.args[2:end]

        func_name = gensym()
        define_code = input_ex
        define_code.args[1].args[1] = func_name

        output_ex = :(begin
            $define_code
            AUX_SYM_DEFINITION[$(Meta.quot(aux_symbol))] =
            ($func_name, tuple($((Meta.quot.(syntactic_symbols))...)))
        end)
        return esc(output_ex)
    else
        error("Wrong syntax")
    end
end

function match_Global(source_term, matcher::Matcher, matchinginfo::MatchingInfo)
    matchinginfo.current_branch = 1
    node_ID, last_host_subterm_id = 1, 0
    subterms = [source_term]
    is_success = true
    matcher_size = length(matcher.matcher_nodes)
    while true
        if is_success
            (is_success, matched_size) = match_Local!(matcher.matcher_nodes[node_ID],
            node_ID, last_host_subterm_id, subterms, matcher, matchinginfo)
        else
            target_branch = (matchinginfo.current_branch -= 1)
            (target_branch == 0) && return false # out of stacks, fail
            node_ID, last_host_subterm_id, _ = matchinginfo.branching_infos[:, target_branch]

            subterms = reset_subterms(node_ID, matcher, matchinginfo)
            (is_success, matched_size) = match_Subterm!(IDPDT_FREE(), node_ID,
            last_host_subterm_id, subterms, matcher, matchinginfo, false)
        end
        is_success || continue

        if matcher.node_tail_length[node_ID] == 0 # is node tail, next should switch
            if (matched_size + last_host_subterm_id) == length(subterms)
                (node_ID == matcher_size) && return true #success

                last_host_subterm_id = 0
                subterms = reset_subterms(node_ID += 1, matcher, matchinginfo)
            else
                is_success = false
            end
        else
            node_ID += 1
            last_host_subterm_id += matched_size
        end
    end
end
get_matched_node(node_ID::Integer, matcher::Matcher, matchinginfo::MatchingInfo) =
matchinginfo.matched_sym_nodes[matcher.node_sym_IDs[node_ID]]
reset_subterms(node_ID::Integer, matcher::Matcher, matchinginfo::MatchingInfo) =
matchinginfo.matched_parent_nodes[matcher.node_parent_IDs[node_ID]].subterms

function match_Local!(this_node::GroundTerm, node_ID::Integer,
last_host_subterm_id::Integer, subterms::Vector, matcher::Matcher,
matchinginfo::MatchingInfo)
    (last_host_subterm_id == length(subterms)) && return (false, 1)
    return (this_node == subterms[last_host_subterm_id + 1]), 1
end
function match_Local!(this_node::FunctionVariable, node_ID::Integer,
last_host_subterm_id::Integer, subterms::Vector, matcher::Matcher, matchinginfo::MatchingInfo)
    (last_host_subterm_id == length(subterms)) && return (false, 1)
    target = subterms[last_host_subterm_id + 1]
    (target isa SymbolicTerm) || return (false, 1)
    matchinginfo.matched_parent_nodes[node_ID] = target
    return match_Function!(this_node.tag, this_node.operation, node_ID,
    target.operation, matcher, matchinginfo), 1
end
match_Local!(this_node::SubtermVariable, node_ID::Integer,
last_host_subterm_id::Integer, subterms::Vector, matcher::Matcher, matchinginfo::MatchingInfo) =
match_Subterm!(this_node.tag, node_ID, last_host_subterm_id, subterms, matcher, matchinginfo)

match_Function!(::FIXED_OP, node_op::Symbol, node_ID::Integer, target_op::Symbol,
matcher::Matcher, matchinginfo::MatchingInfo) = node_op == target_op
function match_Function!(::IDPDT_OP, node_op::Symbol, node_ID::Integer,
target_op::Symbol, matcher::Matcher, matchinginfo::MatchingInfo)
    matchinginfo.matched_sym_nodes[matcher.node_sym_IDs[node_ID]] = target_op
    return checker(target_op)
end
match_Function!(::DPDT_OP, node_op::Symbol, node_ID::Integer, target_op::Symbol,
checker::Function, matchinginfo::MatchingInfo) =
get_matched_node(node_ID, matcher, matchinginfo) == target_op

function match_Subterm!(::IDPDT_SINGLE, node_ID::Integer,
last_host_subterm_id::Integer, subterms::Vector, matcher::Matcher, matchinginfo::MatchingInfo)
    (last_host_subterm_id == length(subterms)) && return (false, 1)
    sym_ID = matcher.node_sym_IDs[node_ID]
    matchinginfo.matched_sym_nodes[sym_ID] = subterms[last_host_subterm_id + 1]
    checker = matcher.sym_constraints[sym_ID]
    return checker(subterms[last_host_subterm_id + 1]), 1
end

function match_Subterm!(::DPDT_SINGLE, node_ID::Integer, last_host_subterm_id::Integer,
subterms::Vector, matcher::Matcher, matchinginfo::MatchingInfo)
```

```
    (last_host_subterm_id == length(subterms)) && return (false, 1)
    return (get_matched_node(node_ID, matcher, matchinginfo) ==
    subterms[last_host_subterm_id + 1]), 1
end

function match_Subterm!(::DPDT_FREE, node_ID::Integer, last_host_subterm_id::Integer,
subterms::Vector, matcher::Matcher, matchinginfo::MatchingInfo)
    prev_matched_nodes = get_matched_node(node_ID, matcher, matchinginfo)
    prev_length = length(prev_matched_nodes)

    ((last_host_subterm_id + prev_length) <= length(subterms)) || return (false, prev_length)
    return (prev_matched_nodes ==
    subterms[last_host_subterm_id .+ (1:prev_length)]), prev_length
end

function match_Subterm!(::IDPDT_FREE, node_ID::Integer, last_host_subterm_id::Integer,
subterms::Vector, matcher::Matcher, matchinginfo::MatchingInfo, is_first_entry::Bool = true)
    @Takeout (current_branch, branching_infos) FROM matchinginfo

    if is_first_entry
        branching_infos[1, current_branch] = node_ID
        branching_infos[2, current_branch] == last_host_subterm_id
        new_length = (branching_infos[3, current_branch] = 0)
    else
        new_length = (branching_infos[3, current_branch] += 1)
    end

    sym_ID = matcher.node_sym_IDs[node_ID]
    occupied_space, infer_num = tail_Space(node_ID, matcher.node_tail_length[node_ID],
    sym_ID, matcher, matchinginfo)
    rest_target_length = length(subterms) - last_host_subterm_id - occupied_space
    if (new_length * infer_num) <= rest_target_length
        matchinginfo.matched_sym_nodes[sym_ID] =
        subterms[last_host_subterm_id .+ (1:new_length)]
        matchinginfo.current_branch += 1
        return true, new_length # no practical need to check free term
    else
        return false, 0
    end
end

function match_Subterm!(::IDPDT_INFER, node_ID::Integer, last_host_subterm_id::Integer,
subterms::Vector, matcher::Matcher, matchinginfo::MatchingInfo)
    sym_ID = matcher.node_sym_IDs[node_ID]
    occupied_space, infer_num = tail_Space(node_ID, matcher.node_tail_length[node_ID],
    sym_ID, matcher, matchinginfo)
    rest_target_length = length(subterms) - last_host_subterm_id - occupied_space

    inferred_length = rest_target_length / infer_num
    if isinteger(inferred_length)
        matchinginfo.matched_sym_nodes[sym_ID] = subterms[last_host_subterm_id .+
        (1:Int(inferred_length))]
        return true, inferred_length # no practical need to check free term
    else
        return false, 0
    end
end

local_Length(::Union{FIXED_OP, IDPDT_OP, DPDT_OP, FIXED_SINGLE, IDPDT_SINGLE, DPDT_SINGLE},
node_ID::Integer, matcher::Matcher, matchinginfo::MatchingInfo) = 1
local_Length(::DPDT_FREE, node_ID::Integer, matcher::Matcher, matchinginfo::MatchingInfo) =
length(get_matched_node(node_ID, matcher, matchinginfo))
local_Length(::Union{IDPDT_FREE, IDPDT_INFER}, node_ID::Integer, matcher::Matcher,
matchinginfo::MatchingInfo) = 0

function tail_Space(node_ID::Integer, tail_length::Integer, sym_ID::Integer,

matcher::Matcher, matchinginfo::MatchingInfo)

    infer_num = 1

    occupied_space = 0

    for local_node_ID in ((1:tail_length) .+ node_ID)

        if matcher.node_sym_IDs[local_node_ID] == sym_ID

            infer_num += 1

        else

            occupied_space += local_Length(get_Tag(matcher.matcher_nodes[local_node_ID]),

            local_node_ID, matcher, matchinginfo)

        end

    end

    return occupied_space, infer_num

end


embody_GeneralTerm(this_node::GroundTerm, this_match::Dict) = deepcopy(this_node)

function embody_GeneralTerm(this_node::SubtermVariable, this_match::Dict)

    if this_node.sym in keys(this_match)

        return deepcopy(this_match[this_node.sym])

    elseif this_node.sym in keys(AUX_SYM_DEFINITION)

        this_func, syntactic_symbols = AUX_SYM_DEFINITION[this_node.sym]

        semantic_symbols = [this_match[sym] for sym in syntactic_symbols]

        return this_func(semantic_symbols...)

    else

        error("Wrong syntax")

    end

end

function embody_GeneralTerm(this_node::FunctionVariable, this_match::Dict)

    semantic_subterms = GroundTerm[]

    for this_syntactic_subnode in this_node.subterms

        this_semantic_subterm = embody_GeneralTerm(this_syntactic_subnode, this_match)
        if this_semantic_subterm isa GroundTerm
            push!(semantic_subterms, this_semantic_subterm)
        else
            append!(semantic_subterms, this_semantic_subterm)
        end
    end

    syntax_operation = this_node.operation
    if this_node.tag isa FIXED_OP
        semantic_operation = syntax_operation
    else
        if syntax_operation in keys(this_match)
            semantic_operation = this_match[syntax_operation]
        elseif syntax_operation in keys(AUX_SYM_DEFINITION)
            this_func, syntactic_symbols = AUX_SYM_DEFINITION[syntax_operation]
            semantic_operation = [this_match[sym] for sym in syntactic_symbols]
        end
    end
    return construct_Term(semantic_operation, semantic_subterms)
end

function apply_Rules_One_Node(source_term::GroundTerm, this_rule::RewritingRule)
    @Takeout (matcher, matchinginfo, structure_to_produce) FROM this_rule
    is_match = match_Global(source_term, matcher, matchinginfo)
    return is_match ? (true, embody_GeneralTerm(structure_to_produce,
    Dict(matcher.syms .=> matchinginfo.matched_sym_nodes))) : (false, source_term)
end
function apply_Rules_One_Node(source_term::GroundTerm, rules::Vector{RewritingRule})
    term_changed, this_term = false, source_term
    for this_rule in rules
        this_check, this_term = apply_Rules_One_Node(this_term, this_rule)
        term_changed |= this_check
    end
    return term_changed, this_term
end

apply_Rules(source_term::Union{FEM_Float, SymbolicWord}, rules) =
apply_Rules_One_Node(source_term, rules)

function apply_Rules(source_term::SymbolicTerm, rules)
    local_changed, this_term = true, source_term

    head_changed = false
    while local_changed
        local_changed, this_term = apply_Rules_One_Node(this_term, rules)
        head_changed |= local_changed
    end

    subterm_changed = false
    if this_term isa SymbolicTerm
        for i = 1:length(this_term.subterms)
            local_changed, this_term.subterms[i] = apply_Rules(this_term.subterms[i], rules)
            subterm_changed |= local_changed
        end
    end
    return (head_changed || subterm_changed), refresh_Term(this_term, subterm_changed)
end
@Define_Semantic_Constraint F₁ ∈ (isempty(get_FreeIndex(F₁)))

@Define_Rewrite_Rule Add_Splat ≔ ((a...) + (+(b...)) + (c...) => a + b + c)
@Define_Rewrite_Rule Mul_Splat ≔ ((a...) * (*(b...)) * (c...) => a * b * c)
@Define_Rewrite_Rule Pow_Splat ≔ ((a ^ b) ^ c => a ^ (b * c))

@Define_Rewrite_Rule Distributive_MP ≔ ((F₁ * (b...)) ^ c => (a ^ c) * (*(b)) ^ c)
```

```
@Define_Rewrite_Rule Distributive_AM ≔ ((a...) * (b + (c...)) * (d...) =>
a * b * d + a * (+(c)) * d)

expand_And_Flatten(source_term::Union{Number, SymbolicWord}) = source_term
function expand_And_Flatten(source_term::SymbolicTerm) # regulate the term, before merging
    is_changed, this_term = true, source_term
    while is_changed
        is_changed, this_term = apply_Rules(this_term, [Distributive_MP, Distributive_AM])
        is_changed_1, this_term = apply_Rules(this_term, [Add_Splat, Mul_Splat, Pow_Splat])
        is_changed |= is_changed_1
    end
    return this_term
end

check_Merge(src::Union{Number, SymbolicWord}) = src
function check_Merge(source_term::SymbolicTerm)
    new_subterms = check_Merge.(source_term.subterms)
    if source_term.operation == :+
        classifier = Dict{GroundTerm, FEM_Float}()
        for subterm in new_subterms
            if subterm isa Number
                main_term = 1.
                factor = subterm
            elseif (subterm isa SymbolicTerm) && (subterm.operation == :*) &&
            (subterm.subterms[1] isa Number)
                main_term = mult(subterm.subterms[2:end])
                factor = subterm.subterms[1]
            else
                main_term = subterm
                factor = 1.
            end
            classifier[main_term] = get(classifier, main_term, 0.) + factor
        end
        res_subterms = GroundTerm[]
        for (main_term, factor) in classifier
            if main_term isa SymbolicTerm && main_term.operation == :*
                push!(res_subterms, mult([factor; main_term.subterms]))
            else
                push!(res_subterms, mult([factor; main_term]))
            end
        end
        return plus(res_subterms)
    elseif source_term.operation == :*
        classifier = Dict{GroundTerm, GroundTerm}()

        has_free_idx = map(x -> isempty(get_FreeIndex(x)), new_subterms)
        preserved_terms = new_subterms[.~(has_free_idx)]
        processing_terms = new_subterms[has_free_idx]

        for subterm in processing_terms
            if subterm isa Number
                main_term = subterm
                factor = 1.
            elseif (subterm isa SymbolicTerm) && (subterm.operation == :^)
                main_term, factor = subterm.subterms
            else
                main_term = subterm
                factor = 1.
            end
            classifier[main_term] = plus([get(classifier, main_term, 0.), factor])
        end
        processed_terms = [construct_Term(:^, [main_term; check_Merge(factor)])
        for (main_term, factor) in classifier]
        return mult([processed_terms; preserved_terms])
    else
        return construct_Term(source_term.operation, new_subterms)
    end
end
simplify_Common(source_term) = check_Merge(expand_And_Flatten(source_term))


replace_SpecialTerm!(x) = _replace_SpecialTerm!(x) |> simplify_Common
_replace_SpecialTerm!(this_term::Number) = this_term
function _replace_SpecialTerm!(this_term::SymbolicWord)
    @Takeout (base_variable, td_order, c_ids, sd_ids) FROM this_term
    if base_variable == :δ
        if ~((td_order == 0) && isempty(sd_ids))
            return FEM_Float(0.)
        elseif (length(c_ids) == 2) && (c_ids[1] isa Number) && (c_ids[2] isa Number)
            return (c_ids[1] == c_ids[2]) ? FEM_Float(1.) : FEM_Float(0.)
        end
    elseif base_variable == :ϵ
        if length(c_ids) == 3
            if ~((td_order == 0) && isempty(sd_ids))
                return FEM_Float(0.)
            elseif (c_ids[1] isa Number) && (c_ids[2] isa Number) && (c_ids[3] isa Number)
                if (c_ids == [1, 2, 3]) || (c_ids == [2, 3, 1]) || (c_ids == [3, 1, 2])
                    return 1.
                elseif (c_ids == [1, 3, 2]) || (c_ids == [3, 2, 1]) || (c_ids == [2, 1, 3])
                    return -1.
                else
                    return 0.
                end
            end
        end
    end
    return this_term
```

```
end
function _replace_SpecialTerm!(this_term::SymbolicTerm)
    for i = 1:length(this_term.subterms)
        this_term.subterms[i] = _replace_SpecialTerm!(this_term.subterms[i])
    end
    return this_term
end

base_TensorInfo(sym::Symbol, order::Integer) = (sym, order, 0, 0) |> TensorInfo
word_To_TensorInfo(x::SymbolicWord) = (x.base_variable, length(x.c_ids),
x.td_order, length(x.sd_ids)) |> TensorInfo
tensorinfo_To_Word(x::TensorInfo, ids::Vector{IndexSym}) =
SymbolicWord(x[1], x[3], ids[1:x[2]], ids[(x[2]+ 1):x[4]])
_tensor_DOF(x::TensorInfo) = x[2] + x[4]

unroll_And_Simplify(source_term::GroundTerm, dim::Integer) =
unroll_Dumb_Indices(source_term, dim) |> replace_SpecialTerm!
function construct_Tensor(tensor_info::TensorInfo,
declared_free_index::Vector{Symbol}, definition::GroundTerm)
    src_free_index = get_FreeIndex(definition)
    isempty(symdiff(declared_free_index, src_free_index)) ||
    error("Free indices must match for readability,
    but not in $declared_free_index vs $src_free_index in $definition")
    free_index, definition = reindex_Term!(definition, declared_free_index)
    indexed_instances = Dict{Vector, GroundTerm}()
    println("Building $(_tensor_DOF(tensor_info) == 0 ? :Scalar : :Tensor) $tensor_info")
    println(definition)
    if length(free_index) == 2
        swapped_def = substitute_Term!(deepcopy(definition), free_index, reverse(free_index))
        if definition == swapped_def
            println("$(tensor_info[1]) is a symmetric tensor")
            push!(get!(VARIABLE_ATTRIBUTES, tensor_info[1], Symbol[]), :SYMMETRIC_TENSOR)
```

```julia
        end
    end
    return PhysicalTensor(tensor_info, definition, free_index, indexed_instances)
end

function get_Tensor(tb::TensorTable, tensor_info::TensorInfo)
    if tensor_info in keys(tb.tensors)
        return tb.tensors[tensor_info]
    else
        return tb.tensors[tensor_info] = build_Tensor(tb, tensor_info)
    end
end

function build_Tensor(tb::TensorTable, tensor_info::TensorInfo)
    sym, base_order, td_order, sd_order = tensor_info
    if td_order > 0
        base_tensor = get_Tensor(tb, (sym, base_order, td_order - 1, sd_order) |> TensorInfo)
        target_def = diff_Time(base_tensor.definition) # time derivative doesn't need args
        target_ids = base_tensor.free_index
    elseif sd_order > 0
        base_tensor = get_Tensor(tb, (sym, base_order, td_order, sd_order - 1) |> TensorInfo)
        placeholder = gensym()
        target_def = diff_Space(base_tensor.definition, placeholder)
        target_ids, target_def = reindex_Term!(target_def, [base_tensor.free_index; placeholder])
    else
        target_ids, raw_def = deepcopy(DEFINITION_TABLE[sym])
        target_def = inline_TensorDiff!(tb, unroll_And_Simplify(raw_def, tb.dim))
    end
    return construct_Tensor(tensor_info, target_ids, target_def)
end

collect_ids(this_word::SymbolicWord) = IndexSym[this_word.c_ids; this_word.sd_ids]
```

```julia
function evaluate_Tensor(tb::TensorTable, this_word::SymbolicWord)

    tensor = get_Tensor(tb, word_To_TensorInfo(this_word))

    target_ids = collect_ids(this_word)

    if isempty(target_ids) || target_ids == tensor.free_index

        return tensor.definition

    elseif target_ids in keys(tensor.indexed_instances)

        return tensor.indexed_instances[target_ids]

    else # Here the inline_TensorDiff is to inline the number only

        return tensor.indexed_instances[target_ids] = inline_TensorDiff!(tb,

        replace_SpecialTerm!(substitute_Term!(deepcopy(tensor.definition),

        tensor.free_index, target_ids)))

    end

end


inline_TensorDiff!(tb, x) = _inline_TensorDiff!(tb, x)[2] |> simplify_Common

_inline_TensorDiff!(tb::TensorTable, x::Number) = false, x

function _inline_TensorDiff!(tb::TensorTable, x::SymbolicWord)

    var_attribute = get_VarAttribute(x)

    if (:INTERNAL_VAR in var_attribute) || (:EXTERNAL_VAR in var_attribute)

        return false, x

    else

        local_def = evaluate_Tensor(tb, x)

        if local_def isa Number

            return true, local_def

        else

            return false, x

        end

    end

end


function _inline_TensorDiff!(tb::TensorTable, this_term::SymbolicTerm)

    if this_term.operation == :d
```

```
        ((length(this_term.subterms) == 2) &&
        (this_term.subterms[2] isa SymbolicWord)) || error("AST error")
        return true, diff_Symbol(inline_TensorDiff!(tb, this_term.subterms[1]),
        tb, this_term.subterms[2])
    else
        term_changed, subterms = false, this_term.subterms
        for i = 1:length(subterms)
            subterm_changed, subterms[i] = _inline_TensorDiff!(tb, subterms[i])
            term_changed |= subterm_changed
        end
        return term_changed, refresh_Term(this_term, term_changed)
    end
end

function get_TensorDiff!(tb::TensorTable, src_info::TensorInfo, diff_info::TensorInfo)
    info_pair = (src_info, diff_info)
    if info_pair in keys(tb.diff_tensors)
        return tb.diff_tensors[info_pair]
    else
        return tb.diff_tensors[info_pair] = construct_TensorDiff!(tb, src_info, diff_info)
    end
end

function construct_TensorDiff!(tb::TensorTable, src_info::TensorInfo, diff_info::TensorInfo)
    diff_DOF = _tensor_DOF(diff_info)
    target_info = base_TensorInfo(get_DiffSym(src_info, diff_info),
    _tensor_DOF(src_info) + diff_DOF)

    src_tensor = get_Tensor(tb, src_info)

    diff_ids = IndexSym[gensym() for i = 1:diff_DOF]
    diff_word = tensorinfo_To_Word(diff_info, diff_ids)
    target_ids = Symbol[src_tensor.free_index; diff_ids]
    target_def = diff_Symbol(src_tensor.definition, tb, diff_word)

    println("Building tensor derivative $target_info by $target_def")
    tb.tensors[target_info] = construct_Tensor(target_info, target_ids, target_def)
    return target_info
end

count_Words(x::Number) = 0
count_Words(x::SymbolicWord) = 1
count_Words(x::SymbolicTerm) = sum(count_Words.(x.subterms))

propagate_Symbol(tb, x) = _propagate_Symbol!(tb, x)[2] |> simplify_Common
propagate_Symbol(tb, x::SymbolicTerm) = _propagate_Symbol!(tb,
deepcopy(x))[2] |> simplify_Common
_propagate_Symbol!(tb::TensorTable, x::Number) = false, x
function _propagate_Symbol!(tb::TensorTable, x::SymbolicWord)
    var_attribute = get_VarAttribute(x)
    if (:INTERNAL_VAR in var_attribute) || (:EXTERNAL_VAR in var_attribute)
        return false, x
    else
        local_def = evaluate_Tensor(tb, x)
        if local_def isa SymbolicTerm
            return (count_Words(local_def) > 1) ? (false, x) :
            (true, propagate_Symbol(tb, local_def))
        else
            return true, propagate_Symbol(tb, local_def)
        end
    end
end
function _propagate_Symbol!(tb::TensorTable, this_term::SymbolicTerm)
```

