GetFEM: Automated FE modeling of multiphysics problems based on a generic weak form language

Yves Renard, Konstantinos Poulios

▶ To cite this version:

Yves Renard, Konstantinos Poulios. GetFEM: Automated FE modeling of multiphysics problems based on a generic weak form language. 2020. hal-02532422

HAL Id: hal-02532422
https://hal.science/hal-02532422

Preprint submitted on 8 Apr 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

GetFEM: Automated FE modeling of multiphysics problems based on a generic weak form language

Yves Renard[*] Konstantinos Poulios[†]

April 5, 2020

Abstract

This paper presents the major mathematical and implementation features of a weak form language (GWFL) for an automated finite element (FE) solution of partial differential equation systems. The language is implemented in the GetFEM framework and strategic modeling and software architecture choices both for the language and the framework are presented in detail. Moreover, conceptual similarities and differences to existing high level FE frameworks are discussed. Special attention is given to the concept of a generic transformation mechanism that contributes to the high expressive power of GWFL, allowing to interconnect multiple computational domains or parts of the same domain. Finally, the capabilities of the language for expressing strongly coupled multiphysics problems in a compact and readable form are shown by means of modeling examples.

keywords: Automated FEM, coupled PDEs, symbolic differentiation, weak form language

1 Introduction and aim

Modern numerical modeling tasks often require the solution of multiple coupled nonlinear partial differential equations (PDEs) possibly also subjected to additional algebraic equality or inequality constraints. The FE method typically employed for the discretization of such PDE systems relies on well founded mathematical principles [7], but the increasing complexity of the problems to solve leads to major programming challenges.

The repetitive nature of programming new FE formulations has been realized by many FE software developers and several parallel efforts have occurred for automating this process. The possible gains in terms of development and debugging time are evident, especially in the context of rigorous implicit solution schemes that require the derivation of a consistent Jacobian matrix. However, creating a high level FE modeling framework which is user friendly while at the same time as computationally efficient and versatile as lower level implementations, is a challenging assignment and many different approaches have been proposed so far.

[*]Université de Lyon, CNRS, INSA-Lyon, ICJ UMR5208, LaMCoS UMR5259, F-69621 Villeurbanne, France, (yves.renard@insa-lyon.fr)

[†]Technical University of Denmark, Department of Mechanical Engineering, Nils Koppels Allé, Building 404, Kgs. Lyngby, 2800, Denmark, (kopo@mek.dtu.dk)

Historically, the idea of automated FE development is rather old, with the FE system FINGER by [33] being one of the first implementations, along with the interactive system by [34]. The latter authors have in fact also highlighted the importance of using weak forms as a neutral mathematical description appropriate for automated FE modeling purposes. A thorough review of the first efforts in automation of FE development was given by [18], who also introduced the symbolic mechanics system (SMS), developed based on the computer algebra system MATHEMATICA® [19]. A similar solution based on the computer algebra system MAPLE® was made available by [1].

A common characteristic of these solutions is the generation of intermediate code that is compiled and incorporated into FE software with an otherwise rather classical element centered architecture. The current trend in FE is towards a more flexible use of finite element spaces, appearing in the late 90s and early 2000s in software such as FreeFEM [13, 14], GetFEM (`http://getfem.org/`), deal.II [3], FEniCS [24, 23], Firedrake [31] and oomph-lib [15], among several others. Both FreeFEM and FEniCS have early emphasized on modeling automation based on weak forms as user input, following very different implementation approaches though. FEniCS uses symbolic differentiation for the linearization of the global system of equations and similar to the solutions reviewed previously it relies on an external compiler to compile the generated code. FreeFEM implements its own interpreted language instead, which evaluates provided weak form expressions at runtime, but it lacks automatic linearization capabilities. A third approach is to extend the syntax of an existing low level language such as C++ to allow expressing weak forms corresponding to PDEs directly in the source code, as proposed e.g. by [30], [32] and [26, 25].

Addressing an important part of modeling automation, the weak form language GWFL is implemented in the GetFEM runtime library, which executes provided expressions without intermediate code generation and compilation. Technically, GWFL expressions are passed as text string arguments to functions of the available GetFEM APIs in C++, Python, Scilab and Matlab®. The underlying C++ library parses the provided expressions, allocates the necessary data structure and defines an optimized sequence of precompiled C++ function calls operating on the built data structure. Repetitive execution of this sequence of functions evaluates the provided expression efficiently and assembles it into a global residual vector or Jacobian matrix. Compared to the very modular architecture of FEniCS and Firedrake, GetFEM has GWFL as its only major level of abstraction. The less modular and pure runtime architecture is more convenient for implementing and exposing complex features to GWFL, such as inter-domain coupling.

The underlying mathematical formalism and terminology is presented in section 2, along with some basic syntax of the introduced weak form language. Section 3 describes the overall architecture of GetFEM that enables the implementation of the language and section 4 describes the core implementation of the weak form language. The numerical examples presented in section 5 demonstrate the expressive capabilities of the language for a rapid development of strongly coupled multiphysics models and the last section summarizes and concludes the work.

2 The generic weak form language

Let $\Omega \subset \mathbb{R}^d$ denote the reference domain of the problem to solve, with the d dimensions typically representing spatial coordinates but without excluding other kinds of dimensions such as time or frequency. A set of relevant physical, or also geometric, laws in Ω may be expressed in terms of a certain number of variables $u_1, \dots, u_n$ lying in function spaces $V_1, V_2, \dots, V_n$. In general, each variable u_i can be defined on a subset of Ω or its boundary $\partial\Omega$ or some internal interface contained in Ω and it can be a scalar,

vector or tensor field with a total number of components equal to q_i, e.g. $q_i = 1$ for a scalar field. Each variable u_i may be involved in physical laws valid in one or several subdomains $S_A, S_B, \dots$, either in the interior or on external boundaries or internal interfaces, c.f. Figure 1. Let the multi-index $I = \{i_1, i_2, \dots\}$ denote for convenience some specific subset of $\{1 \dots n\}$, so that any quantity indexed with I will be be constrained accordingly, i.e. $u_I \equiv \{u_{i_1}, u_{i_2}, \dots\}$.

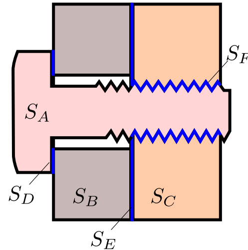


Figure 1: Possible subdomains in the interior, external boundary and internal interfaces of the problem domain Ω.

2.1 Zero-order terms, functionals

Many problems in science can be expressed as stationary points of a functional $J : V_{i_1} \times V_{i_2} \times \cdots \to \mathbb{R}$, mapping the unknown variables u_I to a scalar, defined through an integral on a subdomain S as

$$F_0(u_I) = \int_S G_0(u_I, \nabla u_I, \mathbb{H} u_I) dS. \tag{1}$$

The sought stationary point can be for instance a minimum of a strain energy potential or a saddle point of a Lagrangian function for a constrained problem. For some scalar variable u_i, the spatial gradient ∇u_i and Hessian $\mathbb{H} u_i$ correspond to a vector and a matrix respectively, but for a tensor variable u_i in general, they will just be tensors increased by one and two ranks, respectively.

Functionals in the form of F_0 will be referred to as zero order terms, and a problem definition may include several zero order terms on different subdomains S and with different subsets I of the problem variables. Of course, it is assumed that all variables u_I, represented in I, are defined on S.

2.2 First order terms, residuals

The first variation of a functional F_0 with respect to the set of variables u_I can be written as

$$\delta F_0(u_I; \delta u_I) = \sum_{i \in I} \int_S \frac{\partial G_0}{\partial u_i} [\delta u_i] + \frac{\partial G_0}{\partial \nabla u_i} [\nabla \delta u_i] + \frac{\partial G_0}{\partial \, \mathbb{H} u_i} [\mathbb{H} \delta u_i] \; dS, \tag{2}$$

where all involved directional derivatives of G_0 are by definition linear with respect to their direction argument inside the square brackets. For nonlinear problems, these derivatives will also depend on u_I, ∇u_I or $\mathbb{H} u_I$ nonlinearly. The linear form δF_0 with respect to the variations δu_I can be seen as a special case of so called first order terms, which are linear forms generally defined as

$$F_1(u_I; \delta u_I) = \int_S G_1(u_I, \nabla u_I, \mathbb{H} u_I \;; \quad \delta u_I, \nabla \delta u_I, \mathbb{H} \delta u_I) \; dS. \tag{3}$$

Here, G_1 is a function which is linear only with respect to the arguments after the semicolon separator, i.e. with respect to all listed variations. In a mathematical context, first order terms, defined exclusively through the integrand G_1, correspond to the weak form of some governing equations. In the context of numerical methods, first order terms are employed directly in the calculation of a residual vector, where variations δu_I are substituted with all relevant test functions.

2.3 Second order terms, Jacobians

One further variation of F_1 with respect to u_I leads to a second order term

$$F_2(u_I;\delta u_I,\varDelta u_I) = \int_S G_2(u_I,\nabla u_I,\mathbb{H}u_I\ ;\ \ \delta u_I,\nabla\delta u_I,\mathbb{H}\delta u_I,\ \ \ \varDelta u_I,\nabla\varDelta u_I,\mathbb{H}\varDelta u_I)\ dS, \tag{4}$$

where the function G_2 is linear with respect to all arguments listed after the semicolon separator. For each instance of the fields u_I in the space $V_{i_1}\times V_{i_2}\times\cdots$, the second order term F_2 is a bilinear form with respect to the variations δu_I and Δu_I.

In many cases, a second order term F_2 is obtained as the differential δF_1 of a first order term, through Gateaux differentiation. If the first order term is itself obtained from a zero order term, then δF_1 is the second derivative of F_0, in which case $F_2 = \delta F_1 = \delta^2 F_0$. Every piecewise differentiable zero order term can produce a corresponding first order term and every piecewise differentiable first order term can be converted to the corresponding second order term. However, not every bilinear form F_2 has an underlying linear form F_1, in the same manner as there is not a functional F_0 for every linear form F_1. In the context of numerical methods, second order terms are evaluated repeatedly for different test functions substituted into δu_I and Δu_I for assembling the corresponding components in the overall Jacobian matrix.

2.4 Algebraic variables

In addition to field variables defined on a single or different subdomains, covered so far, modeling of multiphysics problems often requires global scalar, vector, or tensor variables that are available in any domain. For such algebraic variables the notion of spatial gradient or Hessian obviously does not apply and the corresponding terms in Eqs. (1)-(4) have to be disregarded.

An algebraic variable u_1 can for example be used to apply a constraint on the integral or average over a volume S_A of a quantity that depends e.g. on a field variable u_2. If the desired constraint can for instance be expressed as minimization of a zero order term, it can be defined in the form

$$F_0(u_1,u_2,\nabla u_2,\mathbb{H}u_2) = \int_{S_A} G_0(u_1,u_2,\nabla u_2,\mathbb{H}u_2)\ dS. \tag{5}$$

Alternatively, instead of a zero order term, an appropriate first order weak form expression can be used for defining the desired constraint as well. Applying the same kind of coupling between the algebraic variable u_1 and another field variable u_3 defined on a subdomain S_B as between u_1 and u_2 is an indirect way of coupling two field variables, possibly defined in different subdomains.