```
    term_changed, subterms = false, this_term.subterms
    for i = 1:length(subterms)
        subterm_changed, subterms[i] = _propagate_Symbol!(tb, subterms[i])
        term_changed |= subterm_changed
    end
    return term_changed, refresh_Term(this_term, term_changed)
end

function generates_All_Related_ITG_Symbols(tb::TensorTable,
this_word::SymbolicWord, attributes::Vector{Symbol})
    dim = tb.dim
    ((this_word.td_order == 0) && isempty(this_word.sd_ids)) ||
    error("Integration point can only have direction IDs")
    tensor_order = length(this_word.c_ids)

    if (tensor_order == 0)
        return Symbol[word_To_TotalSym(tb.dim, this_word)]
    elseif (tensor_order == 1)
        return Symbol[word_To_TotalSym(tb.dim, SymbolicWord(this_word.base_variable,
        0, IndexSym[FEM_Int(i)], Symbol[])) for i = 1:dim]
    elseif (tensor_order == 2)
        if (:SYMMETRIC_TENSOR in get_VarAttribute(this_word.base_variable))
            return Symbol[word_To_TotalSym(tb.dim, SymbolicWord(this_word.base_variable,
            0, IndexSym[inverse_Voigt_ID(i, dim)...], Symbol[])) for i = 1:6]
        else
            return Symbol[word_To_TotalSym(tb.dim, SymbolicWord(this_word.base_variable,
            0, IndexSym[FEM_Int(i), FEM_Int(j)], Symbol[])) for i = 1:dim, j = 1:dim]
        end
    else
        error("Order > 3 integration point variables are explicitly forbidden")
    end
end
```

```
_parse_Term2Expr!(intermediate_code::Vector{Expr}, declared_syms::Set{Symbol},
tb::TensorTable, this_num::FEM_Float) = this_num
function _parse_Term2Expr!(intermediate_code::Vector{Expr},
declared_syms::Set{Symbol}, tb::TensorTable, this_word::SymbolicWord)
    totalsym = word_To_TotalSym(tb.dim, this_word)
    if ~(totalsym in declared_syms)
        attributes = get_VarAttribute(this_word)
        if (:INTERNAL_VAR in attributes) || (:EXTERNAL_VAR in attributes)
            if (:INTEGRATION_POINT_VAR in attributes) &&
            (this_word.base_variable != :n)
                all_syms = generates_All_Related_ITG_Symbols(tb, this_word, attributes)
                sym_tuple = (length(all_syms) == 1) ? all_syms[1] : :(($(all_syms...),))

                _, raw_def = deepcopy(DEFINITION_TABLE[this_word.base_variable])
                this_def = _parse_Term2Expr!(intermediate_code,
                declared_syms, tb, propagate_Symbol(tb, raw_def))

                push!(intermediate_code, :($sym_tuple = $this_def))
                union!(declared_syms, all_syms)
            else
                # println("$(this_word.base_variable) is a base_variable")
                push!(declared_syms, totalsym)
            end
        else
            defs = parse_Term2Expr!(intermediate_code,
            declared_syms, tb, evaluate_Tensor(tb, this_word))
            if defs[1] isa Expr
                push!(intermediate_code, :($totalsym = @. $(defs[1])))
                for this_def in defs[2:end]
                    push!(intermediate_code, :(@. $totalsym += $this_def))
                end
            else
                push!(intermediate_code, :($totalsym = $this_def))
            end
            push!(declared_syms, totalsym)
        end
    end
    return totalsym
end

function _parse_Term2Expr!(intermediate_code::Vector{Expr},
declared_syms::Set{Symbol}, tb::TensorTable, this_term::SymbolicTerm)
    op = this_term.operation
    args = [_parse_Term2Expr!(intermediate_code, declared_syms,
    tb, subterm) for subterm in this_term.subterms]
    if isoperator(op)
        return Expr(:call, op, args...)
    else
        return :(Main.$op($(args...)))
    end
end

function parse_Term2Expr!(intermediate_code::Vector{Expr},
declared_syms::Set{Symbol}, tb::TensorTable, this_term)
    inlined_term = propagate_Symbol(tb, this_term)

    if inlined_term isa SymbolicTerm && inlined_term.operation == :+
        word_nums = count_Words.(inlined_term.subterms)
        limit, counter = 64, 0
        buffer, defs = GroundTerm[], Expr[]
        for (id, word_num) in enumerate(word_nums)
            if (counter += word_num) > limit
                counter = 0
                push!(defs, _parse_Term2Expr!(intermediate_code,
                declared_syms, tb, plus(buffer)))
                empty!(buffer)
            end
            push!(buffer, inlined_term.subterms[id])
        end
        return push!(defs, _parse_Term2Expr!(intermediate_code, declared_syms, tb, p(buffer)))
    else
        return [_parse_Term2Expr!(intermediate_code, declared_syms, tb, inlined_term)]
    end
end

@Define_Semantic_Constraint N₁ ∈ (N₁ isa Number)

@Define_Rewrite_Rule Num_Diff ≔ ∂(N₁) => 0

@Define_Rewrite_Rule Add_Diff ≔ ∂(a + (b...)) => ∂(a) + ∂(+(b...))
@Define_Rewrite_Rule Mul_Diff ≔ ∂(a * (b...)) => ∂(a) * (b...) + a * lus, ∂(c))

DIFF_RULES = [Add_Diff, Mul_Diff, Pow_Diff, Log_Diff, Cond_Diff, Num_Diff]

function _diff_Core!(source_term)
    this_term, is_changed = construct_Term(:∂, [source_term]), true
    while is_changed
        is_changed, this_term = apply_Rules(this_term, DIFF_RULES)
    end
    return this_term
end

for (tag, args) in zip([:Time, :Space, :Variation, :Symbol], [[],
[:(d_id::Symbol)], [:(tb::TensorTable)], [:(tb::TensorTable), :(diff_word::SymbolicWord)]])
    diff_funcname = Symbol("diff_$(tag)")
    eval_global_funcname = Symbol("diff_Eval_$(tag)_Global!")

    eval_local_funcname = Symbol("diff_Eval_$(tag)_Local!")

    @eval begin

        $diff_funcname(x::Number, $(args...)) = 0.

        $diff_funcname(x::Union{SymbolicWord, SymbolicTerm}, $(args...)) =

        $eval_global_funcname(_diff_Core!(deepcopy(x)), $(args...))[2] |> simplify_Common

        $eval_global_funcname(x::Union{Number, SymbolicWord}, $(args...)) = false, x

        function $(eval_global_funcname)(this_term::SymbolicTerm, $(args...))

            if this_term.operation == :∂

                ((length(this_term.subterms) == 1) && (this_term.subterms[1] isa

                SymbolicWord)) || error("AST error")

                return true, $eval_local_funcname(this_term.subterms[1],

                get_VarAttribute(this_term.subterms[1]), $(args...))

            else

                term_changed, subterms = false, this_term.subterms

                for i = 1:length(subterms)

                    subterm_changed, subterms[i] = $eval_global_funcname(

                        subterms[i], $(args...))

                    term_changed |= subterm_changed

                end

                return term_changed, refresh_Term(this_term, term_changed)

            end

        end

    end

end


function diff_Eval_Time_Local!(this_word::SymbolicWord, src_attributes::Vector{Symbol})

    if (:EXTERNAL_VAR in src_attributes)

        return 0

    else

        this_word.td_order += 1

        return this_word
    end
end


function diff_Eval_Space_Local!(this_word::SymbolicWord, src_attributes::Vector{Symbol},
d_id::Symbol)
    if (:EXTERNAL_VAR in src_attributes) && (~(:CONTROLPOINT_VAR in src_attributes))
        return 0.
    else
        this_word.sd_ids = sort!([this_word.sd_ids; d_id])
        return this_word
    end
end


function diff_Eval_Variation_Local!(this_word::SymbolicWord, src_attributes::Vector{Symbol},
tb::TensorTable)
    if (:INTERNAL_VAR in src_attributes)
        return construct_Term(:δ, [this_word])
    elseif (:EXTERNAL_VAR in src_attributes)
        return 0.
    else
        return diff_Variation(evaluate_Tensor(tb, this_word), tb)
    end
end


is_variation(x) = false
is_variation(x::SymbolicTerm) = x.operation == :δ
collect_Variations(x::GroundTerm, tb) = collect_Variations(Dict{SymbolicWord,
Vector{GroundTerm}}(), x, tb)
collect_Variations(buffer::Dict{SymbolicWord, Vector{GroundTerm}},
x::GroundTerm, tb) = _collect_Variations(buffer, diff_Variation(x, tb))
_collect_Variations(buffer::Dict{SymbolicWord, Vector{GroundTerm}},
this_term::Union{Number, SymbolicWord}) = buffer
```

```
function _collect_Variations(buffer::Dict{SymbolicWord,
Vector{GroundTerm}}, this_term::SymbolicTerm)
    term_op = this_term.operation
    if term_op == :δ
        push!(get!(buffer, this_term.subterms[1], GroundTerm[]), 1.)
    elseif term_op == :+
        for subterm in this_term.subterms
            _collect_Variations(buffer, subterm)
        end
    elseif term_op == :*
        is_var = is_variation.(this_term.subterms)
        var_IDs = findall(is_var)
        length(var_IDs) != 1 && error("One LHS should contain and only
        contain one variation but not in $(this_term),
        it is an internal error that shouldn't appear")
        push!(get!(buffer, this_term.subterms[var_IDs[1]].subterms[1],
        GroundTerm[]), mult(this_term.subterms[.~is_var]))
    else
        error("Wrong IR, internal bug")
    end
    return buffer
end


_delta_Func(c1::Number, c2::Number) = c1 == c2 ? 1. : 0.
_delta_Func(c1, c2) = construct_Word(:δ, [c1, c2], IndexSym[])
function diff_Eval_Symbol_Local!(src_word::SymbolicWord, src_attributes::Vector{Symbol},
tb::TensorTable, diff_word::SymbolicWord) # symbolic derivative
    if (src_word.base_variable == diff_word.base_variable) && (src_word.td_order ==
    diff_word.td_order) && (length(src_word.c_ids) == length(diff_word.c_ids)) &&
    (length(src_word.sd_ids) == length(diff_word.sd_ids))
        return mult([[_delta_Func(i1, i2) for (i1, i2) in
        zip(src_word.c_ids, diff_word.c_ids)];
        [_delta_Func(i1, i2) for (i1, i2) in zip(src_word.sd_ids, diff_word.sd_ids)]])
    else
        if (:INTERNAL_VAR in src_attributes) || (:EXTERNAL_VAR in src_attributes)
            return 0.
        else
            this_tensorinfo = get_TensorDiff!(tb, word_To_TensorInfo(src_word),
            word_To_TensorInfo(diff_word))
            return tensorinfo_To_Word(this_tensorinfo,
            IndexSym[collect_ids(src_word); collect_ids(diff_word)])
        end
    end
end

eval_Constant!(x::FEM_Float) = x
function eval_Constant!(x::SymbolicWord)
    if (x.base_variable in keys(VARIABLE_ATTRIBUTES)) ||
    (x.base_variable in keys(DEFINITION_TABLE))
        return x
    else
        local_def = Core.eval(Main, x.base_variable)
        local_def isa Number || error("The \"$local_def\" is not declared before,
        so it can only be a constant number, but it is not.")
        return FEM_Float(local_def)
    end
end
function eval_Constant!(this_term::SymbolicTerm)
    subterms = this_term.subterms
    for i = 1:length(subterms)
        subterms[i] = eval_Constant!(subterms[i])
    end
    return this_term
end
```

```
_print_hline() = println("$(repeat("-", 100))")

function FEM_Define!(base_sym::Symbol, term_ex, declared_free_index::Vector{Symbol} = Symbol[])

    this_term = construct_Term(term_ex) |> eval_Constant! |> simplify_Common

    free_index = get_FreeIndex(this_term)

    attributes = get_VarAttribute(base_sym)

    if (:INTEGRATION_POINT_VAR in attributes)

        isempty(free_index) || error("Integration variables are external variables,

        so the definition must be concrete and has no free index, such as $free_index")

    else

        ((:INTERNAL_VAR in attributes) || (:EXTERNAL_VAR in attributes)) &&

        error("Controlpoint or global variable should not be defined.")

        isempty(symdiff(declared_free_index, free_index)) || error("Free indices

         must match for readability, but not in $declared_free_index vs $free_index")

    end


    _print_hline()

    if isempty(declared_free_index)

        println("Scalar $base_sym is declared as $(visualize(this_term)).")

    else

        println("Tensor $base_sym{$(join(declared_free_index, ", "))}

        is declared as $(visualize(this_term))")

    end

    DEFINITION_TABLE[base_sym] = (declared_free_index, this_term)

    return nothing

end

"""

    @Sym b1, c1, b2, c2


    @Def a1 = f(b1, c1)

    @Def begin

        a2{i} = b2{i} + c2{i}
        a3{i,j} = a2{i} * b2{j}
        ...
    end
`@Def a1 = f(b1, c1)` defines scalar `a` as `f(b, c)` while `a3{i,j} = a2{i} * b2{j}`
defines tensor `a3` being the tensor product of vector `a2`,`b2`.
Note, free indices are required to match.
"""
macro Def(input_ex)
    input_ex_batch = vectorize_Args(input_ex)
    output_ex = Expr(:block)
    for this_ex in input_ex_batch
        lhs, rhs = this_ex.args
        if lhs isa Symbol
            term_name = lhs
            declared_free_index = Symbol[]
        elseif lhs.head == :curly && length(lhs.args) > 1 && lhs.args[2] isa Symbol
            term_name = lhs.args[1]
            declared_free_index = [x for x in lhs.args[2:end]]
            declared_free_index isa Vector{Symbol} || error("Grammar error, please
            declare a tensor by whole using kronecker delta instead of
            directly assigning components. A{1} = 1 is bad, while A{i} = δ{i,1} is good.")
        else
            error("Wrong grammar, lhs = $lhs, rhs = $rhs")
        end
        push!(output_ex.args, :(FEM_Define!($(Meta.quot(term_name)),
        $(Meta.quot(rhs)), $declared_free_index)))
        push!(output_ex.args, :($term_name = SymbolicWord($(Meta.quot(term_name)))))
    end
    return esc(output_ex)
end

function build_WeakForm(tb::TensorTable, src_def)
```

```
    raw_BFs = collect_BilinearTerms!(tb, SymbolicTerm[], src_def isa SymbolicTerm ?
    unroll_And_Simplify(src_def) : src_def)
    db_dict = regulate_LHS!(tb, raw_BFs)
    return Symbolic_BilinearForm[Symbolic_BilinearForm(this_dual,
    simplify_Common(plus(bases))) for (this_dual, bases) in db_dict]
end

collect_BilinearTerms!(tb::TensorTable,
buffer_vec::Vector{SymbolicTerm}, ::Number) = buffer_vec
function collect_BilinearTerms!(tb::TensorTable,
buffer_vec::Vector{SymbolicTerm}, source_word::SymbolicWord)
    sym_attribute = get_VarAttribute(source_word)
    if ~(:INTERNAL_VAR in sym_attribute) && ~(:EXTERNAL_VAR in sym_attribute)
        if isempty(source_word.sd_ids) && (source_word.td_order == 0)
            raw_ids, raw_def = deepcopy(
                DEFINITION_TABLE[source_word.base_variable])
            target_def = substitute_Term!(unroll_And_Simplify(
                raw_def, tb.dim), raw_ids, source_word.c_ids)
            return collect_BilinearTerms!(tb, buffer_vec, target_def)
        end
    end
    return buffer_vec
end
function collect_BilinearTerms!(tb::TensorTable,
buffer_vec::Vector{SymbolicTerm}, source_term::SymbolicTerm)
    term_op = source_term.operation
    if term_op == :Bilinear
        push!(buffer_vec, deepcopy(source_term))
    elseif term_op == :+
        for subterm in source_term.subterms
            collect_BilinearTerms!(tb, buffer_vec, subterm)
        end
    elseif term_op == :*
        sub_vecs = Vector{SymbolicTerm}[collect_BilinearTerms!(tb,
        SymbolicTerm[], subterm) for subterm in source_term.subterms]
        is_normal = isempty.(sub_vecs)
        bilinear_IDs = findall(.~is_normal)
        if length(bilinear_IDs) > 1
            error("One product should only contain one BilinearTerm in: $source_term")
        elseif length(bilinear_IDs) == 1
            other_terms = source_term.subterms[is_normal]
            for bil in sub_vecs[bilinear_IDs[1]]
                new_base_term = mult(push!(deepcopy(other_terms), bil.subterms[2]))
                push!(buffer_vec, construct_Term(:Bilinear, [bil.subterms[1], new_base_term]))
            end
        end
    end
    return buffer_vec
end


function regulate_LHS!(tb::TensorTable, raw_BFs::Vector{SymbolicTerm})
    db_dict = Dict{SymbolicWord, Vector{GroundTerm}}()
    for raw_BF in raw_BFs
        raw_dual_term, raw_base_term = raw_BF.subterms
        for (dual_word, factors) in collect_Variations(raw_dual_term, tb)
            push!(get!(db_dict, dual_word, GroundTerm[]),
            simplify_Common(mult([plus(factors), raw_base_term])))
        end
    end
    return db_dict
end
```

A.3. FEM kernel

The code inside the folder /src/solver defines the FEM kernel.

```
abstract type FEM_Geometry{ArrayType} end #2D / 3D, Mesh / bdy

abstract type FEM_Object{ArrayType} end


abstract type FEM_WP_Mesh{ArrayType} end

abstract type FEM_Tool_Mesh{ArrayType} end

abstract type FEM_Spatial_Discretization{ArrayType} end

abstract type FEM_Temporal_Discretization end


#physics

#--------------------------

mutable struct FEM_Physics{ArrayType}

    global_vars::Dict{Symbol, FEM_Float}

    extra_var::Vector{Symbol}

    boundarys::Vector #each boundary group is a set of ref_edge_IDs

    boundary_weakform_pairs::Dict{FEM_Int, Vector{Symbolic_BilinearForm}}

    domain_weakform::Vector{Symbolic_BilinearForm}

    FEM_Physics(::Type{ArrayType}) where {ArrayType} =

    new{ArrayType}(Dict{Symbol, FEM_Float}(), Symbol[],

    AbstractArray{FEM_Int, 1}[], Dict{FEM_Int, Vector{Symbolic_BilinearForm}}())

end


#assembly

#--------------------------

InnervarInfo = Tuple{Symbol, FEM_Int, Tuple, FEM_Int}

ExtervarInfo = Tuple{Symbol, Symbol, Symbol, Tuple, Tuple}


struct AssembleBilinear
```

```
    base_term::GroundTerm

    dual_info::InnervarInfo
    derivative_info::InnervarInfo
end

struct AssembleWeakform
    residues::Vector{AssembleBilinear}
    linear_gradients::Vector{AssembleBilinear}
    nonlinear_gradients::Vector{AssembleBilinear}

    innervar_infos::Vector{InnervarInfo}
    linear_extervar_infos::Vector{ExtervarInfo}
    extervar_infos::Vector{ExtervarInfo}
end

mutable struct FEM_LocalAssembly
    basic_vars::Vector{Symbol} #basic symbol for x allocation
    local_innervar_infos::Vector{Tuple{Symbol, FEM_Int, FEM_Int}}
    controlpoint_extervars::Vector{Symbol} #local symbol for local var allocation

    assembled_boundary_weakform_pairs::Dict{FEM_Int, AssembleWeakform} #bg_ID, wf
    assembled_weakform::AssembleWeakform

    sparse_entry_ID::FEM_Int
    sparse_unitsize::FEM_Int
    sparse_mapping::Dict{Tuple{FEM_Int, FEM_Int}, FEM_Int}
end

#object
#--------------------------
"""
```

```
    WorkPiece

A `WorkPiece` is a meshed part assigned with some known physics. The attributes are:

* `ref_geometry`, the first order mesh to describe the FEM_Geometry.
* `physics`, the raw PDE weakforms.
* `local_assembly`, the re-organized PDE weakforms with sorted/indexed variables.
* `max_sd_order`, an explicit limit of the maximum spatial derivative order to save memory.
* `element_space`, the information about the spatial discritization,
i.e., interpolation and intergration.
* `mesh`, the mesh regenerated according to the `element_space`
and actually used in simulation.

To add a `Workpiece` with the geometry `ref_geometry`
to the [`FEM_Domain`](@ref) `fem_domain`, the exposed API is:

    add_WorkPiece!(ref_geometry; fem_domain::FEM_Domain)

which returns the `WorkPiece` ID in the `fem_domain`.`workpieces`.
"""
mutable struct WorkPiece{ArrayType} <: FEM_Object{ArrayType}
    ref_geometry::FEM_Geometry{ArrayType}

    physics::FEM_Physics{ArrayType}
    local_assembly::FEM_LocalAssembly

    max_sd_order::Integer
    element_space::FEM_Spatial_Discretization
    mesh::FEM_WP_Mesh{ArrayType}

    function WorkPiece(ref_geometry::FEM_Geometry{ArrayType}) where {ArrayType}
        new{ArrayType}(ref_geometry, FEM_Physics(ArrayType))
    end
end


mutable struct Tool{ArrayType} <: FEM_Object{ArrayType}
    ref_geometry::FEM_Geometry{ArrayType}
    boundarys::Vector #each boundary group is a set of ref_edge_IDs
    mesh::FEM_Tool_Mesh{ArrayType}

    function Tool(ref_geometry::FEM_Geometry{ArrayType}) where {ArrayType}
        new{ArrayType}(ref_geometry, AbstractArray{FEM_Int, 1}[])
    end
end


mutable struct FEM_Contact
    master_id::Integer
    master_type::Symbol
    slave_id::Integer
    bg_pairs::Vector{Tuple{FEM_Int, FEM_Int}}
end


#Time domain
#--------------------
mutable struct GlobalField{ArrayType} #global infos & FEM data, should be separated later
    max_time_level::Integer
    basicfield_size::Integer

    converge_tol::FEM_Float

    t::FEM_Float
    dt::FEM_Float

    x::ArrayType #x0 u0 a0
    dx::ArrayType
    x_star::ArrayType

    residue::ArrayType

    K_I::ArrayType
    K_J_ptr::ArrayType
    K_J::ArrayType
    K_val_ids::AbstractArray
    K_linear::ArrayType
    K_total::ArrayType
    GlobalField(::Type{ArrayType}) where {ArrayType} = new{ArrayType}(0, 0, 0., 0., 1.)
end

"""
    FEM_Domain

A `FEM_Domain` contains everything needed to assemble a linear system
`Kx=d` in FEM. The attributes are:

* `dim`, the dimension, 2 or 3.
* `workpieces`, the array of all the `WorkPiece`s in this domain,
which will be finally solved in fully coupling.
* `tools`, reserved for the external geometry `Tool`, e.g.,
for contact. Not implemented.
* `time_discretization`, the temporal discritization scheme.
Currently only generalized-α method is implemented.
* `globalfield`, the container for sparse `K`, dense `x` and `d` in `Kx=d`.
* `K_linear_func` the generated function to update the linear part of `K`.
* `K_nonlinear_func` the generated function to update the nonlinear part
of `K` and the residue `d`.
* `linear_solver`, the applied linear solver.
```

```
To add a `FEM_Domain` of dimension `dim`, the exposed API is:

    FEM_Domain(; dim::Integer)

which returns the new `FEM_Domain`.
"""
mutable struct FEM_Domain{ArrayType} #workgroup
    dim::Integer
    tensor_table::TensorTable

    workpieces::Vector{WorkPiece{ArrayType}}
    tools::Vector{Tool{ArrayType}}

    time_discretization::FEM_Temporal_Discretization
    globalfield::GlobalField{ArrayType}

    K_linear_func::Function
    K_nonlinear_func::Function
    linear_solver::Function

    FEM_Domain(::Type{ArrayType} = DEFAULT_ARRAYINFO._type; dim::Integer,
    dissipative::Bool = true) where {ArrayType} = new{ArrayType}(dim,
    TensorTable(dim), WorkPiece{ArrayType}[], Tool{ArrayType}[],
    GeneralAlpha(; dissipative = dissipative), GlobalField(ArrayType)) # need to rewrite
end

function add_WorkPiece!(ref_geometry::FEM_Geometry{ArrayType};
fem_domain::FEM_Domain{ArrayType}) where {ArrayType}
    push!(fem_domain.workpieces, WorkPiece(ref_geometry))
    wp_ID = length(fem_domain.workpieces)
    println("Workpiece $wp_ID added!")
    return wp_ID
end


"""
    add_Boundary!(ID::Integer, bdy_ref_edge_IDs::CuVector;
    fem_domain::FEM_Domain = fem_domain, target::Symbol = :WorkPiece)

In a 2D/3D `FEM_Domain` `fem_domain`, mark the segment/face IDs of the `WorkPiece`
`fem_domain`.`workpieces`[`wp_ID`] as a boundary, to assign physics later.
Multiple boundaries are independent from each other and can share the same segment/face IDs.
If target = `:Tool`, the boundary is of `fem_domain`.`tools`[`wp_ID`], not implemented.

The function returns the boundary group ID `bg_ID`.
"""
function add_Boundary!(wp_ID::Integer, bdy_ref_edge_IDs::AbstractVector;
fem_domain::FEM_Domain{ArrayType} = fem_domain, target::Symbol = :WorkPiece) where {ArrayType}
    if target == :WorkPiece
        boundarys = fem_domain.workpieces[wp_ID].physics.boundarys
    elseif target == :Tool
        boundarys = fem_domain.tools[wp_ID].boundarys
    end
    push!(boundarys, FEM_convert(ArrayType, bdy_ref_edge_IDs))
    bg_ID = length(boundarys)
    println("Boundary $bg_ID added to $target $(wp_ID)!")
    return bg_ID
end


"""
    assign_WorkPiece_WeakForm!(wp_ID::Integer, this_term::SymbolicTerm; fem_domain::FEM_Domain)
    assign_Boundary_WeakForm!(wp_ID::Integer, bg_ID::Integer,
    this_term::SymbolicTerm; fem_domain::FEM_Domain)
```

```
The functions assign `this_term`, which is either a bilinear term Bilinear(·, ·),
or a sum of the bilinear terms, to the target `WorkPiece` or boundary.
"""
function assign_Boundary_WeakForm!(wp_ID::Integer, bg_ID::Integer,
this_term; fem_domain::FEM_Domain)
    this_weakform = build_WeakForm(fem_domain.tensor_table, this_term)
    if isempty(this_weakform)
        println("The weakform for Boundary $bg_ID is not added becase it is empty.")
    else
        fem_domain.workpieces[wp_ID].physics.boundary_weakform_pairs[bg_ID] = this_weakform
    end
end
function assign_WorkPiece_WeakForm!(wp_ID::Integer, this_term; fem_domain::FEM_Domain)
    this_weakform = build_WeakForm(fem_domain.tensor_table, this_term)
    if isempty(this_weakform)
        println("The weakform for WorkPiece $wp_ID is not added becase it is empty.")
    else
        fem_domain.workpieces[wp_ID].physics.domain_weakform = this_weakform
    end
end

extract_Words(tb::TensorTable, x) = _extract_Words!(tb,
Set{SymbolicWord}(), Set{SymbolicWord}(), x)
_extract_Words!(tb::TensorTable, internal_words::Set{SymbolicWord},
external_words::Set{SymbolicWord}, source::Number) = internal_words, external_words
function _extract_Words!(tb::TensorTable, internal_words::Set{SymbolicWord},
external_words::Set{SymbolicWord}, source::SymbolicWord)
    var_attribute = get_VarAttribute(source)
    if :INTERNAL_VAR in var_attribute
        push!(internal_words, source)
    elseif :EXTERNAL_VAR in var_attribute
        if (:INTEGRATION_POINT_VAR in var_attribute) && (source.base_variable != :n)
```

```
            _extract_Words!(tb, internal_words, external_words,
            DEFINITION_TABLE[source.base_variable][2])
        else
            push!(external_words, source)
        end
    else
        _extract_Words!(tb, internal_words, external_words, evaluate_Tensor(tb, source))
    end
    return internal_words, external_words
end
_extract_Words!(tb::TensorTable, internal_words::Set{SymbolicWord},
external_words::Set{SymbolicWord},
 source::SymbolicTerm) = _extract_Words!(tb, internal_words, external_words, source.subterms)
_extract_Words!(tb::TensorTable, internal_words::Set{SymbolicWord},
external_words::Set{SymbolicWord},
source::Symbolic_BilinearForm) = _extract_Words!(tb, internal_words,
external_words, source.base_term)
function _extract_Words!(tb::TensorTable, internal_words::Set{SymbolicWord},
external_words::Set{SymbolicWord}, srcs::Vector)
    for _src in srcs
        _extract_Words!(tb, internal_words, external_words, _src)
    end
    return internal_words, external_words
end

construct_InnervarInfo(dim::Integer, source::SymbolicWord,
bvar_mapping::Dict{Symbol, FEM_Int}) =
(word_To_TotalSym(dim, source), source.td_order, Tuple(source.sd_ids),
bvar_mapping[word_To_BaseSym(dim, source)])
construct_ExtervarInfo(dim::Integer, source::SymbolicWord) = (word_To_TotalSym(dim, source),
 word_To_LocalSym(dim, source), source.base_variable, Tuple(source.sd_ids), Tuple(source.c_ids))
```

```
function construct_AssembleWeakform(tb::TensorTable, srcs::Vector{Symbolic_BilinearForm},
 bvar_mapping::Dict{Symbol, FEM_Int})
    dim = tb.dim
    residues, linear_gradients, nonlinear_gradients =
    AssembleBilinear[], AssembleBilinear[], AssembleBilinear[]
    innervar_infos, linear_extervar_infos, extervar_infos =
    InnervarInfo[], ExtervarInfo[], ExtervarInfo[]

    for _src in srcs
        @Takeout (dual_word, base_term) FROM _src
        dual_info = construct_InnervarInfo(dim, dual_word, bvar_mapping)
        innervar_words, extervar_words = extract_Words(tb, base_term)

        push!(residues, AssembleBilinear(base_term, dual_info, tuple(:nothing, 0, (), 0)))
        union!(innervar_infos, InnervarInfo[construct_InnervarInfo(dim,
        word, bvar_mapping) for word in innervar_words])
        union!(extervar_infos, ExtervarInfo[construct_ExtervarInfo(dim,
        word) for word in extervar_words])

        for (diff_word, diffed_termvec) in collect_Variations(base_term, tb)
            diffed_term = simplify_Common(plus(diffed_termvec))
            diff_innervar_words, diff_extervar_words = extract_Words(tb, diffed_term)
            derivative_info = construct_InnervarInfo(dim, diff_word, bvar_mapping)
            this_diff_bilinear = AssembleBilinear(diffed_term, dual_info, derivative_info)
            if isempty(diff_innervar_words) && prod(Bool[(~(:INTEGRATION_POINT_VAR in
            get_VarAttribute(word))) || (word.base_variable == :n)
            for word in diff_extervar_words])
                push!(linear_gradients, this_diff_bilinear)
                union!(linear_extervar_infos, ExtervarInfo[construct_ExtervarInfo(dim,
                word) for word in diff_extervar_words])
            else
                push!(nonlinear_gradients, this_diff_bilinear)
            end
        end
    end
    return @Construct AssembleWeakform
end

function _collect_SparsePos!(buffer::Set{Tuple{FEM_Int, FEM_Int}}, wf::AssembleWeakform)
    for bil in wf.linear_gradients
        push!(buffer, (bil.dual_info[4], bil.derivative_info[4]))
    end
    for bil in wf.nonlinear_gradients
        push!(buffer, (bil.dual_info[4], bil.derivative_info[4]))
    end
    return buffer
end

function collect_SparsePos(wp_wf::AssembleWeakform, bd_wf_pairs::Dict{FEM_Int, AssembleWeakform})
    buffer = _collect_SparsePos!(Set{Tuple{FEM_Int, FEM_Int}}(), wp_wf)
    for (_, wf) in bd_wf_pairs _collect_SparsePos!(buffer, wf) end
    return collect(buffer) |> sort!
end

"""
    initialize_LocalAssembly!(fem_domain::FEM_Domain; explicit_max_sd_order::Integer = 9)
    initialize_LocalAssembly!(tb::TensorTable, workpieces::Vector{WorkPiece};
    explicit_max_sd_order::Integer = 9)

This function preprocesses/reorganizes the weakforms.
The input `explicit_max_sd_order` is the exposed API for
explicitly limit high order spatial derivative.
"""
function initialize_LocalAssembly!(tb::TensorTable,
workpieces::Vector; explicit_max_sd_order::Integer = 9)
    dim = tb.dim
    for wp in workpieces
        @Takeout (extra_var, domain_weakform, boundary_weakform_pairs) FROM wp.physics

        innervar_words, extervar_words = extract_Words(tb, domain_weakform)
        for (_, wf) in boundary_weakform_pairs
            _extract_Words!(tb, innervar_words, extervar_words, wf)
        end

        basic_vars = Symbol[word_To_BaseSym(dim, x)
        for x in innervar_words] |> sort! |> unique!
        bvar_mapping = Dict(var => FEM_Int(i - 1) for (i, var) in enumerate(basic_vars))

        local_innervar_infos = [(word_To_LocalSym(dim, x),
        bvar_mapping[word_To_BaseSym(dim, x)], x.td_order) for x in innervar_words] |> unique!
        controlpoint_extervars = [extra_var; word_To_LocalSym.(dim, filter!(x ->
        :CONTROLPOINT_VAR in get_VarAttribute(x), extervar_words))]

        assembled_boundary_weakform_pairs = Dict(i => construct_AssembleWeakform(tb,
        wf, bvar_mapping) for (i, wf) in boundary_weakform_pairs)
        assembled_weakform = construct_AssembleWeakform(tb, domain_weakform, bvar_mapping)

        sparse_entry_ID, sparse_unitsize = 0, 0

        parse_poses = collect_SparsePos(assembled_weakform, assembled_boundary_weakform_pairs)
        sparse_mapping = Dict(parse_pose => (i - 1)
        for (i, parse_pose) in enumerate(parse_poses))

        wp.local_assembly = @Construct FEM_LocalAssembly
        wp.max_sd_order = min(max(collect_SDOrder(assembled_weakform),
        collect_SDOrder(values(assembled_boundary_weakform_pairs))), explicit_max_sd_order)
        # wp.max_sd_order = min(max(get_MaxSDOrder(innervar_words),
        get_MaxSDOrder(extervar_words)), explicit_max_sd_order)
    end
end
initialize_LocalAssembly!(fem_domain::FEM_Domain; explicit_max_sd_order::Integer = 9) =
initialize_LocalAssembly!(fem_domain.tensor_table,
fem_domain.workpieces; explicit_max_sd_order)

collect_SDOrder(x::InnervarInfo) = length(x[3])
collect_SDOrder(x::ExtervarInfo) = length(x[4])
collect_SDOrder(x::AssembleBilinear) = collect_SDOrder(x.dual_info)
collect_SDOrder(x::AssembleWeakform) = max(collect_SDOrder(x.residues),
collect_SDOrder(x.innervar_infos), collect_SDOrder(x.extervar_infos))
collect_SDOrder(x) = isempty(x) ? 1 : maximum(collect_SDOrder.(x))

# get_MaxSDOrder(words) = isempty(words) ? 1 : maximum([length(x.sd_ids) for x in words])
get_MaxTimeSteps(local_asm::FEM_LocalAssembly) = maximum([x[3]
for x in local_asm.local_innervar_infos])
get_MaxTimeSteps(wp::WorkPiece) = get_MaxTimeSteps(wp.local_assembly)

"""
    assemble_Global_Variables!(; fem_domain::FEM_Domain)

This function allocates the sparse `K`, dense `x` and `d`.
"""
function assemble_Global_Variables!(; fem_domain::FEM_Domain{ArrayType}) where {ArrayType}
    @Takeout (workpieces, globalfield) FROM fem_domain

    basicfield_size = 0
    for wp in workpieces
        @Takeout (mesh, local_assembly) FROM wp
        @Takeout (global_cpID, is_occupied) FROM mesh.controlpoints WITH PREFIX c_
        @Takeout (global_cpIDs, is_occupied,
        controlpoint_IDs) FROM mesh.elements WITH PREFIX el_

        variable_size = mesh.variable_size = sum(c_is_occupied)
        cpIDs = findall(c_is_occupied)
        elIDs = findall(el_is_occupied)
        c_global_cpID[cpIDs] .= (findall(c_is_occupied[c_is_occupied]) .+ basicfield_size)
        el_global_cpIDs[:, elIDs] .= c_global_cpID[el_controlpoint_IDs[:, elIDs]]
        basicfield_size += length(local_assembly.basic_vars) * variable_size
    end

    globalfield.basicfield_size = basicfield_size
    max_time_level = globalfield.max_time_level = workpieces .|> get_MaxTimeSteps |> maximum
    globalfield_size = (max_time_level + 1) * basicfield_size

    globalfield.x = FEM_zeros(ArrayType, FEM_Float, globalfield_size)
    globalfield.dx = FEM_zeros(ArrayType, FEM_Float, globalfield_size)
    globalfield.x_star = FEM_zeros(ArrayType, FEM_Float, globalfield_size)
    globalfield.residue = FEM_zeros(ArrayType, FEM_Float, basicfield_size)

    println("Global field x and d allocated with basic DOF =
    $basicfield_size, global DOF = $globalfield_size.")

    assemble_X!(workpieces, globalfield)

    assemble_SparseID!(workpieces, globalfield)
end

"""
    assemble_X!(workpieces::Vector{WorkPiece}, globalfield::GlobalField)

This function assembles/synchronizes local data to `globalfield`.`x`, i.e.,
to setup the initial values for some
`workpiece`.`mesh`.`controlpoint`.`sym`,
like `workpiece.mesh.controlpoint.T`.
"""
function assemble_X!(workpieces::Vector{WorkPiece{ArrayType}},
globalfield::GlobalField{ArrayType}) where {ArrayType}
    for wp in workpieces
        @Takeout (mesh.controlpoints, mesh.variable_size,
        local_assembly.local_innervar_infos) FROM wp
        @Takeout (global_cpID, is_occupied) FROM controlpoints WITH PREFIX c_
        local_data = get_Data(controlpoints)
        cpIDs = findall(c_is_occupied)
        basic_cpID = c_global_cpID[cpIDs]
        for (local_sym, basic_pos, td_order) in local_innervar_infos
            global_cpIDs = basic_cpID .+
            (basic_pos * variable_size + td_order * globalfield.basicfield_size)
            globalfield.x[global_cpIDs] .= local_data[local_sym][cpIDs]
        end
    end
end

"""
    dessemble_X!(workpieces::Vector{WorkPiece}, globalfield::GlobalField)

This function dessembles `globalfield`.`x` to local data, i.e., for
ploting some `workpiece`.`mesh`.`controlpoint`.`sym`, like `workpiece.mesh.controlpoint.T`.
"""
function dessemble_X!(workpieces::Vector{WorkPiece{ArrayType}},
globalfield::GlobalField{ArrayType}) where {ArrayType}
    for wp in workpieces
        @Takeout (mesh.controlpoints, mesh.variable_size,
        local_assembly.local_innervar_infos) FROM wp
        @Takeout (global_cpID, is_occupied) FROM controlpoints WITH PREFIX c_
        local_data = get_Data(controlpoints)
        cpIDs = findall(c_is_occupied)
        basic_cpID = c_global_cpID[cpIDs]
        for (local_sym, basic_pos, td_order) in local_innervar_infos
            global_cpIDs = basic_cpID .+
            (basic_pos * variable_size + td_order * globalfield.basicfield_size)
            local_data[local_sym][cpIDs] .= globalfield.x[global_cpIDs]
        end
    end
end

function assemble_SparseID!(workpieces::Vector{WorkPiece{ArrayType}},
globalfield::GlobalField{ArrayType}) where {ArrayType}
    last_sparse_ID = 0
    cp_cp_2_sparseID_dicts = FEM_Dict[]
    for wp in workpieces
        @Takeout (mesh, local_assembly) FROM wp
        @Takeout (sparse_IDs_by_el, controlpoint_IDs, is_occupied) FROM mesh.elements

        elIDs = findall(is_occupied)
        elnum = length(elIDs)
        el_cp_num = size(controlpoint_IDs, 1)

        local_assembly.sparse_entry_ID = last_sparse_ID
        cp_cp_2_sparseID = dumb_FEM_Dict_Init(ArrayType, Int32)

        cp_cp_keys = FEM_zeros(ArrayType, UInt64, el_cp_num * el_cp_num * elnum)
        counter = 0
        for i = 1:el_cp_num
            for j = 1:el_cp_num
                dict_ids = (counter * elnum + 1):((counter + 1) * elnum)
                cp_cp_keys[dict_ids] .=
                I32I32_To_UI64.(controlpoint_IDs[i, elIDs],
                controlpoint_IDs[j, elIDs])
                counter += 1
            end
        end
        FEM_Dict_SetID!(cp_cp_2_sparseID, cp_cp_keys)

        total_dict_IDs = get_Total_IDs(cp_cp_2_sparseID)
        this_unitsize = local_assembly.sparse_unitsize = length(total_dict_IDs)

        unsorted_keys = cp_cp_2_sparseID.keys[total_dict_IDs]
        unsorted_dict_IDs = FEM_Dict_GetID(cp_cp_2_sparseID, unsorted_keys)

        local_sparse_ids = findall(FEM_ones(ArrayType, Bool, this_unitsize)) .+ last_sparse_ID
        cp_cp_2_sparseID.vals[unsorted_dict_IDs] .= local_sparse_ids

        counter = 0
        for i = 1:el_cp_num
            for j = 1:el_cp_num
                cp_cp_keys = I32I32_To_UI64.(controlpoint_IDs[i, elIDs],
                controlpoint_IDs[j, elIDs])
                local_dict_IDs = FEM_Dict_GetID(cp_cp_2_sparseID, cp_cp_keys)
                sparse_IDs_by_el[i, j, elIDs] .= cp_cp_2_sparseID.vals[local_dict_IDs]
            end
        end
        last_sparse_ID += length(local_assembly.sparse_mapping) * this_unitsize
        push!(cp_cp_2_sparseID_dicts, cp_cp_2_sparseID)

        println("Temporary hash table with $(report_memory(cp_cp_2_sparseID))
        is allocated for sparse matrix assembly")
    end
```

```
    println("Allocating sparse matrix with size = $((last_sparse_ID *
    (3 * sizeof(FEM_Int) + 2 * sizeof(FEM_Float)) + globalfield.basicfield_size
    * sizeof(FEM_Int)) / MEM_UNIT.u_size) $(MEM_UNIT.u_name).")
    globalfield.K_I = FEM_zeros(ArrayType, FEM_Int, last_sparse_ID)
    globalfield.K_J = FEM_zeros(ArrayType, FEM_Int, last_sparse_ID)
    assemble_KIJ!(workpieces, globalfield, cp_cp_2_sparseID_dicts)

    globalfield.K_val_ids = sort_CUSPARSE_COO!(length(globalfield.residue),
    globalfield.K_I, globalfield.K_J) #later save the space

    globalfield.K_J_ptr = generate_J_ptr(globalfield.K_I, globalfield.basicfield_size)

    globalfield.K_linear = FEM_zeros(ArrayType, FEM_Float, last_sparse_ID)
    globalfield.K_total = FEM_zeros(ArrayType, FEM_Float, last_sparse_ID)

    println("Global K is allocated as a sparse maxtrix of $last_sparse_ID slots.")

    println("Global field finally in total takes up $(report_memory(globalfield)),
    where the x and d takes $((3 * length(globalfield.x) + globalfield.basicfield_size)
    * sizeof(FEM_Float)  / MEM_UNIT.u_size) $(MEM_UNIT.u_name) and the sparse K takes up
    $(last_sparse_ID * (3 * sizeof(FEM_Int) + 2 *
    sizeof(FEM_Float)) / MEM_UNIT.u_size) $(MEM_UNIT.u_name).")
end

function assemble_KIJ!(workpieces::Vector{WorkPiece{ArrayType}},
globalfield::GlobalField, cp_cp_2_sparseID_dicts::Vector{FEM_Dict}) where ArrayType
    @Takeout (K_I, K_J) FROM globalfield
    for (wp, cp_cp_2_sparseID) in zip(workpieces, cp_cp_2_sparseID_dicts)
        @Takeout (sparse_entry_ID, sparse_unitsize, sparse_mapping) FROM wp.local_assembly
        @Takeout (controlpoints, variable_size) FROM wp.mesh
        for ((dual_pos, base_pos), sparse_unit_num) in collect(sparse_mapping)
            sparse_ID_shift = sparse_unit_num * sparse_unitsize
            cpID_shift = (dual_pos, base_pos) .* variable_size
            _assemble_KIJ!(cpID_shift, sparse_ID_shift, K_I, K_J,
            cp_cp_2_sparseID.keys, cp_cp_2_sparseID.vals)
        end
    end
end

@Dumb_GPU_Kernel _assemble_KIJ!(cpID_shift, sparse_ID_shift, K_I::Array,
K_J::Array, keys::Array, sparseID::Array) begin #can be rewrote
    this_key = keys[thread_idx]
    this_key == 0 && return

    this_sparse_ID = sparseID[thread_idx] + sparse_ID_shift
    dual_shift, base_shift = cpID_shift

    this_cpID = UI64_To_UpperHalf(this_key)
    that_cpID = UI64_To_LowerHalf(this_key)

    K_I[this_sparse_ID] = this_cpID + dual_shift
    K_J[this_sparse_ID] = that_cpID + base_shift
end

mutable struct GeneralAlpha <: FEM_Temporal_Discretization
    alpha_params::Tuple{Vararg{FEM_Float}}
    gamma_params::Tuple{Vararg{FEM_Float}}
    beta_params::Vector{FEM_Float}
    K_params::Vector{FEM_Float}
end
GeneralAlpha(; dissipative::Bool = false) = GeneralAlpha((1., 1., 1.) .|> FEM_Float,
(dissipative ? (1., 1.) : (0.5, 0.5)) .|> FEM_Float, FEM_Float[], FEM_Float[])
```

```
# GeneralAlpha() = GeneralAlpha((1., 1., 1.) .|> FEM_Float,
(0.5, 0.5) .|> FEM_Float, FEM_Float[], FEM_Float[])

function update_Time!(globalfield::GlobalField, time_discretization::GeneralAlpha)
    globalfield.t += globalfield.dt

    prod_gamma = [prod(time_discretization.gamma_params[1:i])
    for i = 0:globalfield.max_time_level]
    dt_params = [globalfield.dt ^ i for i = 0:globalfield.max_time_level]
    time_discretization.beta_params = 1 ./ (prod_gamma .* dt_params)

    time_discretization.K_params = time_discretization.alpha_params[
        1:(globalfield.max_time_level + 1)] .* time_discretization.beta_params
end

function initialize_dx!(globalfield::GlobalField, time_discretization::GeneralAlpha)
    @Takeout (max_time_level, basicfield_size, dt, x, dx) FROM globalfield
    @Takeout (gamma_params) FROM time_discretization
    dx.= 0.
    for t_level = max_time_level:-1:1
        low_start, low_final   = (t_level - 1) *
        basicfield_size + 1,  t_level      * basicfield_size
        high_start, high_final =  t_level      *
        basicfield_size + 1, (t_level + 1) * basicfield_size

        dx[low_start:low_final] .= dt * (x[high_start:high_final] .+
        gamma_params[t_level] * dx[high_start:high_final])
    end
end

function update_dx!(globalfield::GlobalField,
delta_x::CuVector, time_discretization::GeneralAlpha)
```