2.5 The generic weak form language

In the above presented framework, the mathematical definition of a model is complete once all governing physics are expressed through zero or first order integrands, G_0 or G_1 respectively, on given integration

subdomains S for each of the provided integrands. The use of second order integrands G_2 in defining a model is more rare as it lacks the definition of a residual for the governing equations. Beyond the mathematical description, linearization and discretization are further modeling steps which represent the most tedious but rather repetitive part of an implementation and they are hence a major subject of automation. Leaving the discussion about such an automation for later, the present subsection introduces GWFL as a simple language for defining integrands in form of G_0, G_1 or G_2, which should ideally be the only input required for creating a new model.

GWFL is mainly meant for transferring formulations from paper to an ASCII text string that can be parsed and interpreted by the GetFEM software that implements the language. It is strictly limited to the symbolic definition of integrands G_0, G_1 and G_2 and is not meant to resemble a programming language or extend an existing programming language as the corresponding solutions in FEniCS, FreeFEM or Sundance. Compared to these, GWFL is somewhat similar to UFL in FEniCS, but it is even closer to the mathematical expressions, discretization agnostic and implemented as a runtime module. Its runtime implementation facilitates the incorporation of interpolate transformations, described in subsection 2.7, which in combination with a comprehensive set of linear and nonlinear operators, lead to a very high expressive power and compactness.

The proposed GWFL syntax supports user defined variables which can be scalars, vectors, matrices or higher rank tensors. Any number and naming of such variables is possible, with the only limitation of using ASCII characters and avoiding conflicts with predefined operator names. For scalar variables, the four common math operators `+`, `-`, `*`, `/` as well as functions such as `sqr`, `sqrt`, `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `pow` and `atan2(y,x)` apply. For non-scalar variables, operators `+`, `-`, `.*` and `./` are used for component-wise operations, operators `.` and `:` for single and double contraction between vectors, matrices or tensors, operator `*` for matrix-vector and matrix-tensor multiplications and operator `@` for tensor products, usually denoted in math as $\otimes$. As common in other scripting languages, the operator `'` expresses matrix transposition.

The non-smooth operators `pos_part` and `neg_part` correspond to the ramp functions $\langle x\rangle$ and $\langle -x\rangle$, respectively, and the `min(x,y)` and `max(x,y)` operators are also available in the language. Since these operators can only be differentiated once, they only allow derivation and computation of expressions of up to one order higher than the expression that they are contained in. The language also includes the non-continuous `Heaviside` operator. Its use is rather limited though, because it does not allow to derive any expression of higher order from the provided expression.

Spatial derivatives and variations appearing in the aforementioned integrands of the three kinds, G_0, G_1 and G_2, are expressed through special operators and prefixes, listed in Table 1. The variation prefixes apply to both field variables and algebraic variables while spatial derivative operators obviously apply only to field variables. The gradient and Hessian operators result in a tensor which will be respectively increased by one and two ranks compared to the original variable. Spatial derivative operators and variation prefixes can be combined to express for instance quantities like the gradient $\nabla\delta u$ of the variation of a variable u as `Grad(Test_u)`.

The language assumes that the dimension of all variables is provided at their definition, so that the compatibility of dimensions for all performed operations can be checked for. Syntax checking should also be able to identify the kind of integrand among the three possibilities G_0, G_1 and G_2, based on the presence of first variations (`Test_`) and second variations (`Test2_`) in the provided expression. The GWFL implementation also checks that the provided expression is indeed linear with respect to first variations δu_I for a first order term integrand G_1 , or bilinear with respect to first and second variations δu_I and

Table 1: Spatial derivative operators and test function prefixes used in GWFL.

`Grad(u)`, `Hess(u)`	Spatial gradient and Hessian of a variable `u`
`Div(u)`	Divergence of a vector variable `u`
`Test_`, `Test2_`	First and second variation of a variable (symbols δ and Δ respectively)

Δu_I in the case of a G_2 integrand.

Although not explicitly stated in Eqs. (1), (3) and (4), the integrands G_0, G_1 and G_2 may also depend directly on the current location X within the integration domain $S \subset \Omega$. Moreover, if S is a boundary or an internal interface of Ω, it is also very common that an integrand may depend on the normal direction with respect to S at the current location. In GWFL, the special variable name `X` is used for the current location X and the special variable name `Normal` for the unit normal vector. Both `X` and `Normal` are vectors of size d, hence for syntax checking purposes the language requires the dimension d of the space Ω to be known. If the integration domain S is in the interior of a domain where a normal direction cannot be defined, use of the keyword `Normal` in an expression results in an error message.

Square brackets can be used to define explicit tensors, including vectors and matrices, either using a nested, comma separated format or using comma and semicolon separators as in a Matlab®. Available tensors can also be sliced in a Matlab® like manner, using parentheses, the colon symbol and one-based indexing. The special operator `Id(n)` is used for defining an `n×n` identity matrix. In order to allow a compact writing of integrand expressions and an efficient implementation, GWFL supports a comprehensive set of linear and nonlinear operators on vectors, matrices and tensors, with the most important ones listed in Table 2. This includes efficient low level implementations of the derivatives of all nonlinear operators, including the matrix exponential and logarithm.