```
    @Takeout (max_time_level, basicfield_size, x, dx) FROM globalfield
    @Takeout beta_params FROM time_discretization
    for t_level = 0:max_time_level
        start_id, final_id = t_level * basicfield_size + 1, (t_level + 1) * basicfield_size
        dx[start_id:final_id] .+= beta_params[t_level + 1] * delta_x
    end
end

function update_x_star!(globalfield::GlobalField, time_discretization::GeneralAlpha)
    @Takeout (max_time_level, basicfield_size, dt, x, dx) FROM globalfield
    @Takeout (alpha_params) FROM time_discretization
    globalfield.x_star .= x
    for t_level = 0:max_time_level
        start_id, final_id = t_level * basicfield_size + 1, (t_level + 1) * basicfield_size
        globalfield.x_star[start_id:final_id] .+=
        alpha_params[t_level + 1] * dx[start_id:final_id]
    end
end

normalized_norm(x) = norm(x) / sqrt(length(x))

"""
    update_OneStep!(time_discretization::GeneralAlpha;
    max_iter::Integer = 4, fem_domain::FEM_Domain)

This function calculates `K(Δx)=d` and updates `x += (Δx)`. `max_iter`
 determines the maximum iteration for Δx with different `K` and `d` in nonlinear cases.
The converge tolerance is determined by `fem_domain`.`globalfield`.`converge_tol`
 while the linear solver is `fem_domain`.`linear_solver`.
"""
function update_OneStep!(time_discretization::GeneralAlpha;
max_iter::Integer = 4, fem_domain::FEM_Domain)
```

```
    @Takeout (workpieces, globalfield) FROM fem_domain

    update_Time!(globalfield, time_discretization)
    initialize_dx!(globalfield, time_discretization)
    fem_domain.K_linear_func(time_discretization; fem_domain = fem_domain)
    counter = -1
    while true
        update_x_star!(globalfield, time_discretization)
        @time fem_domain.K_nonlinear_func(time_discretization; fem_domain = fem_domain)
        res = normalized_norm(globalfield.residue)

        println(repeat("_", 100))
        println("step $(counter += 1) residue = $res")

        (res < globalfield.converge_tol || counter > max_iter) && break

        @time delta_x = fem_domain.linear_solver(globalfield)
        update_dx!(globalfield, .- delta_x, time_discretization)
    end
    globalfield.x .+= globalfield.dx
end

function declare_Innervar_GPU(dim::Integer, innervar_infos::Vector{InnervarInfo},
max_sd_order::Integer; itg_val_fixed::Bool)
    declare_code = Expr(:block)
    for (total_sym, td_order, sd_order, basic_pos) in sort(innervar_infos)
        push!(declare_code.args, :($total_sym = FEM_buffer(ArrayType,
        FEM_Float, local_itg_func_num, elnum)))

        length(sd_order) > max_sd_order && continue
        sd_IDs = sd_ids_To_sd_IDs(dim, sd_order)
        push!(declare_code.args, itg_val_fixed ?
            :(_Var_Cut(local_integral_vals, $sd_IDs, ($td_order * basicfield_size +
            $basic_pos * variable_size), global_cpIDs, x_star, $total_sym, elIDs)) :
            :(_Var_Basic(local_integral_vals, $sd_IDs, ($td_order * basicfield_size +
            $basic_pos * variable_size), global_cpIDs, x_star,
            $total_sym, local_itg_hostIDs, elIDs)))
    end
    return declare_code
end

function declare_Extervar_GPU(dim::Integer, extervar_infos::Vector{ExtervarInfo},
max_sd_order::Integer; itg_val_fixed::Bool, is_boundary::Bool = false)
    var_host = is_boundary ? :facets : :elements
    declare_code = Expr(:block)
    for (total_sym, local_sym, type_sym, sd_order, c_ids) in sort(extervar_infos)
        var_attribute = get_VarAttribute(type_sym)
        if :GLOBAL_VAR in var_attribute
            if type_sym == :t
                arg = :($total_sym = globalfield.t)
            elseif type_sym == :dt
                arg = :($total_sym = globalfield.dt)
            else
                arg = :($total_sym = wp.physics.global_vars[$(Meta.quot(total_sym))])
            end
        elseif :CONTROLPOINT_VAR in var_attribute
            push!(declare_code.args, :($total_sym = FEM_buffer(ArrayType, FEM_Float,
            local_itg_func_num, elnum)))

            length(sd_order) > max_sd_order && continue
            sd_IDs = sd_ids_To_sd_IDs(dim, sd_order)
            arg = itg_val_fixed ? :(_Var_Cut(local_integral_vals, $sd_IDs, 0,
            elements.controlpoint_IDs, controlpoints.$local_sym, $total_sym, elIDs)) :
                                  :(_Var_Basic(local_integral_vals, $sd_IDs, 0,
                                  elements.controlpoint_IDs, controlpoints.$local_sym,
                                  $total_sym, local_itg_hostIDs, elIDs))
        elseif :INTEGRATION_POINT_VAR in var_attribute
            isempty(sd_order) || error("Integration point variable cant have
            spatial derivative, use controlpoint variable instead")
            if type_sym == :n
                length(c_ids) == 1 || error("normal only has one direction")
                is_boundary || error("Body don't have normal")
                arg = :($total_sym = facets.normal_directions[:, $(c_ids[1]), facet_IDs])
            else
                error("Unresolved external variable")
            end
        else
            error("Unresolved external variable")
        end
        push!(declare_code.args, arg)
    end
    return declare_code
end

function gen_K_Linear_GPU(tb::TensorTable, asm_wf::AssembleWeakform,
sparse_mapping::Dict{Tuple{FEM_Int, FEM_Int}, FEM_Int}, max_sd_order::Integer;
    is_boundary::Bool = false, itg_val_fixed::Bool, itg_weight_fixed::Bool)
    dim = tb.dim
    @Takeout (linear_gradients, linear_extervar_infos) FROM asm_wf
    extervar_declaration = declare_Extervar_GPU(dim, linear_extervar_infos, max_sd_order;
    itg_val_fixed = itg_val_fixed, is_boundary = is_boundary)

    K_func_code = Expr(:block)
    intermediate_code, declared_syms = Expr[], Set{Symbol}()
    for this_bilinear in linear_gradients
        func_exs = parse_Term2Expr!(intermediate_code, declared_syms,
        tb, this_bilinear.base_term)
        _, _, dual_sd_order, dual_basic_pos = this_bilinear.dual_info
        _, derivative_td_order, derivative_sd_order,
        derivative_basic_pos = this_bilinear.derivative_info
        sparse_unit_num = sparse_mapping[(dual_basic_pos, derivative_basic_pos)]

        max(length(dual_sd_order), length(derivative_sd_order)) > max_sd_order && continue
        dual_sd_IDs = sd_ids_To_sd_IDs(dim, dual_sd_order)
        base_sd_IDs = sd_ids_To_sd_IDs(dim, derivative_sd_order)

        local_func_code = quote
            sparse_ID_shift = $sparse_unit_num * sparse_unitsize
        end

        for this_func_ex in func_exs
            val_arg = itg_weight_fixed ? :(vals = @. $this_func_ex *
            K_params[$(derivative_td_order + 1)] * local_integral_weights) :
            :(vals = @. $this_func_ex *
            K_params[$(derivative_td_order + 1)] * local_integral_weights[:, local_itg_hostIDs])
            ker_arg = itg_val_fixed ? :(_Kval_Cut(local_integral_vals, $dual_sd_IDs,
            $base_sd_IDs, vals, sparse_IDs_by_el, sparse_ID_shift, K_linear, elIDs)) :
            :(_Kval_Basic(local_integral_vals, $dual_sd_IDs,
            $base_sd_IDs, vals, sparse_IDs_by_el, sparse_ID_shift,
            K_linear, local_itg_hostIDs, elIDs))
            push!(local_func_code.args, val_arg)
            push!(local_func_code.args, ker_arg)
        end
        push!(K_func_code.args, local_func_code)
    end
    intermediate_declaration = Expr(:block, intermediate_code...)
    final_block = quote
        $extervar_declaration
        $intermediate_declaration
        $K_func_code
    end
    return final_block
end

function gen_Res_K_NonLinear_GPU(tb::TensorTable, asm_wf::AssembleWeakform,
sparse_mapping::Dict{Tuple{FEM_Int, FEM_Int}, FEM_Int}, max_sd_order::Integer;
    is_boundary::Bool = false, itg_val_fixed::Bool, itg_weight_fixed::Bool)
    dim = tb.dim
    @Takeout (residues, nonlinear_gradients, innervar_infos, extervar_infos) FROM asm_wf
    innervar_declaration = declare_Innervar_GPU(dim, innervar_infos, max_sd_order;
    itg_val_fixed = itg_val_fixed)
    extervar_declaration = declare_Extervar_GPU(dim, extervar_infos, max_sd_order;
    itg_val_fixed = itg_val_fixed, is_boundary = is_boundary)

    res_func_code = Expr(:block)
    intermediate_code, declared_syms = Expr[], Set{Symbol}()
    for this_bilinear in asm_wf.residues
        func_exs = parse_Term2Expr!(intermediate_code, declared_syms,
        tb, this_bilinear.base_term)
        _, _, dual_sd_order, dual_basic_pos = this_bilinear.dual_info
        length(dual_sd_order) > max_sd_order && continue
        dual_sd_IDs = sd_ids_To_sd_IDs(dim, dual_sd_order)

        local_func_code = Expr(:block)
        for this_func_ex in func_exs
            val_arg = itg_weight_fixed ? :(vals = @. $this_func_ex * local_integral_weights) :
                   :(vals = @. $this_func_ex * local_integral_weights[:, local_itg_hostIDs])
            ker_arg = itg_val_fixed ? :(_Res_Cut(local_integral_vals, $dual_sd_IDs, vals,
            $dual_basic_pos * variable_size, global_cpIDs, residue, elIDs)) :
            :(_Res_Basic(local_integral_vals, $dual_sd_IDs, vals,
            $dual_basic_pos * variable_size, global_cpIDs, residue, local_itg_hostIDs, elIDs))
            push!(local_func_code.args, val_arg)
            push!(local_func_code.args, ker_arg)
        end
        push!(res_func_code.args, local_func_code)
    end

    K_func_code = Expr(:block)
    for this_bilinear in nonlinear_gradients
        func_exs = parse_Term2Expr!(intermediate_code, declared_syms,
        tb, this_bilinear.base_term)
        _, _, dual_sd_order, dual_basic_pos = this_bilinear.dual_info
        _, derivative_td_order, derivative_sd_order, derivative_basic_pos =
        this_bilinear.derivative_info
        sparse_unit_num = sparse_mapping[(dual_basic_pos, derivative_basic_pos)]

        max(length(dual_sd_order), length(derivative_sd_order)) > max_sd_order && continue
        dual_sd_IDs = sd_ids_To_sd_IDs(dim, dual_sd_order)
        base_sd_IDs = sd_ids_To_sd_IDs(dim, derivative_sd_order)

        local_func_code = quote
            sparse_ID_shift = $sparse_unit_num * sparse_unitsize
        end

        for this_func_ex in func_exs
            val_arg = itg_weight_fixed ? :(vals = @. $this_func_ex *
            K_params[$(derivative_td_order + 1)] * local_integral_weights) :
            :(vals = @. $this_func_ex *
            K_params[$(derivative_td_order + 1)] * local_integral_weights[:, local_itg_hostIDs])
            ker_arg = itg_val_fixed ? :(_Kval_Cut(local_integral_vals, $dual_sd_IDs,
            $base_sd_IDs, vals, sparse_IDs_by_el, sparse_ID_shift, K_total, elIDs)) :
            :(_Kval_Basic(local_integral_vals, $dual_sd_IDs,
            $base_sd_IDs, vals, sparse_IDs_by_el, sparse_ID_shift, K_total,
            local_itg_hostIDs, elIDs))
            push!(local_func_code.args, val_arg)
            push!(local_func_code.args, ker_arg)
        end
        push!(K_func_code.args, local_func_code)
    end
    intermediate_declaration = Expr(:block, intermediate_code...)
    final_block = quote
        $innervar_declaration
        $extervar_declaration
        $intermediate_declaration
        $res_func_code
        $K_func_code
    end
    return final_block
end

function gen_CodeBody(fem_genfunc::Function; fem_domain::FEM_Domain)
    @Takeout (tensor_table, workpieces, globalfield) FROM fem_domain

    wp_total_code = Expr(:block)
    for (wp_ID, wp) in pairs(workpieces)
        @Takeout (max_sd_order, local_assembly) FROM wp
        @Takeout (assembled_boundary_weakform_pairs,
        assembled_weakform, sparse_mapping) FROM local_assembly
        if wp.element_space isa Classical_Discretization
            el_block = fem_genfunc(tensor_table, assembled_weakform, sparse_mapping,
            max_sd_order; is_boundary = false,
            itg_val_fixed = false, itg_weight_fixed = false)
            el_loop = quote
                elIDs = findall(elements.is_occupied)
                elnum = length(elIDs)
                local_itg_hostIDs = elIDs
                local_integral_vals = elements.integral_vals
                local_integral_weights = elements.integral_weights
                local_itg_func_num = size(local_integral_weights)[1]
                $el_block
            end
            bg_total_loop = quote
                local_integral_vals = facets.integral_vals
                local_integral_weights = facets.integral_weights
                local_itg_func_num = size(local_integral_weights)[1]
            end
            for (bg_ID, bg_asm_wf) in assembled_boundary_weakform_pairs
                bg_local_block = fem_genfunc(tensor_table, bg_asm_wf, sparse_mapping,
                max_sd_order; is_boundary = true,
                itg_val_fixed = false, itg_weight_fixed = false)
                bg_loop = quote
                    facet_IDs = bg_fIDs[$bg_ID]
                    elIDs = facets.element_ID[facet_IDs]
                    elnum = length(elIDs)
                    local_itg_hostIDs = facet_IDs
                    $bg_local_block
                end
                push!(bg_total_loop.args, bg_loop)
            end
            wp_local_code = quote
                wp = workpieces[$wp_ID]
                @Takeout (controlpoints, facets, elements, bg_fIDs, variable_size) FROM wp.mesh
                @Takeout sparse_unitsize FROM wp.local_assembly
                @Takeout (global_cpIDs, sparse_IDs_by_el) FROM elements
                $el_loop
                $bg_total_loop
```

```
            end
        end
        push!(wp_total_code.args, wp_local_code)
    end
    return wp_total_code
end
"""
    compile_Updater_GPU(; domain_ID::Integer, fem_domain::FEM_Domain)


The function generates `fem_domain`.`K_linear_func` and `fem_domain`.`K_nonlinear_func`.
The input `domain_ID` is only used to generate the function name.
"""
function compile_Updater_GPU(; domain_ID::Integer, fem_domain::FEM_Domain)
    linear_func_name = Symbol("update_K_Linear_", domain_ID)
    linear_func_body = :(
    function ($linear_func_name)(time_discretization::GeneralAlpha;
    fem_domain::FEM_Domain{ArrayType}) where {ArrayType}
        @Takeout (workpieces, globalfield) FROM fem_domain
        @Takeout (basicfield_size, K_linear) FROM fem_domain.globalfield
        @Takeout K_params FROM time_discretization
        K_linear .= 0.
        $(gen_CodeBody(gen_K_Linear_GPU; fem_domain = fem_domain))
    end)


    nonlinear_func_name = Symbol("update_K_NonLinear_", domain_ID)
    nonlinear_func_body = :(
    function ($nonlinear_func_name)(time_discretization::GeneralAlpha;
    fem_domain::FEM_Domain{ArrayType}) where {ArrayType}
        @Takeout (workpieces, globalfield) FROM fem_domain
        @Takeout (basicfield_size, x_star, residue, K_linear, K_total) FROM globalfield
        @Takeout K_params FROM time_discretization
        residue .= 0.
        K_total .= K_linear
        $(gen_CodeBody(gen_Res_K_NonLinear_GPU; fem_domain = fem_domain))
    end)

    fem_domain.K_linear_func = eval(linear_func_body)
    fem_domain.K_nonlinear_func = eval(nonlinear_func_body)

    return (linear_func_body, nonlinear_func_body) .|> striplines .|> stripblocks
end

@Dumb_GPU_Kernel _Var_Basic(itp_vals::Array, sd_IDs, cpID_shift,
el_g_cpIDs::Array, x_star::Array, target::Array, itg_hostIDs::Array,
elIDs::Array) begin #Note this is a sum on itp, not itg
    itg_func_num = size(itp_vals, 1)
    itp_num = size(itp_vals, 2)

    this_elID = elIDs[thread_idx]
    this_itghostID = itg_hostIDs[thread_idx]
    for this_base_id = 1:itp_num
        this_cp_ID = el_g_cpIDs[this_base_id, this_elID] + cpID_shift
        for this_itg_id = 1:itg_func_num
            CUDA.@atomic target[this_itg_id, thread_idx] += itp_vals[this_itg_id,
            this_base_id, sd_IDs..., this_itghostID] * x_star[this_cp_ID]
        end
    end
end

@Dumb_GPU_Kernel _Var_Cut(itp_vals::Array, sd_IDs, cpID_shift, el_g_cpIDs::Array,
x_star::Array, target::Array, elIDs::Array) begin #Note this is a sum on itp, not itg
    itg_func_num = size(itp_vals, 1)
    itp_num = size(itp_vals, 2)
    this_elID = elIDs[thread_idx]
    for this_base_id = 1:itp_num
        this_cp_ID = el_g_cpIDs[this_base_id, this_elID] + cpID_shift
        for this_itg_id = 1:itg_func_num
            CUDA.@atomic target[this_itg_id, thread_idx] += itp_vals[this_itg_id,
            this_base_id, sd_IDs...] * x_star[this_cp_ID]
        end
    end
end

@Dumb_GPU_Kernel _Kval_Basic(itp_vals::Array, dual_sd_IDs, base_sd_IDs,
vals::Array, sparse_IDs_by_el::Array, sparse_ID_shift, K_val::Array,
itg_hostIDs::Array, elIDs::Array) begin #Note all local
    itg_func_num = size(itp_vals, 1)
    itp_num = size(itp_vals, 2)

    this_elID = elIDs[thread_idx]
    this_itghostID = itg_hostIDs[thread_idx]
    for this_dual_id = 1:itp_num
        for this_base_id = 1:itp_num
            this_sparse_ID =
            sparse_IDs_by_el[this_dual_id, this_base_id, this_elID] + sparse_ID_shift
            sum = 0.
            for this_itg_id = 1:itg_func_num
                sum += itp_vals[this_itg_id, this_dual_id, dual_sd_IDs...,
                this_itghostID] *
                itp_vals[this_itg_id, this_base_id, base_sd_IDs...,
                this_itghostID] * vals[this_itg_id, thread_idx]
            end
            CUDA.@atomic K_val[this_sparse_ID] += sum
        end
    end
```

```
end


@Dumb_GPU_Kernel _Kval_Cut(itp_vals::Array, dual_sd_IDs, base_sd_IDs, vals::Array,
sparse_IDs_by_el::Array, sparse_ID_shift, K_val::Array, elIDs::Array) begin #Note all local
    itg_func_num = size(itp_vals, 1)
    itp_num = size(itp_vals, 2)

    this_elID = elIDs[thread_idx]
    for this_dual_id = 1:itp_num
        for this_base_id = 1:itp_num
            this_sparse_ID =
            sparse_IDs_by_el[this_dual_id, this_base_id, this_elID] + sparse_ID_shift
            sum = 0.
            for this_itg_id = 1:itg_func_num
                sum += itp_vals[this_itg_id, this_dual_id, dual_sd_IDs...] *
            itp_vals[this_itg_id, this_base_id, base_sd_IDs...] * vals[this_itg_id, thread_idx]
            end
            CUDA.@atomic K_val[this_sparse_ID] += sum
        end
    end
end


@Dumb_GPU_Kernel _Res_Basic(itp_vals::Array, dual_sd_IDs, vals::Array, cpID_shift,
el_g_cpIDs::Array, residue::Array, itg_hostIDs::Array, elIDs::Array) begin
    itg_func_num = size(itp_vals, 1)
    itp_num = size(itp_vals, 2)

    this_elID = elIDs[thread_idx]
    this_itghostID = itg_hostIDs[thread_idx]
    for this_dual_id = 1:itp_num
        this_cp_ID = el_g_cpIDs[this_dual_id, this_elID]
        sum = 0.
        for this_itg_id = 1:itg_func_num
            sum += itp_vals[this_itg_id, this_dual_id, dual_sd_IDs...,
            this_itghostID] * vals[this_itg_id, thread_idx]
        end
        CUDA.@atomic residue[this_cp_ID + cpID_shift] += sum
    end
end

@Dumb_GPU_Kernel _Res_Cut(itp_vals::Array, dual_sd_IDs, vals::Array,
cpID_shift, el_g_cpIDs::Array, residue::Array, elIDs::Array) begin
    itg_func_num = size(itp_vals, 1)
    itp_num = size(itp_vals, 2)

    this_elID = elIDs[thread_idx]
    for this_dual_id = 1:itp_num
        this_cp_ID = el_g_cpIDs[this_dual_id, this_elID]
        sum = 0.
        for this_itg_id = 1:itg_func_num
            sum +=
            itp_vals[this_itg_id, this_dual_id, dual_sd_IDs...] * vals[this_itg_id, thread_idx]
        end
        CUDA.@atomic residue[this_cp_ID + cpID_shift] += sum
    end
end
```

A.4. Mesh System

The code inside the folder /src/mesh defines the mesh system.

```
mutable struct Geo_Vertex2D

    x1::FEM_Float

    x2::FEM_Float

end


mutable struct Geo_Vertex3D

    x1::FEM_Float

    x2::FEM_Float

    x3::FEM_Float

end


mutable struct Geo_Segment

    vertex_IDs::Vector{FEM_Int}

end


mutable struct Geo_Face

    vertex_IDs::Vector{FEM_Int}

    segment_IDs::Vector{FEM_Int} #has an order

end


mutable struct Geo_Block

    vertex_IDs::Vector{FEM_Int}

    segment_IDs::Vector{FEM_Int} #has an order

    face_IDs::Vector{FEM_Int}

end


mutable struct Geo_TotalMesh2D{ArrayType} <: FEM_Geometry{ArrayType}

    vertices::FEM_Table{ArrayType}

    segments::FEM_Table{ArrayType}

    faces::FEM_Table{ArrayType}

end
```

```
mutable struct Geo_TotalMesh3D{ArrayType} <: FEM_Geometry{ArrayType}

    vertices::FEM_Table{ArrayType}

    segments::FEM_Table{ArrayType}

    faces::FEM_Table{ArrayType}

    blocks::FEM_Table{ArrayType}

end


mutable struct Geo_BoundaryMesh2D{ArrayType} <: FEM_Geometry{ArrayType}

    vertices::FEM_Table{ArrayType}

    segments::FEM_Table{ArrayType}

end


mutable struct Geo_BoundaryMesh3D{ArrayType} <: FEM_Geometry{ArrayType}

    vertices::FEM_Table{ArrayType}

    segments::FEM_Table{ArrayType}

    faces::FEM_Table{ArrayType}

end


function Geo_TotalMesh2D(::Type{ArrayType}, mesh_type::Symbol) where {ArrayType}

    if mesh_type == :SIMPLEX

        vertex_per_face = 3

    elseif mesh_type == :CUBE

        vertex_per_face = 4

    else

        error("Undefined mesh type")

    end

    vertex_example = Geo_Vertex2D(fill(FEM_Float(0.), 2)...)

    segment_example = Geo_Segment(zeros(FEM_Int, 2))

    face_example = Geo_Face(zeros(FEM_Int, vertex_per_face), zeros(FEM_Int, vertex_per_face))

    examples = (vertex_example, segment_example, face_example)

    return Geo_TotalMesh2D((construct_FEM_Table.(ArrayType, examples))...)

end
```

```julia
function Geo_TotalMesh3D(::Type{ArrayType}, mesh_type::Symbol) where {ArrayType}
    if mesh_type == :SIMPLEX
        vertex_per_face = 3
        vertex_per_block = 4
        segment_per_block = 6
        face_per_block = 4
    elseif mesh_type == :CUBE
        vertex_per_face = 4
        vertex_per_block = 8
        segment_per_block = 12
        face_per_block = 6
    else
        error("Undefined mesh type")
    end
    vertex_example = Geo_Vertex3D(fill(FEM_Float(0.), 3)...)
    segment_example = Geo_Segment(zeros(FEM_Int, 2))
    face_example = Geo_Face(zeros(FEM_Int, vertex_per_face),
    zeros(FEM_Int, vertex_per_face))
    block_example = Geo_Block(zeros(FEM_Int, vertex_per_block),
    zeros(FEM_Int, segment_per_block), zeros(FEM_Int, face_per_block))
    examples = (vertex_example, segment_example, face_example, block_example)
    return Geo_TotalMesh3D((construct_FEM_Table.(ArrayType, examples))...)
end

function Geo_BoundaryMesh2D(::Type{ArrayType}) where {ArrayType}
    vertex_example = Geo_Vertex2D(fill(FEM_Float(0.), 2)...)
    segment_example = Geo_Segment(zeros(FEM_Int, 2))
    examples = (vertex_example, segment_example)
    return Geo_BoundaryMesh2D((construct_FEM_Table.(ArrayType, examples))...)
end
```

```
function Geo_BoundaryMesh3D(::Type{ArrayType}) where {ArrayType}
    vertex_example = Geo_Vertex3D(fill(FEM_Float(0.), 3)...)
    segment_example = Geo_Segment(zeros(FEM_Int, 2))
    face_example = Geo_Face(zeros(FEM_Int, 3), zeros(FEM_Int, 3))
    examples = (vertex_example, segment_example, face_example)
    return Geo_BoundaryMesh3D((construct_FEM_Table.(ArrayType, examples))...)
end

F_S_V_SIMPLEX = [[1, 2], [2, 3], [3, 1]] #f = face = face, s = segment to distinguish
F_S_V_CUBE = [[1, 2], [2, 3], [3, 4], [4, 1]]

B_S_V_SIMPLEX = [[1, 2], [2, 3], [3, 1], [1, 4], [2, 4], [3, 4]]
B_S_V_CUBE = [[1, 2], [2, 3], [3, 4], [4, 1], [1, 5], [2, 6],
[3, 7], [4, 8], [5, 6], [6, 7], [7, 8], [8, 5]]

B_F_S_SIMPLEX = [[1, 2, 3], [1, 5, 4], [2, 6, 5], [3, 4, 6]]
B_F_S_CUBE = [[1, 2, 3, 4], [1, 6, 9, 5], [2, 7, 10, 6],
[3, 8, 11, 7], [4, 8, 12, 5], [9, 10, 11, 12]]

"""
    construct_TotalMesh(coors, connections)
    construct_TotalMesh(::Type{ArrayType}, coors, connections)

The function reads (vert, connections) and declares the first order mesh, `FEM_Geometry`.
If not explicitly declared, the defualt ArrayType is
`DEFAULT_ARRAYINFO`.`_type`, which is GPU_DeviceArray if unspecified.

There are 4 concepts in a `FEM_Geometry`:
* Each vertex in `vertices` stores the vertex coordinates.
* Each segment in `segments` stores the vertex IDs connected to this segment.
* Each face in `faces` stores both the vertex IDs and the segment IDs connected to this face.
* Each block in `blocks` stores the vertex IDs,
the segment IDs and the face IDs connected to this block.
If applicable, the IDs are in order, e.g., segment IDs in a face are
clockwise/counter-clockwise (since we can't define an outward normal for a bare face).

The function returns either a `Geo_TotalMesh2D` with attributes
(`vertices`, `segments`, `faces`) or a `Geo_TotalMesh3D`
with attributes (`vertices`, `segments`, `faces`, `blocks`).
"""
construct_TotalMesh(coors::AbstractArray, connections::AbstractArray) =
construct_TotalMesh(DEFAULT_ARRAYINFO._type, coors, connections)
function construct_TotalMesh(::Type{ArrayType}, coors::AbstractArray,
connections::AbstractArray) where {ArrayType}
    coors = FEM_convert(ArrayType, coors)
    connections = FEM_convert(ArrayType, connections)

    dim, _ = size(coors)
    if dim == 2
        ref_geometry = construct_TotalMesh_2D(ArrayType, coors, connections)
    elseif dim == 3
        ref_geometry = construct_TotalMesh_3D(ArrayType, coors, connections)
    else
        error(dim, "Undefined dimension")
    end
end

construct_BoundaryMesh(coors::AbstractArray, connections::AbstractArray) =
construct_BoundaryMesh(DEFAULT_ARRAYINFO._type, coors, connections)
function construct_BoundaryMesh(::Type{ArrayType}, coors::AbstractArray,
connections::AbstractArray) where {ArrayType}
    coors = FEM_convert(ArrayType, coors)
    connections = FEM_convert(ArrayType, connections)
    dim, _ = size(coors)
    vertices_per_block, _ = size(connections)

    if dim == 2 && vertices_per_block == 2
        ref_geometry = construct_BoundaryMesh_2D(ArrayType, coors, connections)
    elseif dim == 3 && vertices_per_block == 3
        ref_geometry = construct_BoundaryMesh_3D(ArrayType, coors, connections)
    else
        error("dim", dim, "vertices_per_block", vertices_per_block, "Undefined mesh type")
    end
end

get_Next_ID(id, size) = id == size ? 1 : (id + 1)
get_Prev_ID(id, size) = id == 1 ? size : (id - 1)

function construct_TotalMesh_2D(::Type{ArrayType},
coors::AbstractArray, connections::AbstractArray) where {ArrayType}
    vertex_per_block, block_number = size(connections)
    if vertex_per_block == 3
        mesh_type = :SIMPLEX
        F_S_V_POS = F_S_V_SIMPLEX
    elseif vertex_per_block == 4
        mesh_type = :CUBE
        F_S_V_POS = F_S_V_CUBE
    else
        error("dim", dim, "vertices_per_block", vertices_per_block, "Undefined mesh type")
    end

    ref_geometry = Geo_TotalMesh2D(ArrayType, mesh_type)
    @Takeout (vertices, segments, faces) FROM ref_geometry
    vIDs = allocate_by_length!(vertices, size(coors, 2))
    vertices.x1[vIDs] .= coors[1, :]
    vertices.x2[vIDs] .= coors[2, :]


    fIDs = allocate_by_length!(faces, block_number)

    faces.vertex_IDs[:, fIDs] .= connections


    seg_dict = dumb_FEM_Dict_Init(ArrayType, FEM_Int)

    for (f_s_pos, s_v_pos) in enumerate(F_S_V_POS)

        s_vIDs = connections[s_v_pos, :]

        rotate_size = length(s_v_pos)


        max_vIDs, cart_pos = findmax(s_vIDs, dims = 1) .|> vec

        max_pos = getindex.(cart_pos, 1)

        last_dim_ids = getindex.(cart_pos, 2)


        next_pos = get_Next_ID.(max_pos, rotate_size)

        next_vIDs = s_vIDs[CartesianIndex.(next_pos, last_dim_ids)]

        dict_keys = I4I30I30_To_UI64.(0, max_vIDs, next_vIDs)


        dict_slots = FEM_Dict_SetID!(seg_dict, dict_keys)

        total_dict_IDs = get_Total_IDs(seg_dict)

        not_allocated = seg_dict.vals[total_dict_IDs] .== 0


        new_sIDs = allocate_by_length!(segments, sum(not_allocated))

        seg_dict.vals[total_dict_IDs[not_allocated]] .= new_sIDs


        local_sIDs = seg_dict.vals[dict_slots]

        faces.segment_IDs[f_s_pos, fIDs] .= local_sIDs


        segments.vertex_IDs[1, local_sIDs] .= max_vIDs

        segments.vertex_IDs[2, local_sIDs] .= next_vIDs

    end

    println("2D geometry constructed with $(length(vIDs)) vertices and $(length(fIDs))
    faces, taking $(report_memory(ref_geometry))")
    return ref_geometry
end

function construct_TotalMesh_3D(::Type{ArrayType}, coors::AbstractArray,
connections::AbstractArray) where {ArrayType}
    vertex_per_block, block_number = size(connections)
    if vertex_per_block == 4
        mesh_type = :SIMPLEX
        vertex_per_face = 3
        B_S_V_POS = B_S_V_SIMPLEX
        B_F_S_POS = B_F_S_SIMPLEX
    elseif vertex_per_block == 8
        mesh_type = :CUBE
        vertex_per_face = 4
        B_S_V_POS = B_S_V_CUBE
        B_F_S_POS = B_F_S_CUBE
    else
        error("dim", dim, "vertices_per_block", vertices_per_block, "Undefined mesh type")
    end

    ref_geometry = Geo_TotalMesh3D(ArrayType, mesh_type)
    @Takeout (vertices, segments, faces, blocks) FROM ref_geometry
    vIDs = allocate_by_length!(vertices, size(coors, 2))
    vertices.x1[vIDs] .= coors[1, :]
    vertices.x2[vIDs] .= coors[2, :]
    vertices.x3[vIDs] .= coors[3, :]

    bIDs = allocate_by_length!(blocks, block_number)
    blocks.vertex_IDs[:, bIDs] .= connections

    seg_dict = dumb_FEM_Dict_Init(ArrayType, FEM_Int)
```

```
for (b_s_pos, s_v_pos) in enumerate(B_S_V_POS)
    s_vIDs = connections[s_v_pos, :]
    rotate_size = length(s_v_pos)

    max_vIDs, cart_pos = findmax(s_vIDs, dims = 1) .|> vec
    max_pos = getindex.(cart_pos, 1)
    last_dim_ids = getindex.(cart_pos, 2)

    next_pos = get_Next_ID.(max_pos, rotate_size)

    next_vIDs = s_vIDs[CartesianIndex.(next_pos, last_dim_ids)]
    dict_keys = I4I30I30_To_UI64.(0, max_vIDs, next_vIDs)

    dict_slots = FEM_Dict_SetID!(seg_dict, dict_keys)
    total_dict_IDs = get_Total_IDs(seg_dict)
    not_allocated = seg_dict.vals[total_dict_IDs] .== 0

    new_sIDs = allocate_by_length!(segments, sum(not_allocated))
    seg_dict.vals[total_dict_IDs[not_allocated]] .= new_sIDs

    local_sIDs = seg_dict.vals[dict_slots]
    blocks.segment_IDs[b_s_pos, bIDs] .= local_sIDs

    segments.vertex_IDs[1, local_sIDs] .= max_vIDs
    segments.vertex_IDs[2, local_sIDs] .= next_vIDs
end

fac_dict = dumb_FEM_Dict_Init(ArrayType, FEM_Int)
for (b_f_pos, f_s_pos) in enumerate(B_F_S_POS)
    f_sIDs = blocks.segment_IDs[f_s_pos, bIDs]
    rotate_size = length(f_s_pos)
```

```
        max_sIDs, cart_pos = findmax(f_sIDs, dims = 1) .|> vec

        max_pos = getindex.(cart_pos, 1)

        last_dim_ids = getindex.(cart_pos, 2)


        prev_pos = get_Prev_ID.(max_pos, rotate_size)

        next_pos = get_Next_ID.(max_pos, rotate_size)

        is_forward = f_sIDs[CartesianIndex.(next_pos, last_dim_ids)] .>=

        f_sIDs[CartesianIndex.(prev_pos, last_dim_ids)]

        next_pos[.~ is_forward] .= prev_pos[.~ is_forward]


        next_sIDs = f_sIDs[CartesianIndex.(next_pos, last_dim_ids)]

        dict_keys = I4I30I30_To_UI64.(0, max_sIDs, next_sIDs)


        dict_slots = FEM_Dict_SetID!(fac_dict, dict_keys)

        total_dict_IDs = get_Total_IDs(fac_dict)

        not_allocated = fac_dict.vals[total_dict_IDs] .== 0


        new_fIDs = allocate_by_length!(faces, sum(not_allocated))

        fac_dict.vals[total_dict_IDs[not_allocated]] .= new_fIDs


        local_fIDs = fac_dict.vals[dict_slots]

        blocks.face_IDs[b_f_pos, bIDs] .= local_fIDs


        faces.segment_IDs[1, local_fIDs] .= max_sIDs

        last_sIDs = max_sIDs

        for i = 1:1:vertex_per_face

            is_first_vID =

            (segments.vertex_IDs[1, last_sIDs] .== segments.vertex_IDs[1, next_sIDs]) .|

            (segments.vertex_IDs[1, last_sIDs] .== segments.vertex_IDs[2, next_sIDs])


            faces.vertex_IDs[i, local_fIDs[is_first_vID]] .=

            segments.vertex_IDs[1, last_sIDs[is_first_vID]]
            faces.vertex_IDs[i, local_fIDs[.~ is_first_vID]] .=
            segments.vertex_IDs[2, last_sIDs[.~ is_first_vID]]

            (i == vertex_per_face) && break
            faces.segment_IDs[i + 1, local_fIDs] .= next_sIDs

            last_sIDs = next_sIDs
            last_pos = next_pos

            prev_pos = get_Prev_ID.(last_pos, rotate_size)
            next_pos = get_Next_ID.(last_pos, rotate_size)
            next_pos[.~ is_forward] .= prev_pos[.~ is_forward]
            next_sIDs = f_sIDs[CartesianIndex.(next_pos, last_dim_ids)]
        end
    end
    println("3D geometry constructed with $(length(vIDs)) vertices and
    $(length(bIDs)) blocks, taking $(report_memory(ref_geometry))")
    return ref_geometry
end

function construct_BoundaryMesh_2D(::Type{ArrayType},
coors::AbstractArray, connections::AbstractArray) where {ArrayType}
    ref_geometry = Geo_BoundaryMesh2D(ArrayType)
    @Takeout (vertices, segments) FROM ref_geometry
    vIDs = allocate_by_length!(vertices, size(coors, 2))
    vertices.x1[vIDs] .= coors[1, :]
    vertices.x2[vIDs] .= coors[2, :]

    sIDs = allocate_by_length!(segments, size(connections, 2))
    segments.vertex_IDs[:, sIDs] .= connections
    return ref_geometry
end
```

```
function construct_BoundaryMesh_3D(::Type{ArrayType}, coors::AbstractArray,
connections::AbstractArray) where {ArrayType}
    ref_geometry = Geo_BoundaryMesh3D(ArrayType)
    @Takeout (vertices, segments, faces) FROM ref_geometry
    vIDs = allocate_by_length!(vertices, size(coors, 2))
    vertices.x1[vIDs] .= coors[1, :]
    vertices.x2[vIDs] .= coors[2, :]
    vertices.x3[vIDs] .= coors[3, :]

    fIDs = allocate_by_length!(faces, size(connections, 2))
    faces.vertex_IDs[:, fIDs] .= connections

    seg_dict = dumb_FEM_Dict_Init(ArrayType, FEM_Int)
    for (f_s_pos, s_v_pos) in enumerate(F_S_V_POS[mesh_type])
        s_vIDs = connections[s_v_pos, :]
        rotate_size = length(s_v_pos)

        max_vIDs, cart_pos = findmax(s_vIDs, dims = 1) .|> vec
        max_pos = getindex.(cart_pos, 1)
        last_dim_ids = getindex.(cart_pos, 2)

        next_pos = get_Next_ID.(max_pos, rotate_size)
        next_vIDs = s_vIDs[CartesianIndex.(next_pos, last_dim_ids)]
        dict_keys = I4I30I30_To_UI64.(0, max_vIDs, next_vIDs)

        dict_slots = FEM_Dict_SetID!(seg_dict, dict_keys)
        total_dict_IDs = get_Total_IDs(seg_dict)
        not_allocated = seg_dict.vals[total_dict_IDs] .== 0

        new_sIDs = allocate_by_length!(segments, sum(not_allocated))
        seg_dict.vals[total_dict_IDs[not_allocated]] .= new_sIDs
        local_sIDs = seg_dict.vals[dict_slots]
        faces.segment_IDs[f_s_pos, fIDs] .= local_sIDs

        segments.vertex_IDs[1, local_sIDs] .= max_vIDs
        segments.vertex_IDs[2, local_sIDs] .= next_vIDs
    end
    return ref_geometry
end

"""
    get_BoundaryMesh(total_mesh::Geo_TotalMesh2D)
    get_BoundaryMesh(total_mesh::Geo_TotalMesh3D)

Helper function to collect all the segment/face IDs which can be marked as boundary.
"""
function get_BoundaryMesh(total_mesh::Geo_TotalMesh2D{ArrayType}) where {ArrayType}
    f_el_num = FEM_zeros(ArrayType, FEM_Int, length(total_mesh.segments.is_occupied))
    elIDs = findall(total_mesh.faces.is_occupied)
    inc_Num!(f_el_num, 1, vec(total_mesh.faces.segment_IDs[:, elIDs]))
    return findall(f_el_num .== 1)
end

function get_BoundaryMesh(total_mesh::Geo_TotalMesh3D{ArrayType}) where {ArrayType}
    f_el_num = FEM_zeros(ArrayType, FEM_Int, length(total_mesh.faces.is_occupied))
    elIDs = findall(total_mesh.blocks.is_occupied)
    inc_Num!(f_el_num, 1, vec(total_mesh.blocks.face_IDs[:, elIDs]))
    return findall(f_el_num .== 1)
end

struct Classical_Element_Structure
    vertex_cp_ids::Array{FEM_Int}
    segment_cp_ids::Array{FEM_Int}
    face_cp_ids::Array{FEM_Int}
    block_cp_ids::Array{FEM_Int}

    segment_cp_pos::Array{FEM_Float}
    face_cp_pos::Array{FEM_Float}
    block_cp_pos::Array{FEM_Float}

    segment_start_vertex::Array{FEM_Int}
    face_start_segments::Array{FEM_Int}
end

mutable struct Classical_Discretization{ArrayType} <: FEM_Spatial_Discretization{ArrayType}
    element_attributes::Dict{Symbol, Any}
    #topology
    element_structure::Classical_Element_Structure
    #interpolation
    itp_func_num::FEM_Int
    itp_funcs::Vector{Polynomial}
    #domain integral
    itg_func_num::FEM_Int
    itg_weight::ArrayType

    ref_itp_vals::ArrayType
    #boundary integral
    bdy_itg_func_num::FEM_Int
    bdy_itg_weights::Vector{ArrayType}
    bdy_tangent_directions::Vector{ArrayType}
    #itg_ID, local vector, tangent ID (2D only 1, 3D 2)...

    bdy_ref_itp_vals::Vector{ArrayType}
    #itg_ID, bdy_cp_ID, diff mode: 1, partialX, partial Y, ...
```

```
end


initialize_Classical_Element(dim::Integer, shape::Symbol, itp_order::Integer,
max_sd_order::Integer, itg_order::Integer; itp_type::Symbol = :Lagrange) =
initialize_Classical_Element(DEFAULT_ARRAYINFO._type, dim, shape, itp_order,
max_sd_order, itg_order; itp_type = :itp_type)
function initialize_Classical_Element(::Type{ArrayType}, dim::Integer, shape::Symbol,
itp_order::Integer, max_sd_order::Integer, itg_order::Integer;
itp_type::Symbol = :Lagrange) where {ArrayType}
    if shape == :CUBE
        if itp_type == :Lagrange
            if dim == 2
                element_structure = init_Structure_Cube2D_Lagrange(itp_order)
            elseif dim == 3
                element_structure = init_Structure_Cube3D_Lagrange(itp_order)
            else
                error("Wrong dimension for structure")
            end
            itp_funcs = init_Interpolation_Cube_Lagrange(itp_order, dim)
        elseif itp_type == :Serendipity
            if dim == 2
                element_structure = init_Structure_Cube2D_Serendipity(itp_order)
            elseif dim == 3
                element_structure = init_Structure_Cube3D_Serendipity(itp_order)
            else
                error("Wrong dimension for structure")
            end
            itp_funcs = init_Interpolation_Cube_Serendipity(itp_order, dim)
        end
        itg_pos, itg_weight = init_Domain_Integration_Cube_Gauss(itg_order, dim)
        bdy_itg_pos, bdy_itg_weights, bdy_tangent_directions =
        init_Boundary_Integration_Cube_Gauss(itg_order, dim)
    elseif shape == :SIMPLEX
        if dim == 2
            element_structure = init_Structure_Triangle_Lagrange(itp_order)
            itg_pos, itg_weight = init_Domain_Integration_Triangle_Gauss(itg_order)
            bdy_itg_pos, bdy_itg_weights, bdy_tangent_directions =
            init_Boundary_Integration_Triangle_Gauss(itg_order)
        elseif dim == 3
            element_structure = init_Structure_Tetrahedron_Lagrange(itp_order)
            itg_pos, itg_weight = init_Domain_Integration_Tetrahedron_Gauss(itg_order)
            bdy_itg_pos, bdy_itg_weights, bdy_tangent_directions =
            init_Boundary_Integration_Tetrahedron_Gauss(itg_order)
        else
            error("Wrong dimension for structure")
        end
        itp_funcs = init_Interpolation_Simplex_Lagrange(itp_order, dim)
    end
    element_attributes = Dict(:dim => dim, :shape => shape, :itp_type => itp_type, :itp_order =>
itp_order)
    itp_func_num, itg_func_num, bdy_itg_func_num =
    (itp_funcs, itg_weight, bdy_itg_weights[1]) .|> length
    itg_weight, bdy_itg_weights, bdy_tangent_directions = (FEM_convert(ArrayType, itg_weight),
    FEM_convert.(ArrayType, bdy_itg_weights), FEM_convert.(ArrayType, bdy_tangent_directions))

    ref_itp_vals = FEM_convert(ArrayType, evaluate_Itp_Funcs(itp_funcs, max_sd_order, itg_pos))
    bdy_ref_itp_vals =
    FEM_convert.(ArrayType, evaluate_Itp_Funcs.(Ref(itp_funcs), max_sd_order, bdy_itg_pos))
    return @Construct Classical_Discretization{ArrayType}
end

function evaluate_Itp_Funcs(itp_funcs::Vector{Polynomial{dim}},
max_sd_order::Integer, itg_pos::Vector) where dim
    itp_func_num = length(itp_funcs)
    itg_func_num = size(itg_pos, 1)

    grad_size = fill(max_sd_order + 1, dim)
    itp_gradient_funcs = fill(Polynomial{dim}[], grad_size...)
    ref_itp_vals = zeros(FEM_Float, itg_func_num, itp_func_num, grad_size...)

    for diff_orders in Iterators.product([0:max_sd_order for i = 1:dim]...)
        diff_ids = diff_orders .+ 1
        itp_gradient_func_batch = derivative.(itp_funcs, Ref(diff_orders))
        itp_gradient_funcs[diff_ids...] = itp_gradient_func_batch
        ref_itp_vals[:, :, diff_ids...] .= [evaluate_Polynomial(this_itp_func, pos)
        for pos in itg_pos, this_itp_func in itp_gradient_func_batch]
    end
    return ref_itp_vals
end

function init_Structure_Cube2D_Lagrange(itp_order::Integer)
    cp_per_dim = itp_order + 1

    cp_per_vertex = 1
    vertex_cp_ids = [1 cp_per_dim cp_per_dim ^ 2 cp_per_dim * (cp_per_dim - 1) + 1]

    cp_per_segment = cp_per_dim - 2
    segment_cp_ids = hcat([i + 1 for i = 1:cp_per_segment],
    [cp_per_dim * (i + 1) for i = 1:cp_per_segment],
    [cp_per_dim * (cp_per_dim - 1) + i + 1 for i = 1:cp_per_segment],
    [cp_per_dim * i + 1 for i = 1:cp_per_segment])
    segment_cp_pos = hcat([[1. - i / itp_order, i / itp_order] for i = 1:cp_per_segment]...)
    segment_start_vertex = [1, 2, 4, 1]

    cp_per_face = cp_per_segment ^ 2
    face_cp_ids = [j * cp_per_dim + i + 1
    for j = 1:cp_per_segment for i = 1:cp_per_segment]
    face_cp_pos = hcat([[
                        (1. - i / itp_order) * (1. - j / itp_order),
                        (     i / itp_order) * (1. - j / itp_order),
                        (     i / itp_order) * (     j / itp_order),
                        (1. - i / itp_order) * (     j / itp_order)
                      ] for j = 1:cp_per_segment for i = 1:cp_per_segment]...)
    face_start_segments = zeros(FEM_Int, 0, 1)

    cp_per_block = 0
    block_cp_ids = zeros(FEM_Int, cp_per_block, 0)
    block_cp_pos = zeros(FEM_Float, 0, cp_per_block)

    return @Construct Classical_Element_Structure
end

function init_Structure_Cube3D_Lagrange(itp_order::Integer)
    cp_per_dim = itp_order + 1

    cp_per_vertex = 1
    v_cp_ids_1D = [1 cp_per_dim cp_per_dim ^ 2 cp_per_dim * (cp_per_dim - 1) + 1]
    vertex_cp_ids = hcat(v_cp_ids_1D, v_cp_ids_1D .+ (cp_per_dim - 1) * cp_per_dim ^ 2)

    cp_per_segment = cp_per_dim - 2
    segment_cp_ids = hcat([i + 1 for i = 1:cp_per_segment],
    [cp_per_dim * (i + 1) for i = 1:cp_per_segment],
    [cp_per_dim * (cp_per_dim - 1) + i + 1 for i = 1:cp_per_segment],
    [cp_per_dim * i + 1 for i = 1:cp_per_segment],