Table 2: Commonly used vector and matrix operators.

Linear operators on a symmetric matrix `s` or general tensor `t`: `Trace(s)`, `Deviator(s)`, `Sym(s)`, `Skew(s)`, `Reshape(t,m,n,...)`
Nonlinear operators on a general matrix or vector `x` or a symmetric matrix `s`: `Norm(x)`, `Norm_sqr(x)`, `Normalized(x)`, `Det(s)`, `Inv(s)`, `Expm(s)`, `Logm(s)`

Regarding scalar functions, GWFL supports user defined one and two argument scalar functions. Definition of a custom scalar function requires providing a name for the new function, and valid GWFL expressions for the value and the first derivatives of the function with respect to its arguments. Alternatively, instead of a GWFL expression, a pointer to a compiled C function can also be provided for the value of the newly defined function.

The basic syntax of GWFL presented in this subsection is complemented by the important `Derivative_` prefix and `Interpolate` operator presented respectively in subsections 2.6 and 2.7. Other, less frequently used, features and operators of the available language implementation in GetFEM are omitted in the interest of space.

2.6 Modeling automation

Providing an appropriate set of zero and first order terms expressed in GWFL, can be seen as a canonical form for defining a problem based on a minimum amount of information. Moreover, having a complete mathematical definition of a problem prior to linearization and discretization is a rigorous modeling approach, compared to enforcing physical laws on an already linearized or discretized system.

Apart from the mathematical definition, a symbolic representation such as GWFL constitutes an excellent format for describing both the input and output of an automated linearization procedure. The present section demonstrates how GWFL expressions are employed in a a fully automated computation of the residual vector and Jacobian matrix for a FE model. The presented example is a relatively simple coupled heat transfer problem but the procedure is directly applicable to much more complex modeling scenarios.

With u and T respectively representing displacements and temperature fields, heat transfer in a solid undergoing large deformations can be expressed in the presented framework by means of a first order integrand

$$G_1(u, T\ ;\ \delta T) = k\,|I + \nabla u|\,\nabla_x T \cdot \nabla_x \delta T, \tag{6}$$

where $\nabla_x = (I + \nabla u)^{-T}\nabla$ is the Eulerian gradient operator and k is the heat conductivity coefficient.

Assuming that corresponding field variables `u` and `T` as well as a constant `k` have been defined by the user, the aforestated integrand G_1 can easily be expressed in GWFL as

```
k*Det(Id(2)+Grad(u))
 *(Inv(Id(2)+Grad(u))'*Grad(T)).(Inv(Id(2)+Grad(u))'*Grad(Test_T))
```
(E1)

This parsable ASCII expression constitutes together with finite element spaces for `u` and `T` and a numerical integration method the only necessary input for assembling a corresponding residual vector for a discretized model. All necessary computations including evaluation of intermediate quantities like e.g. ∇u, ∇T and $(I + \nabla u)^{-T}$ at each integration point can be fully automated, resulting into an execution sequence of precompiled instructions. Such an execution sequence can then be repetitively called for different basis functions substituted into δT depending on the current position in the residual vector.

For the numerical solution of a problem, it is often essential to obtain a linearization of the considered model residual. In case of Eq. (6), the integrand is already linear with respect to T so that the corresponding second order term is obtained by substituting T with the second variation ΔT. The expression is however nonlinear with respect to u requiring a proper differentiation including the dependence of the Eulerian gradient operator ∇_x on u. In total, the resulting second order term integrand can be written as

$$\begin{aligned} G_2(u, T\ ;\ \delta u, \delta T,\ \Delta u, \Delta T) = {} & k\,|I + \nabla u|\,\nabla_x \Delta T \cdot \nabla_x \delta T \\ & + k \left(\left. \frac{\partial\,|A|}{\partial A} \right|_{A=I+\nabla u} : \nabla \Delta u \right) \nabla_x T \cdot \nabla_x \delta T \\ & + k\,|I + \nabla u| \left(\left(\left. \frac{\partial A^{-1}}{\partial A} \right|_{A=I+\nabla u} : \nabla \Delta u \right)^T \nabla T \right) \cdot \nabla_x \delta T \\ & + k\,|I + \nabla u|\,\nabla_x T \cdot \left(\left(\left. \frac{\partial A^{-1}}{\partial A} \right|_{A=I+\nabla u} : \nabla \Delta u \right)^T \nabla \delta T \right), \end{aligned} \tag{7}$$

The derivation of second order terms as the one above and their implementation in the assembly of a global Jacobian matrix is often one of the major modeling tasks, requiring considerable effort even in semi-automated FE frameworks. However, these steps can easily be automated if low level implementations for the derivatives of all involved nonlinear operators and functions are available. In GWFL such derivatives are accessed through the `Derivative_` prefix. For example, `Derivative_Det(A)` and `Derivative_Inv(A)` can be used to respectively express the derivatives $\partial |A| / \partial A$ and $\partial A^{-1} / \partial A$ appearing in Eq. (7). For two argument functions the prefix `Derivative_2_` represents derivatives with respect to the second argument. Based on this syntax, the last three terms of Eq. (7) can be written in GWFL as

```
(Inv(Id(2)+Grad(u))'*Grad(Test_T))
.((k*(Derivative_Det(Id(2)+Grad(u)):Grad(Test2_u)))*(Inv(Id(2)+Grad(u))'*Grad(T))+
  (k*Det(Id(2)+Grad(u)))*((Derivative_Inv(Id(2)+Grad(u)):Grad(Test2_u))'*Grad(T)))     (E2)
+(k*Det(Id(2)+Grad(u))*(Inv(Id(2)+Grad(u))'*Grad(T)))
.((Derivative_Inv(Id(2)+Grad(u)):Grad(Test2_u))'*Grad(Test_T))
```

This is actually, with minor aesthetic modifications, the expression automatically generated by GetFEM based on the GWFL expression (E1). Such basic symbolic algebra processing of expressions of zero or first order to respectively generate first and second order expressions relies essentially on the application of the chain and product rules for differentiation. Further factorization of the generated expression is of minor importance since the presence of repeated subexpressions has anyway to be dealt with in the computational implementation.