    [cp_per_dim ^ 2 * i + 1 for i = 1:cp_per_segment],
    [cp_per_dim ^ 2 * i + cp_per_dim for i = 1:cp_per_segment],
    [cp_per_dim ^ 2 * i + cp_per_dim ^ 2 for i = 1:cp_per_segment],
    [cp_per_dim ^ 2 * i + cp_per_dim * (cp_per_dim - 1) + 1 for i = 1:cp_per_segment],

    [(cp_per_dim - 1) * cp_per_dim ^ 2 + i + 1 for i = 1:cp_per_segment],
    [(cp_per_dim - 1) * cp_per_dim ^ 2 + cp_per_dim * (i + 1) for i = 1:cp_per_segment],
    [(cp_per_dim - 1) * cp_per_dim ^ 2 + cp_per_dim * (cp_per_dim - 1) + i + 1
    for i = 1:cp_per_segment],
    [(cp_per_dim - 1) * cp_per_dim ^ 2 + cp_per_dim * i + 1 for i = 1:cp_per_segment]
    )

    segment_cp_pos = hcat([[1. - i / itp_order, i / itp_order] for i = 1:cp_per_segment]...)
    segment_start_vertex = [1, 2, 4, 1,
    1, 2, 3, 4,
    5, 6, 8, 5]

    cp_per_face = cp_per_segment ^ 2
    face_cp_ids = hcat([j * cp_per_dim + i + 1
    for j = 1:cp_per_segment for i = 1:cp_per_segment],
    [j * cp_per_dim ^ 2 + i + 1 for j = 1:cp_per_segment for i = 1:cp_per_segment],
    [j * cp_per_dim ^ 2 + i * cp_per_dim + cp_per_dim
    for j = 1:cp_per_segment for i = 1:cp_per_segment],
    [j * cp_per_dim ^ 2 + i + 1 + (cp_per_dim - 1) * cp_per_dim
    for j = 1:cp_per_segment for i = 1:cp_per_segment],
    [j * cp_per_dim ^ 2 + i * cp_per_dim + 1 for j = 1:cp_per_segment for i = 1:cp_per_segment],
    [j * cp_per_dim + i + 1 + (cp_per_dim - 1) * cp_per_dim ^ 2
    for j = 1:cp_per_segment for i = 1:cp_per_segment]
    )
    face_cp_pos = hcat([[
                          (1. - i / itp_order) * (1. - j / itp_order),
                          (     i / itp_order) * (1. - j / itp_order),
                          (     i / itp_order) * (     j / itp_order),
                          (1. - i / itp_order) * (     j / itp_order)
                        ] for j = 1:cp_per_segment for i = 1:cp_per_segment]...)
```

```
    face_start_segments = hcat([4, 1], [5, 1], [6, 2], [8, 3], [5, 4], [12, 9])

    cp_per_block = cp_per_segment ^ 3

    block_cp_ids = [k * cp_per_dim ^ 2 + j * cp_per_dim + i + 1 for k =
    1:cp_per_segment for j = 1:cp_per_segment for i = 1:cp_per_segment]
    block_cp_pos = hcat([[
        (1. - i / itp_order) * (1. - j / itp_order) * (1. - k / itp_order),
        (     i / itp_order) * (1. - j / itp_order) * (1. - k / itp_order),
        (     i / itp_order) * (     j / itp_order) * (1. - k / itp_order),
        (1. - i / itp_order) * (     j / itp_order) * (1. - k / itp_order),
        (1. - i / itp_order) * (1. - j / itp_order) * (     k / itp_order),
        (     i / itp_order) * (1. - j / itp_order) * (     k / itp_order),
        (     i / itp_order) * (     j / itp_order) * (     k / itp_order),
        (1. - i / itp_order) * (     j / itp_order) * (     k / itp_order)
    ] for k = 1:cp_per_segment for j = 1:cp_per_segment for i = 1:cp_per_segment]...)
    return @Construct Classical_Element_Structure
end

function init_Structure_Triangle_Lagrange(itp_order::Integer)
    cp_per_dim = itp_order + 1

    cp_per_vertex = 1
    vertex_cp_ids = [1 cp_per_dim FEM_Int(cp_per_dim * (cp_per_dim + 1) / 2)]

    cp_per_segment = (cp_per_dim - 2)
    segment_cp_ids = hcat([i + 1 for i = 1:cp_per_segment],
    [FEM_Int((2 * cp_per_dim - i) * (i + 1) / 2) for i = 1:cp_per_segment],
    [FEM_Int((2 * cp_per_dim - i + 1) * i / 2 + 1) for i = 1:cp_per_segment])

    segment_cp_pos = hcat([[1. - i / itp_order, i / itp_order] for i = 1:cp_per_segment]...)
    segment_start_vertex = [1, 2, 1]

    cp_per_face = FEM_Int((cp_per_segment - 1) * cp_per_segment / 2)
    this_id = 0
    face_cp_ids = zeros(FEM_Int, cp_per_face)
    face_cp_pos = zeros(FEM_Float, 3, cp_per_face)
    for j = 1:(cp_per_segment - 1)
        for i = 1:(cp_per_segment - j)
            this_id += 1
            k = itp_order - j - i
            face_cp_ids[this_id] = (2 * cp_per_dim - i + 1) * i / 2 + i + 1
            face_cp_pos[:, this_id] .= (k, i, j) ./ itp_order
        end
    end
    face_start_segments = zeros(FEM_Int, 0, 1)

    cp_per_block = 0
    block_cp_ids = zeros(FEM_Int, cp_per_block, 0)
    block_cp_pos = zeros(FEM_Float, 0, cp_per_block)

    return @Construct Classical_Element_Structure
end

function init_Structure_Tetrahedron_Lagrange(itp_order::Integer)
    cp_per_dim = itp_order + 1

    cp_per_vertex = 1
    vertex_cp_ids = [1 cp_per_dim FEM_Int(cp_per_dim *
    (cp_per_dim + 1) / 2) FEM_Int(cp_per_dim * (cp_per_dim + 1) * (cp_per_dim + 2) / 6)]

    cp_per_segment = (cp_per_dim - 2)
    segment_cp_ids = zeros(FEM_Int, cp_per_segment, 6)
```

```
segment_cp_ids[:, 1] .= [i + 1 for i = 1:cp_per_segment]
segment_cp_ids[:, 2] .= [FEM_Int((2 * cp_per_dim - i) * (i + 1) / 2)
for i = 1:cp_per_segment]
segment_cp_ids[:, 3] .= [FEM_Int((2 * cp_per_dim - i + 1) * i / 2 + 1)
for i = 1:cp_per_segment] #4,5,6 allocate incrementally

segment_cp_pos = hcat([[1. - i / itp_order, i / itp_order] for i = 1:cp_per_segment]...)
segment_start_vertex = [1, 2, 1, 1, 2, 3]

cp_per_face = FEM_Int((cp_per_segment - 1) * cp_per_segment / 2)
cp_per_block = FEM_Int((cp_per_segment - 2) * (cp_per_segment - 1) * cp_per_segment / 6)

this_id = 0
face_cp_ids = zeros(FEM_Int, cp_per_face, 4)
face_cp_pos = zeros(FEM_Float, 3, cp_per_face)
block_cp_ids = zeros(FEM_Int, cp_per_block)
block_cp_pos = zeros(FEM_Float, 4, cp_per_block)

for j = 1:cp_per_segment
    for i = 1:cp_per_segment
        k = itp_order - j - i
        k <= 0 && continue

        this_id += 1
        face_cp_ids[this_id, 1] = (2 * cp_per_dim - i + 1) * i / 2 + i + 1
        face_cp_pos[:, this_id] .= (k, i, j) ./ itp_order
    end
end

last_final_cp_id = FEM_Int(cp_per_dim * (cp_per_dim + 1) / 2)
face_last_id = 0
block_last_id = 0
```

```
    for k = 1:cp_per_segment
        segment_cp_ids[k, 4] = last_final_cp_id + 1
        current_pos_in_layer = cp_per_dim - k
        segment_cp_ids[k, 5] = last_final_cp_id + current_pos_in_layer

        face_cp_ids[(face_last_id + 1):(face_last_id + (cp_per_segment - k)), 2] .=
        (segment_cp_ids[k, 4] + 1):(segment_cp_ids[k, 5] - 1)

        for j = 1:(cp_per_segment - k)
            face_cp_ids[face_last_id + j, 4] = last_final_cp_id + current_pos_in_layer + 1
            current_pos_in_layer += cp_per_dim - k - j
            face_cp_ids[face_last_id + j, 3] = last_final_cp_id + current_pos_in_layer

            (cp_per_segment - k - j) <= 0 && continue
            block_range = (block_last_id + 1):(block_last_id + (cp_per_segment - k - j))
            block_cp_ids[block_range] .=
            (face_cp_ids[face_last_id + j, 4] + 1):(face_cp_ids[face_last_id + j, 3] - 1)
            block_cp_pos[1, block_range] .= (cp_per_segment - k - j):1
            block_cp_pos[2, block_range] .= 1:(cp_per_segment - k - j)
            block_cp_pos[3, block_range] .= j
            block_cp_pos[4, block_range] .= k

            block_last_id += (cp_per_segment - k - j)
        end
        last_final_cp_id += current_pos_in_layer + 1
        segment_cp_ids[k, 6] = last_final_cp_id
    end
    block_cp_pos ./= itp_order
    face_start_segments = hcat([3, 1], [4, 1], [5, 2], [4, 3])

    return @Construct Classical_Element_Structure
end
```

```
function init_Structure_Cube2D_Serendipity(itp_order::Integer)
    cp_per_dim = itp_order + 1

    cp_per_vertex = 1
    vertex_cp_ids = [1 2 4 3]

    cp_per_segment = cp_per_dim - 2
    segment_orders = [0, 3, 1, 2]
    cp_per_segment = cp_per_dim - 2
    segment_cp_ids = [segment_orders[j] * cp_per_segment + 4 + i for i = 1:cp_per_segment, j =
1:4]

    segment_cp_pos = hcat([[1. - i / itp_order, i / itp_order] for i = 1:cp_per_segment]...)
    segment_start_vertex = [1, 2, 4, 1] # Note top/left is reverse

    cp_per_face = 0
    face_cp_ids = zeros(FEM_Int, cp_per_face, 1)
    face_cp_pos = zeros(FEM_Float, 0, cp_per_face)
    face_start_segments = zeros(FEM_Int, 0, 0)

    cp_per_block = 0
    block_cp_ids = zeros(FEM_Int, cp_per_block, 0)
    block_cp_pos = zeros(FEM_Float, 0, cp_per_block)

    return @Construct Classical_Element_Structure
end

function init_Structure_Cube3D_Serendipity(itp_order::Integer)
    cp_per_dim = itp_order + 1

    cp_per_vertex = 1
    vertex_cp_ids = [1 2 4 3 5 6 8 7]


    cp_per_segment = cp_per_dim - 2

    segment_orders = [0, 5, 1, 4, 8, 9, 11, 10, 2, 7, 3, 6]

    segment_cp_ids = [segment_orders[j] * cp_per_segment + 8 + i for i = 1:cp_per_segment, j =
1:12]

    segment_cp_pos = hcat([[1. - i / itp_order, i / itp_order] for i = 1:cp_per_segment]...)

    segment_start_vertex = [1, 2, 4, 1,

    1, 2, 3, 4,

    5, 6, 8, 5]


    cp_per_face = 0

    face_cp_ids = zeros(FEM_Int, cp_per_face, 6)

    face_cp_pos = zeros(FEM_Float, 0, cp_per_face)

    face_start_segments = zeros(FEM_Int, 0, 0)


    cp_per_block = 0

    block_cp_ids = zeros(FEM_Int, cp_per_block, 1)

    block_cp_pos = zeros(FEM_Float, 0, cp_per_block)

    return @Construct Classical_Element_Structure

end

# 1  2  3  4

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

function init_Interpolation_Lagrange_1D(interp_order::Integer)

    interp_funcs_1D = Polynomial{1}[]

    interp_pos_1D = [i/interp_order for i = 0:interp_order]

    for this_interp_id = 1:(interp_order + 1)

        this_interp_pos = interp_pos_1D[this_interp_id]

        this_interp =  Polynomial(1) + 1

        for this_term_id = 1:(interp_order + 1)

            this_term_id == this_interp_id && continue
            this_term_pos = interp_pos_1D[this_term_id]

            denominator = this_interp_pos - this_term_pos

            #this term is just x

            factors = [1., -this_term_pos]./denominator

            orders = [(1,), (0,)]

            this_term = Polynomial(factors, orders)

            this_interp *= this_term

        end

        push!(interp_funcs_1D, this_interp)

    end

    return interp_funcs_1D

end


#13 14 15 16

# 9 10 11 12

# 5  6  7  8

# 1  2  3  4

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

function init_Interpolation_Cube_Lagrange(interp_order::Integer, dim::Integer)

    interp_funcs_1D = init_Interpolation_Lagrange_1D(interp_order)

    template_funcs = [substitute_Polynomial.(interp_funcs_1D, 1,

    Ref(Polynomial(1., basis_Tup(i, dim)))) for i = 1:dim]


    result_itp_funcs = Polynomial{dim}[]

    for interp_ids in Iterators.product([1:(interp_order + 1) for i = 1:dim]...)

        push!(result_itp_funcs, prod([template_funcs[i][interp_ids[i]] for i = 1:dim]))

    end

    return result_itp_funcs

end


# 10

# 8  9
# 5  6  7
# 1  2  3  4
############################
function init_Interpolation_Simplex_Lagrange(interp_order::Integer, dim::Integer)
    interp_funcs_1D = [Polynomial(1., (0,)),
    [substitute_Polynomial(init_Interpolation_Lagrange_1D(this_order)[end],
    1, Polynomial(interp_order / this_order, (1,))) for this_order = 1:interp_order]...]

    volumetric_coors = [Polynomial(1., basis_Tup(i, dim)) for i = 1:dim]
    push!(volumetric_coors, Polynomial([1., [-1. for i = 1:dim]...],
    [const_Tup(0, dim), [basis_Tup(i, dim) for i = 1:dim]...]))

    template_funcs = [substitute_Polynomial.(interp_funcs_1D, 1,
    Ref(volumetric_coors[i])) for i = 1:(dim + 1)]

    result_itp_funcs = Polynomial{dim}[]
    for interp_pos in Iterators.product([0:interp_order for i = 1:dim]...)
        last_dim_pos = interp_order - sum(interp_pos)
        last_dim_pos < 0 && continue
        push!(result_itp_funcs, prod([template_funcs[i][
            interp_pos[i] + 1] for i = 1:dim]) * template_funcs[dim + 1][last_dim_pos + 1])
    end
    return result_itp_funcs
end

# 3  7  8  4
# 10       12
# 9        11
# 1  5  6  2
############################
function init_Interpolation_Cube_Serendipity(interp_order::Integer, dim::Integer)
    xs = collect_Basis(dim)
```

```
result_itp_funcs = Polynomial{dim}[]
# The corner of serendipity is defined ad-hoc each itp order by nature
if interp_order <= 2
    for coors in Iterators.product([0:1 for i = 1:dim]...)
        this_itp_func = prod((1 .- coors) .- xs)
        for i = 1:(interp_order - 1)
            s = 1 .- 2 .* coors
            this_itp_func *= dot(s, coors) + i / interp_order - sum(s .* xs)
        end
        this_itp_func /= evaluate_Polynomial(this_itp_func, coors) #normalization
        push!(result_itp_funcs, this_itp_func)
    end
elseif interp_order == 3
    shifted_xs = xs .- 0.5
    for coors in Iterators.product([0:1 for i = 1:dim]...)
        this_itp_func = prod((1 .- coors) .- xs) * (sum(shifted_xs .^ 2) -
        ((1 / 6) ^ 2 + (dim - 1) * (1 / 2) ^ 2))

        this_itp_func /= evaluate_Polynomial(this_itp_func, coors)
        push!(result_itp_funcs, this_itp_func)
    end
else
    error("Undefined serendipity order")
end

for edge_direction = 1:dim
    minor_dims = [i for i = 1:dim if i != edge_direction]
    for minor_coors in Iterators.product([0:1 for i = 1:(dim - 1)]...)
        base_itp_func = prod((1 .- minor_coors) .- xs[minor_dims])
        for itp_pos = 1:(interp_order - 1)
            this_itp_func = prod(Ref(xs[edge_direction]) .- [i /
            interp_order for i = 0:interp_order if i != itp_pos]) * base_itp_func
```

```
                this_coor = [itp_pos / interp_order for i = 1:dim]
                this_coor[minor_dims] .= minor_coors

                this_itp_func /= evaluate_Polynomial(this_itp_func, Tuple(this_coor))
                push!(result_itp_funcs, this_itp_func)
            end
        end
    end
    return result_itp_funcs
end

function init_Interpolation_Hermite_1D(interp_order::Integer)
    interp_funcs_1D = Polynomial{1}[]

    T = zeros(CPU_Float, (2 * interp_order), (2 * interp_order))
    rstart, rend = 0, 1
    for this_order = 1:interp_order
        for this_column = this_order:(2 * interp_order)
            T[this_order, this_column] = factorial(this_column - 1)/
            factorial(this_column - this_order) * rstart ^ (this_column - this_order)
            T[(interp_order + this_order), this_column] =
            factorial(this_column - 1)/factorial(this_column -
            this_order) * rend ^ (this_column - this_order)
        end
    end

    T_inv = T^(-1)
    for i = 1:(2 * interp_order)
        this_interp = Polynomial(vec(T_inv[:,i]), [(i,)
        for i = 0:(2 * interp_order - 1)]) |> check_Clear
        push!(interp_funcs_1D, this_interp)
    end
    return interp_funcs_1D
end


shift_gauss_point(x) = x ./ 2.0 .+ 0.5
shift_gauss_weight(x) = x ./ 2.0
const GAUSS_POINT_POS_1D_ORIGINAL = ((0.0,), (-1.0/sqrt(3.0), 1.0/sqrt(3.0)),
(-sqrt(3.0/5.0), 0.0, sqrt(3.0/5.0)),
(-sqrt(3.0/7.0 + 2.0/7.0 * sqrt(6.0/5.0)), -sqrt(3.0/7.0 - 2.0/7.0 * sqrt(6.0/5.0)),
sqrt(3.0/7.0 - 2.0/7.0 * sqrt(6.0/5.0)), sqrt(3.0/7.0 + 2.0/7.0 * sqrt(6.0/5.0))))

const GAUSS_POINT_WEIGHT_1D_ORIGINAL = ((2.0,), (1.0, 1.0), (5.0/9.0, 8.0/9.0, 5.0/9.0),
                        ((18.0 - sqrt(30.0))/36.0, (18.0 + sqrt(30.0))/36.0,
                         (18.0 + sqrt(30.0))/36.0, (18.0 - sqrt(30.0))/36.0))

const GAUSS_POINT_POS_1D_SHIFTED = shift_gauss_point.(GAUSS_POINT_POS_1D_ORIGINAL)
const GAUSS_POINT_WEIGHT_1D_SHIFTED = shift_gauss_weight.(GAUSS_POINT_WEIGHT_1D_ORIGINAL)

function init_Domain_Integration_Cube_Gauss(itg_order::Integer, dim::Integer)
    gauss_order = (itg_order + 1) / 2 |> ceil |> FEM_Int
    itg_pos = [getindex.(Ref(GAUSS_POINT_POS_1D_SHIFTED[gauss_order]), this_id)
    for this_id in Iterators.product([1:gauss_order for i=1:dim]...)] |> vec
    itg_weight = [prod(getindex.(Ref(GAUSS_POINT_WEIGHT_1D_SHIFTED[gauss_order]), this_id))
    for this_id in Iterators.product([1:gauss_order for i=1:dim]...)] |> vec
    return itg_pos, itg_weight
end

function init_Boundary_Integration_Cube_Gauss(itg_order::Integer, dim::Integer)
    gauss_order = (itg_order + 1) / 2 |> ceil |> FEM_Int
    itg_pos, base_itg_weight = init_Domain_Integration_Cube_Gauss(itg_order, dim - 1)
    if dim == 2
        face_ids = [4 2; 1 3]
    elseif dim == 3
```

```
        face_ids = [5 3; 2 4; 1 6]
    else
        error("Undefined dim")
    end

    itg_num, face_num = (itg_pos, face_ids) .|> length
    bdy_itg_pos = [fill(const_Tup(0., dim), itg_num) for i = 1:face_num]
    bdy_tangent_directions = [zeros(FEM_Float, itg_num, dim, dim - 1) for i = 1:face_num]

    for normal_dim = 1:dim
        tangent_dim = [(i + normal_dim - 1) % dim + 1 for i = 1:(dim - 1)]
        for outward_direction = 0:1 #left, right
            this_face_id = face_ids[normal_dim, outward_direction + 1]
            raw_tangent = [j == tangent_dim[i] ? 1 : 0 for k = 1:itg_num, j = 1:dim, i = 1:(dim -
1)]
            # println(raw_tangent)
            if dim == 2
                ((outward_direction + normal_dim) == 2) || (raw_tangent .*= -1)
            elseif dim == 3
                (outward_direction == 0) && (raw_tangent[:, :, 1] .*= -1)
            end
            bdy_tangent_directions[this_face_id] .= raw_tangent
            for i = 1:itg_num
                this_pos = zeros(FEM_Float, dim)
                this_pos[tangent_dim] .= itg_pos[i]
                this_pos[normal_dim] = outward_direction
                bdy_itg_pos[this_face_id][i] = Tuple(this_pos)
            end
        end
    end

    return bdy_itg_pos, [copy(base_itg_weight)
    for i = 1:length(face_ids)], bdy_tangent_directions
end

##

#Order 5, 6, 8

const GAUSS_POINT_POS_TRIANGLE = (((0.101286507323456338800987361915123 83,),
```

```
    for i = 1:length(face_ids)], bdy_tangent_directions

end

##

#Order 5, 6, 8

const GAUSS_POINT_POS_TRIANGLE = (((0.10128650732345633880098736191512383,),

(0.47014206410511508977044120951344760,), ()),

((0.06308901449150222834033160287081916,), (0.24928674517091042129163855310701908,),

(0.05314504984481694735324967163139815, 0.31035245103378440541660773395655215)),

((), (0.17056930775176020662229350149146450,), (0.05054722831703097545842355059659895,),

(0.45929258829272315602881551449416932,), (0.26311282963463811342178578628464359,

0.00839477740995760533721383453929445)))


const GAUSS_POINT_WEIGHT_TRIANGLE = ((0.12593918054482715259568394550018133,

0.13239415278850618073764938783315200, 9.0/40.0),

(0.05084490637020681692093680910686898, 0.11678627572637936602528961138557944,

0.08285107561837357519355345642044245),

(0.14431560767778716825109111048906462, 0.10321737053471825028179155029212903,

0.03245849762319808031092592834178060, 0.09509163426728462479389610438858432,

0.02723031417443499426484469007390892))


# (a,) is perm31 (-a,) is 22

const GAUSS_POINT_POS_TETRAHEDRON = (((0.31088591926330060979734573376345783,),

(0.09273525031089122640232391373703061,), (-0.04550370412564964949188052627933943,)),

((0.21460287125915202928883921938628499,), (0.04067395853461135311557944895641006,),

(0.32233789014227551034399447076249213,), (0.06366100187501752529923552760572698,

0.60300566479164914136743113906093969)),

((0.03967542307038990126507132953938949,), (0.31448780069809631378416056269714830,),

(0.10198669306270330000000000000000000,), (0.18420369694919151227594641734890918,),

(-0.06343628775453989240514123870189827,),

 (0.02169016206772800480266248262493018, 0.71993192203946593588943495335273478),

 (0.20448008063679571424133557487274534, 0.58057719012880922417539817139062041)))
```

```
const GAUSS_POINT_WEIGHT_TETRAHEDRON = ((0.11268792571801585079918565233328633,
0.07349304311636194954371020548632750, 0.04254602077708146643806942812025744),
(0.03992275025816749209969062755747998, 0.01007721105532064294801323744593686,
 0.05535718154365472209515327785372602, 27.0/560.0),
(0.00639714777990232132145142033517302, 0.04019044802096617248816115847981783,
0.02430797550477032117486910877192260, 0.05485889241369744046692412399039144,
0.03571961223409918246495096899661762, 0.00718319069785253940945110521980376,
0.01637218194531911754093813975611913))

function _init_Integration_Triangle_Gauss(itg_order::Integer)
    if itg_order <= 5
        id = 1
    elseif itg_order <= 6
        id = 2
    elseif itg_order <= 8
        id = 3
    else
        error("Wrong integral order")
    end

    itg_pos = Tuple{Vararg{FEM_Float, 3}}[]
    itg_weight = FEM_Float[]

    for (pos, weight) in zip(GAUSS_POINT_POS_TRIANGLE[id], GAUSS_POINT_WEIGHT_TRIANGLE[id])
        if length(pos) == 0
            push!(itg_pos, const_Tup(1/3, 3))
            push!(itg_weight, weight)
        elseif length(pos) == 1
            a = pos[1]
            append!(itg_pos, [basis_Tup(i, 3, (1 - 2 * a), a) for i = 1:3])
            append!(itg_weight, [weight for i = 1:3])
        elseif length(pos) == 2
```

```
            c = 1. - sum(pos)
            src_pos = (pos..., c)
            for (i, j) in Iterators.product([1:3 for tmp = 1:2]...)
                (i == j) && continue
                k = 6 - i - j
                push!(itg_pos, src_pos[[i, j, k]])
                push!(itg_weight, weight)
            end
        end
    end

    return itg_pos, itg_weight
end

function init_Domain_Integration_Triangle_Gauss(itg_order::Integer)
    itg_pos, itg_weight = _init_Integration_Triangle_Gauss(itg_order)
    return getindex.(itg_pos, Ref([2, 3])), itg_weight ./ 2.
end

function init_Boundary_Integration_Triangle_Gauss(itg_order::Integer)
    gauss_order = (itg_order + 1) / 2 |> ceil |> FEM_Int
    itg_pos, base_itg_weight = init_Domain_Integration_Cube_Gauss(gauss_order, 1)
    bdy_itg_weights = [copy(base_itg_weight) for i = 1:3]
    bdy_itg_weights[2] .*= sqrt(2)

    itg_num, face_num, dim = (length(itg_pos), 3, 2)
    bdy_itg_pos = [fill(const_Tup(0., dim), itg_num) for i = 1:face_num]
    bdy_tangent_directions = [zeros(FEM_Float, itg_num, dim, dim - 1) for i = 1:face_num]

    for i = 1:itg_num
        a = itg_pos[i][1]
        bdy_itg_pos[1][i] = (1. - a) .* (0, 0) .+ a .* (1, 0)
        bdy_itg_pos[2][i] = (1. - a) .* (1, 0) .+ a .* (0, 1)
        bdy_itg_pos[3][i] = (1. - a) .* (0, 1) .+ a .* (0, 0)
    end
    bdy_tangent_directions[1][:, :, 1] .= [1. 0.]
    bdy_tangent_directions[2][:, :, 1] .= [-1. 1.] ./ sqrt(2)
    bdy_tangent_directions[3][:, :, 1] .= [0. -1.]
    return bdy_itg_pos, bdy_itg_weights, bdy_tangent_directions
end

function _init_Integration_Tetrahedron_Gauss(itg_order::Integer)
    if itg_order <= 5
        id = 1
    elseif itg_order <= 6
        id = 2
    elseif itg_order <= 8
        id = 3
    else
        error("Wrong integral order")
    end

    itg_pos = Tuple{Vararg{FEM_Float, 4}}[]
    itg_weight = FEM_Float[]

    for (pos, weight) in zip(GAUSS_POINT_POS_TETRAHEDRON[id], GAUSS_POINT_WEIGHT_TETRAHEDRON[id])
        if length(pos) == 0
            push!(itg_pos, const_Tup(1/4, 4))
            push!(itg_weight, weight)
        elseif length(pos) == 1
            a = pos[1]
            if a >= 0
                append!(itg_pos, [basis_Tup(i, 4, (1 - 3 * a), a) for i = 1:4])
                append!(itg_weight, [weight for i = 1:4])
            else
                b = -a
                for (i, j) in Iterators.product([1:4 for tmp = 1:2]...)
                    (i >= j) && continue
                    src_pos = fill(b, 4)
                    src_pos[[i,j]] .= 0.5 - b
                    push!(itg_pos, Tuple(src_pos))
                    push!(itg_weight, weight)
                end
            end
        elseif length(pos) == 2
            a, b = pos
            c = 1 - 2 * a - b
            for (i, j) in Iterators.product([1:4 for tmp = 1:2]...)
                (i == j) && continue
                src_pos = fill(a, 4)
                src_pos[i] = b
                src_pos[j] = c
                push!(itg_pos, Tuple(src_pos))
                push!(itg_weight, weight)
            end
        elseif length(pos) == 3
            d = 1. - sum(pos)
            src_pos = (pos..., d)
            for (i, j, k) in Iterators.product([1:4 for tmp = 1:3]...)
                ((i == j) || (i == k) || (j == k)) && continue
                l = 10 - i - j - k
                push!(itg_pos, src_pos[[i, j, k, l]])
                push!(itg_weight, weight)
            end
        end
        end
        return itg_pos, itg_weight
end

function init_Domain_Integration_Tetrahedron_Gauss(itg_order::Integer)
    itg_pos, itg_weight = _init_Integration_Tetrahedron_Gauss(itg_order)
    return getindex.(itg_pos, Ref([2, 3, 4])), itg_weight ./ 6
end

function init_Boundary_Integration_Tetrahedron_Gauss(itg_order::Integer)
    itg_pos, base_itg_weight = _init_Integration_Triangle_Gauss(itg_order)
    bdy_itg_weights = [copy(base_itg_weight) * 0.5 for i = 1:4]
    bdy_itg_weights[3] .*= sqrt(3)

    itg_num, face_num, dim = (length(itg_pos), 4, 3)
    bdy_itg_pos = [fill(const_Tup(0., dim), itg_num) for i = 1:face_num]
    bdy_tangent_directions = [zeros(FEM_Float, itg_num, dim, dim - 1) for i = 1:face_num]

    for i = 1:itg_num
        a, b, c = itg_pos[i]

        bdy_itg_pos[1][i] = a .* (0, 0, 0) .+ b .* (1, 0, 0) .+ c .* (0, 1, 0)
        bdy_itg_pos[2][i] = a .* (0, 0, 0) .+ b .* (1, 0, 0) .+ c .* (0, 0, 1)
        bdy_itg_pos[3][i] = a .* (0, 0, 1) .+ b .* (1, 0, 0) .+ c .* (0, 1, 0)
        bdy_itg_pos[4][i] = a .* (0, 0, 0) .+ b .* (0, 1, 0) .+ c .* (0, 0, 1)
    end

    bdy_tangent_directions[1][:, :, 1] .= [-1. 0. 0.]
    bdy_tangent_directions[1][:, :, 2] .= [ 0. 1. 0.]

    bdy_tangent_directions[2][:, :, 1] .= [0. 0. -1.]
    bdy_tangent_directions[2][:, :, 2] .= [1. 0.  0.]
    bdy_tangent_directions[3][:, :, 1] .= [-1. 1. 0.] ./ sqrt(2)
    bdy_tangent_directions[3][:, :, 2] .= [-1. -1. 2.] ./ sqrt(6) #must be orthogonal

    bdy_tangent_directions[4][:, :, 1] .= [0. -1. 0.]
    bdy_tangent_directions[4][:, :, 2] .= [0.  0. 1.]
    return bdy_itg_pos, bdy_itg_weights, bdy_tangent_directions
end

mutable struct Basic_ControlPoint2D
    x1::FEM_Float
    x2::FEM_Float

    global_cpID::FEM_Int #cp_IDs of FIRST local variable, the rest variables need to be shifted
end

mutable struct Basic_ControlPoint3D
    x1::FEM_Float
    x2::FEM_Float
    x3::FEM_Float

    global_cpID::FEM_Int #cp_IDs of FIRST local variable, the rest variables need to be shifted
    vID::FEM_Int
end

mutable struct Basic_Facet
    tangent_directions::Array{FEM_Float, 3} #normal_direction
    normal_directions::Array{FEM_Float, 2}

    jacobian::Array{FEM_Float, 3}
    inverse_jacobian::Array{FEM_Float, 3}
    el_dets::Vector{FEM_Float}
    bdy_dets::Vector{FEM_Float}

    element_ID::FEM_Int
    element_eindex::FEM_Int
    integral_vals::Array{FEM_Float} #diff vals separate
    integral_weights::Vector{FEM_Float}
end

mutable struct Basic_Split
    tangent_directions::Array{FEM_Float, 3} #normal_direction
    normal_directions::Array{FEM_Float, 2}

    jacobian::Array{FEM_Float, 3}
    inverse_jacobian::Array{FEM_Float, 3}
    el_dets::Vector{FEM_Float}
    bdy_dets::Vector{FEM_Float}

    element_ID::FEM_Int
    element_eindex::FEM_Int
    integral_vals::Array{FEM_Float} #diff vals separate
    integral_weights::Vector{FEM_Float}

    outer_element_ID::FEM_Int
    outer_element_eindex::FEM_Int
    outer_integral_vals::Array{FEM_Float} #diff vals separate
    outer_integral_weights::Vector{FEM_Float}
end

mutable struct Basic_Element
    #common part about element
    controlpoint_IDs::Vector{FEM_Int}
    global_cpIDs::Vector{FEM_Int}
    sparse_IDs_by_el::Array{FEM_Int, 2}

    integral_vals::Array{FEM_Float} #diff vals separate
    integral_weights::Vector{FEM_Float}
    #for basic mesh
    # facet_IDs::Vector{FEM_Int}
    jacobian::Array{FEM_Float, 3}
    inverse_jacobian::Array{FEM_Float, 3}
    dets::Vector{FEM_Float}
end

mutable struct Basic_WP_Mesh{ArrayType} <: FEM_WP_Mesh{ArrayType}
    controlpoints::FEM_Table{ArrayType}
    facets::FEM_Table{ArrayType} #boundaries
    elements::FEM_Table{ArrayType}

    bg_fIDs::Dict{FEM_Int, AbstractArray{FEM_Int, 1}}
    variable_size::FEM_Int
end

"""
    mesh_Classical(wp_IDs; shape::Symbol, itp_type::Symbol = :Lagrange,
    itp_order::Integer, itg_order::Integer, fem_domain::FEM_Domain)

The function generates the mesh of order `itp_order`, interpolation
type `itp_type` = `:Lagrange`/`:Serendipity` with gaussian quadrature of
order `itg_order` on `fem_domain`.`workpieces`[`wp_IDs`].
The dimension and mesh type (`:CUBE`/`:SIMPLEX`) will
follow the first order mesh of each `WorkPiece`.
"""
function mesh_Classical(wp_IDs; shape::Symbol, itp_type::Symbol = :Lagrange,
itp_order::Integer, itg_order::Integer, fem_domain::FEM_Domain{ArrayType}) where {ArrayType}
    dim = fem_domain.dim
    for wp in fem_domain.workpieces[wp_IDs]
        this_space = wp.element_space = initialize_Classical_Element(
            ArrayType, dim, shape, itp_order, wp.max_sd_order, itg_order; itp_type = itp_type)

        controlpoints = declare_Basic_ControlPoint(dim, wp)
        facets = declare_Basic_Facet(dim, this_space)
        elements = declare_Basic_Element(dim, this_space)

        bg_fIDs = Dict{FEM_Int, AbstractArray{FEM_Int, 1}}()
        variable_size = FEM_Int(0)
        wp.mesh = @Construct Basic_WP_Mesh{ArrayType}

        if dim == 2
            allocate_Basic_WP_Mesh_2D(wp, this_space)

            evaled_max_sd_order = length(BASE_KERNELS_2D)
            if wp.max_sd_order > evaled_max_sd_order
                append!(BASE_KERNELS_2D, [Core.eval(@__MODULE__,
                gen_Kernel_Itpval(i, 2)) for i = evaled_max_sd_order:(wp.max_sd_order)])
            end
        elseif dim == 3
            allocate_Basic_WP_Mesh_3D(wp, this_space)

            evaled_max_sd_order = length(BASE_KERNELS_3D)
            if wp.max_sd_order > evaled_max_sd_order
                append!(BASE_KERNELS_3D, [Core.eval(@__MODULE__,
                gen_Kernel_Itpval(i, 3)) for i = evaled_max_sd_order:(wp.max_sd_order)])
            end
        else
            error("Undefined dimension")
        end
        println("Allocate controlpoints with $(report_memory(controlpoints)),
        facets with $(report_memory(facets)), and elements with $(report_memory(elements))")
    end
end

function declare_Basic_ControlPoint(dim::Integer, wp::WorkPiece{ArrayType}) where {ArrayType}
    @Takeout (local_innervar_infos, controlpoint_extervars) FROM wp.local_assembly

    global_cpID, element_num = (0, 0) .|> FEM_Int
    if dim == 2
        x1, x2 = (0., 0.) .|> FEM_Float
        controlpoints = @Construct Basic_ControlPoint2D
    elseif dim == 3
        x1, x2, x3 = (0., 0., 0.) .|> FEM_Float
        vID = FEM_Int(0)
        controlpoints = @Construct Basic_ControlPoint3D
    end
    local_innervars = getindex.(local_innervar_infos, 1)
    return construct_FEM_Table(ArrayType, controlpoints,
    Symbol[local_innervars..., controlpoint_extervars...] .=> FEM_Float(0.))
end

function declare_Basic_Facet(dim::Integer, this_space::
Classical_Discretization{ArrayType}) where {ArrayType}
    @Takeout (bdy_itg_func_num, bdy_ref_itp_vals) FROM this_space
    tangent_directions = zeros(FEM_Float, bdy_itg_func_num, dim, dim - 1)
    normal_directions = zeros(FEM_Float, bdy_itg_func_num, dim)

    element_ID, element_eindex = (0, 0) .|> FEM_Int
    integral_vals = zeros(FEM_Float, size(this_space.bdy_ref_itp_vals[1]))
    jacobian = zeros(FEM_Float, dim, dim, bdy_itg_func_num)
    inverse_jacobian = zeros(FEM_Float, dim, dim, bdy_itg_func_num)
    el_dets = zeros(FEM_Float, bdy_itg_func_num)
    bdy_dets = zeros(FEM_Float, bdy_itg_func_num)
    integral_weights = zeros(FEM_Float, bdy_itg_func_num)

    outer_element_ID, outer_element_eindex = (0, 0) .|> FEM_Int

    boundaries = @Construct Basic_Facet
    return construct_FEM_Table(ArrayType, boundaries, Symbol[] .=> Array[])
end

function declare_Basic_Element(dim::Integer,
this_space::Classical_Discretization{ArrayType}) where {ArrayType}
    @Takeout (itp_func_num, itg_func_num, ref_itp_vals) FROM this_space

    controlpoint_IDs, global_cpIDs = zeros(FEM_Int, itp_func_num), zeros(FEM_Int, itp_func_num)
    sparse_IDs_by_el = zeros(FEM_Int, itp_func_num, itp_func_num)

    integral_vals = zeros(FEM_Float, size(ref_itp_vals))
    integral_weights  = zeros(FEM_Float, itg_func_num)

    # facet_IDs = zeros(FEM_Int, length(this_space.bdy_ref_itp_vals))
    jacobian = zeros(FEM_Float, dim, dim, itg_func_num)
    inverse_jacobian = zeros(FEM_Float, dim, dim, itg_func_num)
    dets = zeros(FEM_Float, itg_func_num)

    elements = @Construct Basic_Element
    return construct_FEM_Table(ArrayType, elements, Symbol[] .=> Array[])
end

"""
    update_Mesh(dim::Integer, wp::WorkPiece, this_space::Classical_Discretization)
```

```
This function updates Jacobians and interpolation values.
"""
function update_Mesh(dim::Integer, wp::WorkPiece, this_space::Classical_Discretization)
    if dim == 2
        itpval_kernel = BASE_KERNELS_2D[wp.max_sd_order]
        update_BasicElements_2D(wp.mesh, this_space, itpval_kernel)
        update_BasicBoundary_2D(wp.mesh, this_space, itpval_kernel)
    elseif dim == 3
        itpval_kernel = BASE_KERNELS_3D[wp.max_sd_order]
        update_BasicElements_3D(wp.mesh, this_space, itpval_kernel)
        update_BasicBoundary_3D(wp.mesh, this_space, itpval_kernel)
    end
end

function allocate_Basic_WP_Mesh_2D(wp::WorkPiece{ArrayType},
this_space::Classical_Discretization{ArrayType}) where {ArrayType}
    @Takeout (vertices, segments, faces) FROM wp.ref_geometry
    @Takeout (controlpoints, facets, elements, bg_fIDs) FROM
    wp.mesh #Note here facets refers to segments
    @Takeout (vertex_cp_ids, segment_cp_ids, face_cp_ids, segment_cp_pos,
    face_cp_pos, segment_start_vertex) FROM this_space.element_structure

    face_IDs = findall(faces.is_occupied)
    elIDs = allocate_by_length!(elements, length(face_IDs))
    prod(face_IDs .== elIDs) || error("face_IDs should be the same as elIDs")

    vIDs = findall(vertices.is_occupied)
    cp_per_vertex, vertex_per_element = size(vertex_cp_ids)
    v_cpIDs = allocate_by_length!(controlpoints, cp_per_vertex * length(vIDs))
    for i = 1:cp_per_vertex
        batch_cpIDs = v_cpIDs[i:cp_per_vertex:end]
        controlpoints.x1[batch_cpIDs] .= vertices.x1[vIDs]
        controlpoints.x2[batch_cpIDs] .= vertices.x2[vIDs]
    end
    for j = 1:vertex_per_element #separate with above for consistency
        for i = 1:cp_per_vertex
            elements.controlpoint_IDs[vertex_cp_ids[i, j], elIDs] .=
            v_cpIDs[(faces.vertex_IDs[j, face_IDs] .- 1) .* cp_per_vertex .+ i]
        end
    end

    sIDs = findall(segments.is_occupied)
    segment_facet_IDs = FEM_zeros(ArrayType, FEM_Int, length(sIDs))
    for (bg_index, boundary) in enumerate(wp.physics.boundarys)
        bdyIDs = allocate_by_length!(facets, length(boundary))
        bg_fIDs[bg_index] = bdyIDs
        segment_facet_IDs[boundary] .= bdyIDs
    end

    cp_per_segment, segment_per_element = size(segment_cp_ids)
    for j = 1:segment_per_element
        specify_eindex(FEM_Int(j), segment_facet_IDs, faces.segment_IDs,
        facets.element_ID, facets.element_eindex, facets.outer_element_ID,
        facets.outer_element_eindex, elIDs)
    end

    if cp_per_segment > 0
        s_cpIDs = allocate_by_length!(controlpoints, cp_per_segment * length(sIDs))
        segment_cp_pos = FEM_convert(ArrayType, segment_cp_pos)
        for i = 1:cp_per_segment
            batch_cpIDs = s_cpIDs[i:cp_per_segment:end]
            controlpoints.x1[batch_cpIDs] .=
            sum(segment_cp_pos .* vertices.x1[segments.vertex_IDs[:, sIDs]], dims = 1)[1, :]
            controlpoints.x2[batch_cpIDs] .=
            sum(segment_cp_pos .* vertices.x2[segments.vertex_IDs[:, sIDs]], dims = 1)[1, :]
        end
        for j = 1:segment_per_element
            is_aligned = segments.vertex_IDs[1, faces.segment_IDs[j, face_IDs]] .==
            faces.vertex_IDs[segment_start_vertex[j], face_IDs]
            a_elIDs = elIDs[is_aligned]
            n_elIDs = elIDs[.~ is_aligned]
            for i = 1:cp_per_segment
                elements.controlpoint_IDs[segment_cp_ids[i, j], a_elIDs] .=
                s_cpIDs[(faces.segment_IDs[j, a_elIDs] .- 1) .* cp_per_segment .+ i]
                elements.controlpoint_IDs[segment_cp_ids[i, j], n_elIDs] .=
                s_cpIDs[ faces.segment_IDs[j, n_elIDs] .* cp_per_segment .- (i - 1)]
            end
        end
    end

    cp_per_face = length(face_cp_ids) #2d only has one face
    if cp_per_face > 0
        f_cpIDs = allocate_by_length!(controlpoints, cp_per_face * length(face_IDs))
        face_cp_pos = FEM_convert(ArrayType, face_cp_pos)
        for i = 1:cp_per_face
            batch_cpIDs = f_cpIDs[i:cp_per_face:end]
            controlpoints.x1[batch_cpIDs] .= sum(face_cp_pos .*
            vertices.x1[faces.vertex_IDs[:, face_IDs]], dims = 1)[1, :]
            controlpoints.x2[batch_cpIDs] .= sum(face_cp_pos .*
            vertices.x2[faces.vertex_IDs[:, face_IDs]], dims = 1)[1, :]

            elements.controlpoint_IDs[face_cp_ids[i], elIDs] .= batch_cpIDs
        end
    end
end
```

```
function allocate_Basic_WP_Mesh_3D(wp::WorkPiece{ArrayType},
this_space::Classical_Discretization{ArrayType}) where {ArrayType}
    @Takeout (vertices, segments, faces, blocks) FROM wp.ref_geometry
    @Takeout (controlpoints, facets, elements, bg_fIDs) FROM
    wp.mesh #Note here facets refers to segments
    @Takeout (vertex_cp_ids, segment_cp_ids, face_cp_ids, block_cp_ids,
    segment_cp_pos, face_cp_pos, block_cp_pos,
    segment_start_vertex, face_start_segments) FROM this_space.element_structure

    block_IDs = findall(blocks.is_occupied)
    elIDs = allocate_by_length!(elements, length(block_IDs))
    prod(block_IDs .== elIDs) || error("block_IDs should be the same as elIDs")

    vIDs = findall(vertices.is_occupied)
    cp_per_vertex, vertex_per_element = size(vertex_cp_ids)
    v_cpIDs = allocate_by_length!(controlpoints, cp_per_vertex * length(vIDs))
    for i = 1:cp_per_vertex
        batch_cpIDs = v_cpIDs[i:cp_per_vertex:end]
        controlpoints.x1[batch_cpIDs] .= vertices.x1[vIDs]
        controlpoints.x2[batch_cpIDs] .= vertices.x2[vIDs]
        controlpoints.x3[batch_cpIDs] .= vertices.x3[vIDs]
    end
    for j = 1:vertex_per_element #separate with above for consistency
        for i = 1:cp_per_vertex
            elements.controlpoint_IDs[vertex_cp_ids[i, j], elIDs] .=
            v_cpIDs[(blocks.vertex_IDs[j, block_IDs] .- 1) .* cp_per_vertex .+ i]
        end
    end

    sIDs = findall(segments.is_occupied)
    cp_per_segment, segment_per_element = size(segment_cp_ids)
```