In the same manner as an automated assembly of the residual vector for a discretized model relies on first order GWFL expressions like (E1), the assembly of the global Jacobian matrix can be fully automated based on second order GWFL expressions like (E2). The contribution of the considered terms to the $\{k, l\}$ entry in the global Jacobian matrix is computed by evaluating (E2) at each integration point with δT and Δu respectively corresponding to the global k-th and l-th degree of freedom basis functions. A more in-depth implementation description of an efficient automated assembly of GWFL expressions is provided in section 4.

Even in the relatively simple example shown here, a GWFL based automated assembly results in a considerable gain in terms of modeling effort compared to a lower level implementation. More elaborate multiphysics problems with complex nonlinearities and couplings between several unknown fields is however where the value of the proposed FE automation really becomes evident.

As already explained, the main use of the GWFL is in the definition of zero, first and second order terms, typically representing PDEs in weak form. However, the same syntax can also be used in the definition of interpolation expressions, which, similar to a zero order integrand G_0, are expressions without any variable variations. In contrast to the latter though, interpolation expressions are not limited to scalars but can in general evaluate to scalar, vector, matrix or higher rank tensor quantities. They are used in interpolation operations which often constitute a necessary step within more complex numerical models or they are just used for preparing post-processing output. As an example, in connection to the presented heat transfer example, the expression

```
-k*Det(Id(2)+Grad(u))*(Inv(Id(2)+Grad(u))'*Grad(T))     (E3)
```

can be used in order to evaluate and export the heat flux vector field.

2.7 Interpolate transformations

So far, it has been assumed that integrands in functional, residual or Jacobian terms involve field quantities, defined at the current integration point X, and algebraic variables, defined globally. Nevertheless, in order to achieve certain couplings in numerical modeling, there are often terms that involve field quantities defined at different points of the same or different subdomains. To cover this need, interpolate transformations in GWFL provide a generic mechanism that allows to map the current integration point X to another point Y either in the same or in a different subdomain and access the variables, spatial derivatives and test functions defined at Y.

Figure 2 illustrates the mapping between two points X and Y both conceptually and in terms of specific applications in numerical modeling. The wide range of numerical methods that can be implemented based on interpolate transformations in GWFL is indicative of the expressive power of this mechanism in a high level modeling language.

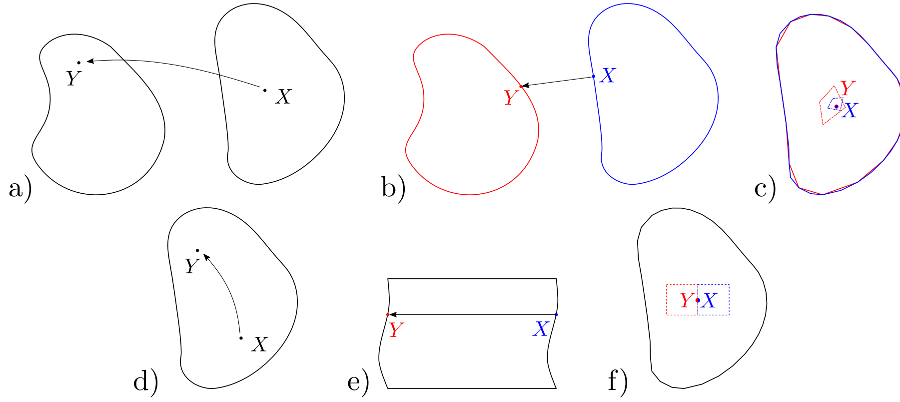


Figure 2: Transformation between different subdomains (a), used e.g. for implementing contact conditions (b) and interpolation between different dicretizations (c), or within a single domain (d), used e.g. for imposing periodic boundary conditions (e), and implementing discontinuous Galerkin methods (f).

In most cases, the mapping of point X to a point Y can be defined as

$$Y = \mathcal{Y}(X). \tag{8}$$

involving only the coordinates of the two points in the reference configuration. In order to support a wider range of applications though, a more generic form is adopted according to the expression

$$Y = \mathcal{Y}(X, u_1, u_2, \dots). \tag{9}$$

with u_1, u_2, ... being problem variables. The dependence of the transformation $\mathcal{Y}$ on problem variables allows for instance the implementation of contact conditions between deformable bodies undergoing large deformations. Another use case is the implementation of advective terms where the point Y can be defined at a small distance from point X in the upstream or downstream direction with respect to an a priori unknown velocity field.

Allowing to use variables and test functions at the mapped point $Y = \mathcal{Y}(X, \dots)$ when evaluating an integrand at point X, is a powerful coupling mechanism, even more so for a point Y lying in a different

subdomain. GWFL provides this mechanism through the syntax of Table 3, with the user defined names `u` and `transname` respectively representing a variable u and some transformation $\mathcal{Y}$. This syntax allows to transfer a variable u, a variation δu, or their spatial gradients, from point Y to X. Additionally, the syntax allows to retrieve the coordinates of the transformed point Y, or the surface unit normal at Y, if Y lies on a boundary.

It should be noted that the syntax simply denotes evaluation of a quantity at the mapped point Y, typically by interpolation in the discretized setting. However, a variation or a spatial derivative of an interpolated variable at Y are not just equivalent to evaluating its variation or spatial derivative at the mapped point Y. These quantities also involve the derivatives of the transformation $Y = \mathcal{Y}(X, \dots)$ with respect to its arguments. For example, the variation of $u|_{\mathcal{Y}}$ is

$$\delta u|_{\mathcal{Y}} + \nabla u|_{\mathcal{Y}} \cdot \frac{\partial \mathcal{Y}}{\partial u_i} \cdot \delta u_i, \tag{10}$$

and the spatial gradient of $u|_{\mathcal{Y}}$ is

$$\nabla u|_{\mathcal{Y}} \cdot \left(\frac{\partial \mathcal{Y}}{\partial X} + \frac{\partial \mathcal{Y}}{\partial u_i} \cdot \nabla u_i \right), \tag{11}$$

where i implies summation over all model variables that the transformation $\mathcal{Y}$ actually depends on. Eq. (10) is part of the GWFL implementation, as it is necessary for the automatic differentiation of zero and first order terms containing interpolate transformation syntax from Table 3. Eq. (11) is also implemented as part of the `Grad()` operator, when it acts on a symbolically defined transformation $\mathcal{Y}$.

A symbolic definition of an interpolate transformation in the form of Eq. (9) is possible by using the GWFL syntax itself through the GetFEM model method `add_interpolate_transformation_from_expression`, which expects a name for the new transformation, a source and a target mesh, which can possibly be the same, and a symbolic GWFL expression defining $\mathcal{Y}(X, \dots)$. For example, the identity transformation $\mathcal{Y}(X) = X$, useful for implementing the case of Figure 2c, can easily be defined by providing `X` as the transformation expression and two distinct source and target meshes. Slightly more complex than the case of Figure 2e, imposing a rotational periodicity condition e.g. at 60° is another example that the transformation defined in GWFL syntax as

Table 3: Interpolate transformation syntax in GWFL.

`Interpolate(X,transname)`	$\mathcal{Y}(X, \dots)$
`Interpolate(Normal,transname)`	Surface unit normal at $\mathcal{Y}(X, \dots)$
`Interpolate(u,transname)`	$u\|_{\mathcal{Y}(X,\dots)}$
`Interpolate(Grad(u),transname)`	$\nabla u\|_{\mathcal{Y}(X,\dots)}$
`Interpolate(Hess(u),transname)`	$\mathbb{H} u\|_{\mathcal{Y}(X,\dots)}$
`Interpolate(Test_u,transname)`	$\delta u\|_{\mathcal{Y}(X,\dots)}$
`Interpolate(Grad(Test_u),transname)`	$\nabla \delta u\|_{\mathcal{Y}(X,\dots)}$
`Interpolate(Hess(Test_u),transname)`	$\mathbb{H} \delta u\|_{\mathcal{Y}(X,\dots)}$

```
[cos(pi/3),-sin(pi/3);sin(pi/3),cos(pi/3)]*X
```
(E4)

can be useful for. As a last example, given a displacement field `u`, a transformation defined as `X+u` can be used for implementing data transfer between the Eulerian and Lagrangian settings. An advantage of such a symbolic definition of an interpolate transformation is that all necessary derivatives appearing in Eq. (10) can be generated automatically, using the same mechanism as explained in section 2.6. In the simple case of expression `X+u` for instance, the derivative of the interpolate transformation with respect to `u`, i.e. $\partial\mathcal{Y}/\partial u$, will be the identity matrix.

There are, nevertheless, useful interpolate transformations that cannot be defined symbolically. Such an example is a raytracing transformation as in Figure 2b but based on the deformed bodies according to corresponding displacement fields. This rather complex transformation is programmed as part of GWFL in GetFEM and it allows to implement the algorithm for large deformations contact according to [29] purely with GWFL syntax. In a model with several deformable solids it might be advantageous to define raytracing transformations between different sets of bodies, hence there is no fixed name for an overall raytracing transformation, but the user can define and name multiple raytracing transformations with different source and target meshes and displacement fields. If, for instance, `ray12` is a user defined raytracing transformation between mesh 1 and mesh 2 with displacement fields `u1` and `u2`, respectively , the distance between master and slave points is obtained in GWFL by

```
Norm(Interpolate(X,ray12)+Interpolate(u2,ray12)-(X+u1))
```
(E5)

where the transformation `ray12` depends on both `u1` and `u2` through its definition. To account for this dependence in Eqs. (10) and (11), the derivatives $\partial\mathcal{Y}/\partial u_1$ and $\partial\mathcal{Y}/\partial u_2$ are needed. As there is no symbolic definition of $\mathcal{Y}$ in this case, these derivatives are pre-implemented numerically in the language and can be accessed in GWFL using a special syntax, which for example for $\partial\mathcal{Y}/\partial u_1$, is

```
Interpolate_derivative(ray12,u1)
```
(E6)

Another important interpolate transformation, which is part of the standard GWFL and is based on a numerical rather than symbolic definition, is the `neighbor_element` transformation. Mathematically, it corresponds to the identity transformation $\mathcal{Y}(X) = X$. Numerically, however, for X being a point on a common face between two elements, the transformation returns the element and face number of the neighbor element compared to the element that X is defined on. Internally, it also returns the coordinates of the transformed point $\mathcal{Y}(X)$ in the reference element corresponding to that neighbor element. The `neighbor_element` transformation is essential for implementing discontinuous Galerkin methods. If, for example, `u` is a user defined variable approximated on a mesh with discontinuous finite elements, the expressions (E7) and (E8) respectively provide the variable jump and average on the common face between two elements.

```
u - Interpolate(u, neighbor_element)
```
(E7)

```
(u + Interpolate(u, neighbor_element))/2
```
(E8)

3 Software architecture

The GWFL syntax, introduced in the previous section, offers a flexible, easy and rather universal way of defining problems and corresponding systems of PDEs, involving an arbitrary number of field variables.