```
if cp_per_segment > 0

    s_cpIDs = allocate_by_length!(controlpoints, cp_per_segment * length(sIDs))

    segment_cp_pos = FEM_convert(ArrayType, segment_cp_pos)

    for i = 1:cp_per_segment

        batch_cpIDs = s_cpIDs[i:cp_per_segment:end]

        controlpoints.x1[batch_cpIDs] .= sum(segment_cp_pos .*

        vertices.x1[segments.vertex_IDs[:, sIDs]], dims = 1)[1, :]

        controlpoints.x2[batch_cpIDs] .= sum(segment_cp_pos .*

        vertices.x2[segments.vertex_IDs[:, sIDs]], dims = 1)[1, :]

        controlpoints.x3[batch_cpIDs] .= sum(segment_cp_pos .*

        vertices.x3[segments.vertex_IDs[:, sIDs]], dims = 1)[1, :]

    end


    for j = 1:segment_per_element

        is_aligned = segments.vertex_IDs[1, blocks.segment_IDs[j, block_IDs]]

        .== blocks.vertex_IDs[segment_start_vertex[j], block_IDs]


        a_elIDs = elIDs[is_aligned]

        n_elIDs = elIDs[.~ is_aligned]

        for i = 1:cp_per_segment

            elements.controlpoint_IDs[segment_cp_ids[i, j], a_elIDs] .=

            s_cpIDs[(blocks.segment_IDs[j, a_elIDs] .- 1) .* cp_per_segment .+ i]

            elements.controlpoint_IDs[segment_cp_ids[i, j], n_elIDs] .=

            s_cpIDs[ blocks.segment_IDs[j, n_elIDs] .* cp_per_segment .- (i - 1)]

        end

    end

end


face_IDs = findall(faces.is_occupied)

face_facet_IDs = FEM_zeros(ArrayType, FEM_Int, length(face_IDs))

for (bg_index, boundary) in enumerate(wp.physics.boundarys)

    bdyIDs = allocate_by_length!(facets, length(boundary))
```

```
        bg_fIDs[bg_index] = bdyIDs
        face_facet_IDs[boundary] .= bdyIDs
    end

    cp_per_face, face_per_element = size(face_cp_ids)
    for j = 1:face_per_element
        specify_eindex(FEM_Int(j), face_facet_IDs, blocks.face_IDs, facets.element_ID,
        facets.element_eindex, facets.outer_element_ID, facets.outer_element_eindex, elIDs)
    end

    if cp_per_face > 0
        if cp_per_face > 1
            error("TO DO face control point matching")
        end
        f_cpIDs = allocate_by_length!(controlpoints, cp_per_face * length(face_IDs))
        face_cp_pos = FEM_convert(ArrayType, face_cp_pos)
        for i = 1:cp_per_face
        batch_cpIDs = f_cpIDs[i:cp_per_face:end]
        controlpoints.x1[batch_cpIDs] .= sum(face_cp_pos .*
        vertices.x1[faces.vertex_IDs[:, face_IDs]], dims = 1)[1, :]
        controlpoints.x2[batch_cpIDs] .= sum(face_cp_pos .*
        vertices.x2[faces.vertex_IDs[:, face_IDs]], dims = 1)[1, :]
        controlpoints.x3[batch_cpIDs] .= sum(face_cp_pos .*
        vertices.x3[faces.vertex_IDs[:, face_IDs]], dims = 1)[1, :]
        end

        for j = 1:face_per_element
            elements.controlpoint_IDs[face_cp_ids[1, j], elIDs] .=
            f_cpIDs[blocks.face_IDs[j, elIDs]]
        end
    end
    cp_per_block = length(block_cp_ids) #2d only has one face
    if cp_per_block > 0
    b_cpIDs = allocate_by_length!(controlpoints, cp_per_block * length(block_IDs))
    block_cp_pos = FEM_convert(ArrayType, block_cp_pos)
    for i = 1:cp_per_block
    batch_cpIDs = b_cpIDs[i:cp_per_block:end]
    controlpoints.x1[batch_cpIDs] .= sum(block_cp_pos .*
    vertices.x1[blocks.vertex_IDs[:, block_IDs]], dims = 1)[1, :]
    controlpoints.x2[batch_cpIDs] .= sum(block_cp_pos .*
    vertices.x2[blocks.vertex_IDs[:, block_IDs]], dims = 1)[1, :]
    controlpoints.x3[batch_cpIDs] .= sum(block_cp_pos .*
    vertices.x3[blocks.vertex_IDs[:, block_IDs]], dims = 1)[1, :]

    elements.controlpoint_IDs[block_cp_ids[i], elIDs] .= batch_cpIDs
    end
    end
end

@Dumb_GPU_Kernel specify_eindex(f_pos, face_facet_mapping::Array, cell_face_IDs::Array,
f_el_ID::Array, eindex::Array, outer_f_el_ID::Array, outer_eindex::Array, elIDs::Array) begin
    this_elID = elIDs[thread_idx]

    this_facet_ID = face_facet_mapping[cell_face_IDs[f_pos, this_elID]]
    this_facet_ID == 0 && return

    old_elID = CUDA.atomic_cas!(pointer(f_el_ID) + sizeof(eltype(f_el_ID)) *
    (this_facet_ID - 1), Int32(0), Int32(this_elID)) #Note dict key cannot be 0
    if old_elID != 0
        outer_f_el_ID[this_facet_ID] = this_elID
        outer_eindex[this_facet_ID] = f_pos
    else
        eindex[this_facet_ID] = f_pos
    end
end


gen_Kernel_Name(prefix::String, max_sd_order::Integer, dim::Integer) =
Symbol(string(prefix, max_sd_order, "_", dim, "D"))
function gen_BasicDomain_Funcs(dim::Integer)
    ID_for_no_diff = fill(1, dim)
    jac_block = :(begin
        X_IDs = basis_Tup(X_dim, $dim, 2, 1)
    end)
    for this_dim = 1:dim
        x_sym = Symbol("x$(this_dim)")
        push!(jac_block.args, :(jacobian[$this_dim, X_dim, :, elIDs] .=
        ref_itp_vals[:, :, X_IDs...] * controlpoints.$x_sym[cpIDs]))
    end

    invJac_kernel = Symbol("inv_Jac_$(dim)D")
    kernel_libs = Symbol("BASE_KERNELS_$(dim)D")
    func = Symbol("update_BasicElements_$(dim)D")
    ex = :(function ($func)(mesh::FEM_WP_Mesh,
    this_space::Classical_Discretization, itpval_kernel::Function)
        @Takeout (controlpoints, elements) FROM mesh
        @Takeout (controlpoint_IDs, jacobian, inverse_jacobian, dets,
        integral_vals, integral_weights, is_occupied) FROM elements
        @Takeout (itp_func_num, itg_func_num, itg_weight, ref_itp_vals) FROM this_space
        elIDs = findall(elements.is_occupied)
        cpIDs = elements.controlpoint_IDs[:, elIDs]
        for X_dim = 1:$dim
            $jac_block
        end

        integral_vals[:, :, $(ID_for_no_diff...), elIDs] .=
        ref_itp_vals[:, :, $(ID_for_no_diff...)]

        $invJac_kernel(jacobian, dets, inverse_jacobian, elIDs)
        itpval_kernel(integral_vals, ref_itp_vals, inverse_jacobian, elIDs)

        integral_weights .= (itg_weight .* dets)
    end)
    return ex
end

function gen_BasicBoundary_Funcs(dim::Integer)
    ID_for_no_diff = fill(1, dim)
    jac_block = :(begin
        X_IDs = basis_Tup(X_dim, $dim, 2, 1)
    end)
    for this_dim = 1:dim
        x_sym = Symbol("x$(this_dim)")
        push!(jac_block.args, :(jacobian[$this_dim, X_dim, :, facet_IDs] .=
        bdy_ref_itp_vals[eindex][:, :, X_IDs...] * controlpoints.$x_sym[cpIDs]))
    end

    invJac_kernel = Symbol("inv_Jac_$(dim)D")
    tangent_kernel = Symbol("update_Basic_Tangent_$(dim)D")
    normal_kernel = Symbol(string("update_Basic_Normal_", dim, "D"))
    kernel_libs = Symbol("BASE_KERNELS_$(dim)D")
    func = Symbol("update_BasicBoundary_$(dim)D")
    ex = :(function ($func)(mesh::FEM_WP_Mesh, this_space::Classical_Discretization,
    itpval_kernel::Function)
        @Takeout (controlpoints, facets, elements) FROM mesh
        @Takeout (tangent_directions, normal_directions, jacobian,
                  inverse_jacobian, el_dets, bdy_dets,
                  element_ID, element_eindex, integral_vals, integral_weights) FROM facets
        @Takeout (itp_func_num, bdy_itg_func_num, bdy_itg_weights,
                  bdy_tangent_directions, bdy_ref_itp_vals) FROM this_space

        for eindex = 1:length(bdy_ref_itp_vals)
            facet_IDs = findall(facets.is_occupied .& (element_eindex .== eindex))
            # facet_IDs = findall(facets.is_occupied .&
            (element_eindex .== eindex) .& (facets.outer_element_ID .== 0))
            isempty(facet_IDs) && continue

            cpIDs = elements.controlpoint_IDs[:, element_ID[facet_IDs]]
            for X_dim = 1:$dim
                $jac_block
            end
            integral_vals[:, :, ($ID_for_no_diff)..., facet_IDs] .=
            bdy_ref_itp_vals[eindex][:, :, ($ID_for_no_diff)...]
            $invJac_kernel(jacobian, el_dets, inverse_jacobian, facet_IDs)
            $tangent_kernel(jacobian, tangent_directions,
            bdy_tangent_directions[eindex], facet_IDs)
            $normal_kernel(normal_directions, tangent_directions, bdy_dets, facet_IDs)
            itpval_kernel(integral_vals, bdy_ref_itp_vals[eindex], inverse_jacobian, facet_IDs)

            integral_weights[:, facet_IDs] .= bdy_itg_weights[eindex] .* bdy_dets[:, facet_IDs]
        end
    end)
    return ex
end

@Dumb_GPU_Kernel inv_Jac_2D(jacobian::Array, dets::Array,
inverse_jacobian::Array, IDs::Array) begin
    this_ID = IDs[thread_idx]
    for itg_id = 1:size(dets, 1)
        dets[itg_id, this_ID] =
        jacobian[1, 1, itg_id, this_ID] * jacobian[2, 2, itg_id, this_ID] -
        jacobian[1, 2, itg_id, this_ID] * jacobian[2, 1, itg_id, this_ID]

        inverse_jacobian[1, 1, itg_id, this_ID] =
        jacobian[2, 2, itg_id, this_ID] / dets[itg_id, this_ID]
        inverse_jacobian[1, 2, itg_id, this_ID] =
      - jacobian[1, 2, itg_id, this_ID] / dets[itg_id, this_ID]
        inverse_jacobian[2, 1, itg_id, this_ID] =
      - jacobian[2, 1, itg_id, this_ID] / dets[itg_id, this_ID]
        inverse_jacobian[2, 2, itg_id, this_ID] =
        jacobian[1, 1, itg_id, this_ID] / dets[itg_id, this_ID]
    end
end

@Dumb_GPU_Kernel inv_Jac_3D(jacobian::Array, dets::Array,
inverse_jacobian::Array, IDs::Array) begin
    this_ID = IDs[thread_idx]
    for itg_id = 1:size(dets, 1)
        dets[itg_id, this_ID] =
        jacobian[1, 1, itg_id, this_ID] *
        jacobian[2, 2, itg_id, this_ID] * jacobian[3, 3, itg_id, this_ID] -
        jacobian[1, 1, itg_id, this_ID] *
        jacobian[2, 3, itg_id, this_ID] * jacobian[3, 2, itg_id, this_ID] -
        jacobian[1, 2, itg_id, this_ID] *
        jacobian[2, 1, itg_id, this_ID] * jacobian[3, 3, itg_id, this_ID] +
        jacobian[1, 2, itg_id, this_ID] *
        jacobian[2, 3, itg_id, this_ID] * jacobian[3, 1, itg_id, this_ID] +
        jacobian[1, 3, itg_id, this_ID] *
        jacobian[2, 1, itg_id, this_ID] * jacobian[3, 2, itg_id, this_ID] -
        jacobian[1, 3, itg_id, this_ID] *
        jacobian[2, 2, itg_id, this_ID] * jacobian[3, 1, itg_id, this_ID]
        inverse_jacobian[1, 1, itg_id, this_ID] =
        (jacobian[2, 2, itg_id, this_ID] * jacobian[3, 3, itg_id, this_ID] -
        jacobian[2, 3, itg_id, this_ID] * jacobian[3, 2, itg_id, this_ID]) /
        dets[itg_id, this_ID]
        inverse_jacobian[1, 2, itg_id, this_ID] =
        (jacobian[1, 3, itg_id, this_ID] * jacobian[3, 2, itg_id, this_ID] -
        jacobian[1, 2, itg_id, this_ID] * jacobian[3, 3, itg_id, this_ID]) /
        dets[itg_id, this_ID]
        inverse_jacobian[1, 3, itg_id, this_ID] = (
            jacobian[1, 2, itg_id, this_ID] * jacobian[2, 3, itg_id, this_ID] -
        jacobian[2, 2, itg_id, this_ID] * jacobian[1, 3, itg_id, this_ID]) /
        dets[itg_id, this_ID]

        inverse_jacobian[2, 1, itg_id, this_ID] =
        (jacobian[2, 3, itg_id, this_ID] * jacobian[3, 1, itg_id, this_ID] -
        jacobian[3, 3, itg_id, this_ID] * jacobian[2, 1, itg_id, this_ID]) /
        dets[itg_id, this_ID]
        inverse_jacobian[2, 2, itg_id, this_ID] =
        (jacobian[1, 1, itg_id, this_ID] * jacobian[3, 3, itg_id, this_ID] -
        jacobian[1, 3, itg_id, this_ID] * jacobian[3, 1, itg_id, this_ID]) /
        dets[itg_id, this_ID]
        inverse_jacobian[2, 3, itg_id, this_ID] =
        (jacobian[1, 3, itg_id, this_ID] * jacobian[2, 1, itg_id, this_ID] -
        jacobian[1, 1, itg_id, this_ID] * jacobian[2, 3, itg_id, this_ID]) /
        dets[itg_id, this_ID]

        inverse_jacobian[3, 1, itg_id, this_ID] =
        (jacobian[2, 1, itg_id, this_ID] * jacobian[3, 2, itg_id, this_ID] -
        jacobian[2, 2, itg_id, this_ID] * jacobian[3, 1, itg_id, this_ID]) /
        dets[itg_id, this_ID]
        inverse_jacobian[3, 2, itg_id, this_ID] =
        (jacobian[1, 2, itg_id, this_ID] * jacobian[3, 1, itg_id, this_ID] -
        jacobian[3, 2, itg_id, this_ID] * jacobian[1, 1, itg_id, this_ID]) /
        dets[itg_id, this_ID]
        inverse_jacobian[3, 3, itg_id, this_ID] =
        (jacobian[1, 1, itg_id, this_ID] * jacobian[2, 2, itg_id, this_ID] -
        jacobian[2, 1, itg_id, this_ID] * jacobian[1, 2, itg_id, this_ID]) /
        dets[itg_id, this_ID]
    end
end

sd_ids_To_sd_IDs(dim::Integer, sd_ids) = (isempty(sd_ids) ? fill(1, dim) :
[sum(sd_ids .== i) + 1 for i = 1:dim]) |> Tuple
function gen_Kernel_Itpval(max_sd_order::Integer, dim::Integer)
    content = Expr(:block)
    itpval_kernel = gen_Kernel_Name("update_Basic_itgval_", max_sd_order, dim)
    for this_sd_order = 1:max_sd_order
        for sd_ids in Iterators.product(fill(1:dim, this_sd_order)...)
            (length(sd_ids) > 1) && (sum(sd_ids[2:end] .> sd_ids[1:(end - 1)]) > 0) && continue
            sd_IDs = sd_ids_To_sd_IDs(dim, sd_ids)
            arg_rhs = Expr(:call, :+)
            for muldim_ids in Iterators.product(fill(1:dim, this_sd_order)...)
                muldim_IDs = sd_ids_To_sd_IDs(dim, muldim_ids)
                single_term = Expr(:call, :*, :(ref_itp_vals[itg_id, itp_id, $(muldim_IDs...)]))

                for (this_sd_id, this_muldim_id) in zip(sd_ids, muldim_ids)
                    push!(single_term.args, :(
                        inverse_jacobian[$this_muldim_id, $this_sd_id, itg_id, this_ID]))
                end
                push!(arg_rhs.args, single_term)
            end
            push!(content.args, :(itg_vals[itg_id, itp_id,
            $(sd_IDs...), this_ID] = $arg_rhs))
        end
    end
    ex = :(@Dumb_GPU_Kernel ($itpval_kernel)(itg_vals::Array,
    ref_itp_vals::Array, inverse_jacobian::Array, IDs::Array) begin
        this_ID = IDs[thread_idx]
        for itg_id = 1:size(itg_vals, 1)
            for itp_id = 1:size(itg_vals, 2)
                $content
            end
        end
    end)
    return ex
end

BASE_KERNELS_2D = [Core.eval(@__MODULE__, gen_Kernel_Itpval(i, 2)) for i = 1:2]
BASE_KERNELS_3D = [Core.eval(@__MODULE__, gen_Kernel_Itpval(i, 3)) for i = 1:2]
for dim = 2:3
    Core.eval(@__MODULE__, gen_BasicDomain_Funcs(dim))
    Core.eval(@__MODULE__, gen_BasicBoundary_Funcs(dim))
end

@Dumb_GPU_Kernel update_Basic_Tangent_2D(jacobian::Array,
tangent_directions::Array, bdy_tangent_directions::Array, IDs::Array) begin
    this_ID = IDs[thread_idx]
    for itg_id = 1:size(jacobian, 3)
        tangent_directions[itg_id, 1, 1, this_ID] =
        jacobian[1, 1, itg_id, this_ID] * bdy_tangent_directions[itg_id, 1, 1] +
        jacobian[1, 2, itg_id, this_ID] * bdy_tangent_directions[itg_id, 2, 1]
        tangent_directions[itg_id, 2, 1, this_ID] =
        jacobian[2, 1, itg_id, this_ID] * bdy_tangent_directions[itg_id, 1, 1] +
        jacobian[2, 2, itg_id, this_ID] * bdy_tangent_directions[itg_id, 2, 1]
    end
end
```

```
@Dumb_GPU_Kernel update_Basic_Tangent_3D(jacobian::Array, tangent_directions::Array,
bdy_tangent_directions::Array, IDs::Array) begin
    this_ID = IDs[thread_idx]
    for itg_id = 1:size(jacobian, 3)
        tangent_directions[itg_id, 1, 1, this_ID] =
        jacobian[1, 1, itg_id, this_ID] * bdy_tangent_directions[itg_id, 1, 1] +
        jacobian[1, 2, itg_id, this_ID] * bdy_tangent_directions[itg_id, 2, 1] +
        jacobian[1, 3, itg_id, this_ID] * bdy_tangent_directions[itg_id, 3, 1]
        tangent_directions[itg_id, 2, 1, this_ID] =
        jacobian[2, 1, itg_id, this_ID] * bdy_tangent_directions[itg_id, 1, 1] +
        jacobian[2, 2, itg_id, this_ID] * bdy_tangent_directions[itg_id, 2, 1] +
        jacobian[2, 3, itg_id, this_ID] * bdy_tangent_directions[itg_id, 3, 1]
        tangent_directions[itg_id, 3, 1, this_ID] =
        jacobian[3, 1, itg_id, this_ID] * bdy_tangent_directions[itg_id, 1, 1] +
        jacobian[3, 2, itg_id, this_ID] * bdy_tangent_directions[itg_id, 2, 1] +
        jacobian[3, 3, itg_id, this_ID] * bdy_tangent_directions[itg_id, 3, 1]

        tangent_directions[itg_id, 1, 2, this_ID] =
        jacobian[1, 1, itg_id, this_ID] * bdy_tangent_directions[itg_id, 1, 2] +
        jacobian[1, 2, itg_id, this_ID] * bdy_tangent_directions[itg_id, 2, 2] +
        jacobian[1, 3, itg_id, this_ID] * bdy_tangent_directions[itg_id, 3, 2]
        tangent_directions[itg_id, 2, 2, this_ID] =
        jacobian[2, 1, itg_id, this_ID] * bdy_tangent_directions[itg_id, 1, 2] +
        jacobian[2, 2, itg_id, this_ID] * bdy_tangent_directions[itg_id, 2, 2] +
        jacobian[2, 3, itg_id, this_ID] * bdy_tangent_directions[itg_id, 3, 2]
        tangent_directions[itg_id, 3, 2, this_ID] =
        jacobian[3, 1, itg_id, this_ID] * bdy_tangent_directions[itg_id, 1, 2] +
        jacobian[3, 2, itg_id, this_ID] * bdy_tangent_directions[itg_id, 2, 2] +
        jacobian[3, 3, itg_id, this_ID] * bdy_tangent_directions[itg_id, 3, 2]
    end
end
```

```
@Dumb_GPU_Kernel update_Basic_Normal_2D(normal_directions::Array,
tangent_directions::Array, dets::Array, IDs::Array) begin
    this_ID = IDs[thread_idx]
    for itg_id = 1:size(normal_directions, 1)
        t1, t2 = tangent_directions[itg_id, 1, 1, this_ID],
        tangent_directions[itg_id, 2, 1, this_ID]
        # dets[itg_id, this_ID] = CUDA.sqrt(CUDA.pow(t1, 2.) + CUDA.pow(t2, 2.))
        dets[itg_id, this_ID] = sqrt(t1 ^ 2. + t2 ^ 2.)
        local_det = dets[itg_id, this_ID]
        normal_directions[itg_id, 1, this_ID] =  t2 / local_det
        normal_directions[itg_id, 2, this_ID] = -t1 / local_det
    end
end

@Dumb_GPU_Kernel update_Basic_Normal_3D(normal_directions::Array,
tangent_directions::Array, dets::Array, IDs::Array) begin
    this_ID = IDs[thread_idx]
    for itg_id = 1:size(normal_directions, 1)
        t1_1, t2_1, t3_1 = tangent_directions[itg_id, 1, 1, this_ID],
        tangent_directions[itg_id, 2, 1, this_ID], tangent_directions[itg_id, 3, 1, this_ID]
        t1_2, t2_2, t3_2 = tangent_directions[itg_id, 1, 2, this_ID],
        tangent_directions[itg_id, 2, 2, this_ID], tangent_directions[itg_id, 3, 2, this_ID]

        rn_1 =   t2_1 * t3_2 - t3_1 * t2_2
        rn_2 = - t1_1 * t3_2 + t3_1 * t1_2
        rn_3 =   t1_1 * t2_2 - t2_1 * t1_2

        local_det = sqrt(rn_1 ^ 2. + rn_2 ^ 2. + rn_3 ^ 2.)
        dets[itg_id, this_ID] = local_det

        normal_directions[itg_id, 1, this_ID] = rn_1 / local_det
        normal_directions[itg_id, 2, this_ID] = rn_2 / local_det
        normal_directions[itg_id, 3, this_ID] = rn_3 / local_det
    end
end
```