It mainly concerns the continuous setting, with only few features being specific to a discretization with finite elements. However, automating the assembly of functional, residuals and Jacobians, addresses only part of the time-consuming and error-prone tasks in numerical modeling. The definition, for instance, of appropriate finite element spaces and numerical integration methods are other areas that a high level modeling framework is expected to minimize implementation effort for. In this context, the present section describes the overall architecture of the GetFEM framework that GWFL has been implemented in.

GetFEM is an object oriented framework implemented in C++, exploiting polymorphism in order to support extendibility at the C++ level. Most of the framework's standard functionality is also available through a common interface to the scripting languages Python, Scilab and Matlab®. The following subsections highlight good software design choices and describe the major C++ objects that implement the GWFL functionality.

3.1 Meshes and integration methods

One central idea behind the design of the GetFEM framework is the separation between mesh, finite element spaces and integration methods. This idea is reflected in the diagram of Figure 3 which shows the overall software architecture.

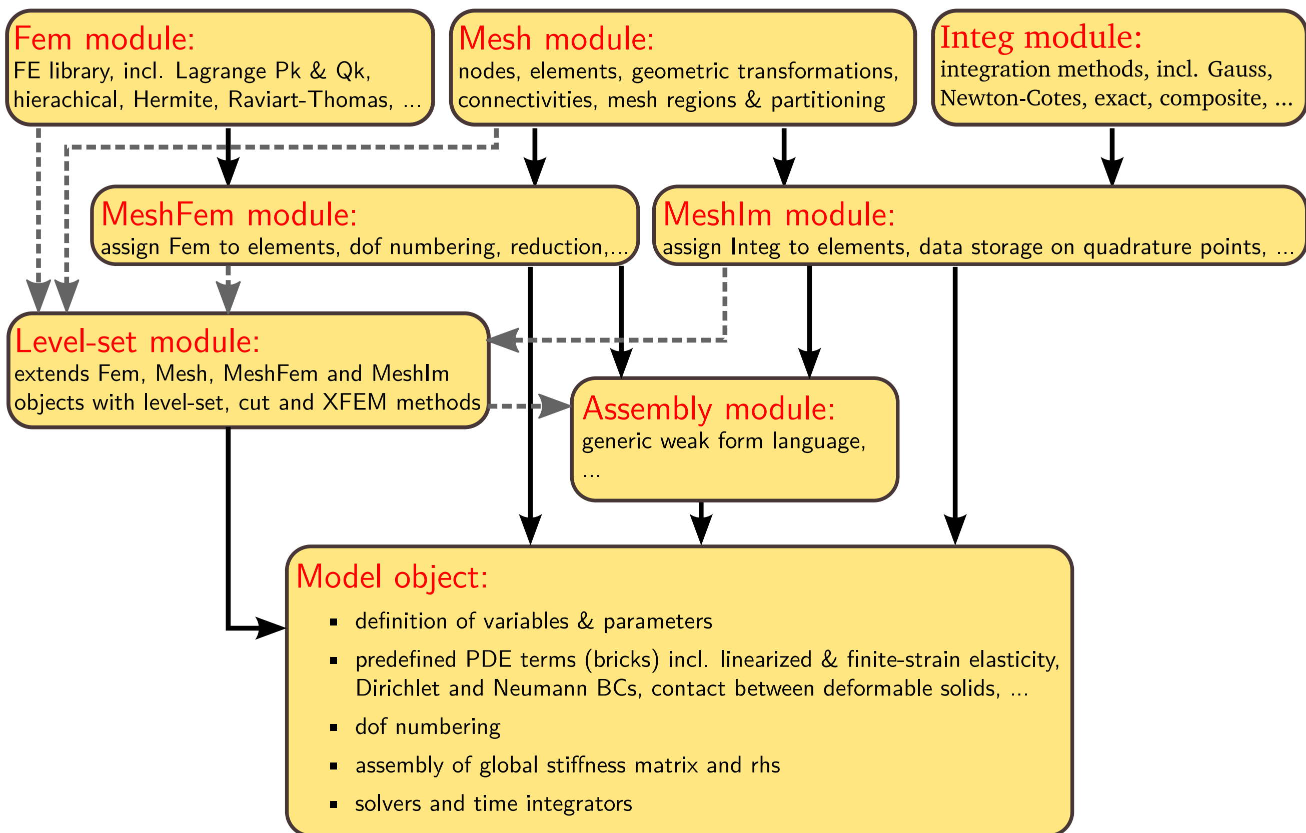


Figure 3: GetFEM software architecture diagram.

In the formalism introduced in the previous section, the whole model definition is expressed in terms

of integrals over different subdomains. The geometric definition of integration domains is hence essential for any respective numerical implementation. In GetFEM, integration domains are represented in an approximate sense through computational meshes, which can either be generated directly in GetFEM or, for more complex geometries, be imported from mesh files generated with GMSH [11], ANSYS® or GiD®.

In standard cases, the overall problem domain $\Omega \subset \mathbb{R}^d$ is approximated by several, possibly overlapping, computational subdomains Ω_i that typically correspond to computational meshes $\mathscr{T}_i^h$. The computational subdomains Ω_i do not necessary all need to be of the same dimension, but they can, for instance in 3D, be a combination of volumes, surfaces and curves. Another possibility for defining a computational subdomain Ω_i is by using a nonconformal mesh in combination with a level-set description of the boundary of Ω_i. In these cases, integration methods adapted to a given level-set are essential.

The discretization of the problem domain into simple elements serves two purposes. It mainly enables the numerical evaluation of integrals as those discussed in subsections 2.1-2.3 and secondly it provides a means for defining solution and test function spaces piecewise. This latter role of computational meshes is an inseparable ingredient of the classical finite element method but not as essential for the construction of basis functions in other Galerkin-type methods such as meshless methods and XFEM. In GetFEM terminology, the finite element method is understood in a broad sense that encompasses Galerkin methods in general.

In that sense, the main role of elements is to allow a simple numerical integration by furnishing a parametrization of their domain through a mapping from a fixed reference element. The mesh module included in Figure 3 supports the definition of geometric transformations τ_T, which map some reference element $\hat{T}$ to each real element T through the usual mapping

$$X = \tau_T(\hat{X}) = G\,\mathcal{N}(\hat{X}), \tag{12}$$

with points $\hat{X}$ and X lying in the reference and real elements, respectively. For an element with n_g nodes, G is a $d \times n_g$ matrix, containing all nodal coordinates in the real element, and $\mathcal{N}(\hat{X})$ is a vector of n_g shape functions. To maintain generality, the reference element $\hat{T}$ is defined in a possibly different space $\mathbb{R}^p$ than the real element space $\mathbb{R}^d$, with $p \leq d$. One consequence of this choice is that the derivative $\partial X/\partial \hat{X}$ of the geometric transformation and its pseudo-inverse are in general non-square $d \times p$ matrices, defined as

$$K(\hat{X}) = G\,\nabla_{\hat{X}}\mathcal{N}(\hat{X}) \qquad \text{and} \qquad B(\hat{X}) = K(\hat{X})\left(K(\hat{X})^T K(\hat{X})\right)^{-1}, \tag{13}$$

where the shape function derivatives matrix $\nabla_{\hat{X}}\mathcal{N}(\hat{X})$ has dimensions $n_g \times p$. Of course in the usual case of $p = d$, matrix K is a square one and its pseudo-inverse B reduces to the regular inverse. For nonlinear geometric transformations, the interpolation function gradients $\nabla_{\hat{X}}\mathcal{N}(\hat{X})$ and consequently also matrices K and B vary spatially. In this case, mapping a given point from the real element space to the reference element space requires an iterative solution. Such a solver is employed whenever for example interpolation or contact between two arbitrary meshes needs to be evaluated, c.f. Figure 2.

Geometric transformations are defined in terms of the reference element's n_g nodes and their connectivity as well as the corresponding shape functions $\mathcal{N}(\hat{X})$. An appropriate naming system in GetFEM, partially documented in the appendix Table 6, provides access to a set of pre-implemented and tabulated geometric transformations. In most of these, the components of $\mathcal{N}(\hat{X})$ are typically polynomials with respect to the p coordinates of $\hat{X}$, but more complex functions are also used such as rational functions

for the implementation of pyramid elements, [12, 5]. In general, it is rather simple to add new element types in GetFEM by specifying the corresponding geometric transformations. Apart from the nodes of the reference element it is only necessary to specify the shape functions $\mathcal{N}(\hat{X})$ analytically. The provided expressions are parsed by a basic symbolic system for polynomials and rational functions, available in GetFEM. All necessary derivatives $\nabla_{\hat{X}}\mathcal{N}(\hat{X})$ are also obtained and evaluated by this system. Moreover, GetFEM supports so called composite elements where shape functions $\mathcal{N}(\hat{X})$ are piecewise defined within a single element, facilitating e.g. the implementation of geometric multigrid algorithms.

Apart from the definition of geometric transformations, the mesh module includes objects and methods for defining the topology of a mesh through node connectivities in elements as well as so called mesh regions, which are sets of elements or element faces, used for specifying the integration domain S for an added weak form term. In contrast to first generation finite element codes, where users had to directly refer to element or node numbers, in high-level frameworks like GetFEM this is rarely the case.

One major role of an element based discretization of the problem domain is the application of numerical integration methods for calculating the weak form integrals discussed in Section 2. The implementation of GWFL is very much linked to numerical integration methods as the language itself basically describes integrands that have to be repeatedly evaluated at every relevant integration point. Integration points and the corresponding weights are defined as usual in the reference element and area scaling between the reference and the real element are accounted for through the geometric transformation matrices from Eq. (10). As the integration domain S of an added weak form term can refer to the interior of a problem subdomain Ω_i or to its surface $\partial\Omega_i$, numerical integration methods need to define integration points and weights not only in the interior of elements but also on their faces.

The MeshIm module in Figure 3 deals with numerical integration, with its main object decorating a mesh object with selected integration methods per element. An appropriate naming system, partially covered in the appendix Table 7, provides access to a set of tabulated integration methods and new methods are easy to define by providing integration points and weights on the reference element. Adaptive integration for fictitious domain methods is also well supported but not covered here.

3.2 Finite element description

The numerical solution of PDEs by Galerkin methods relies on finite dimensional function spaces for both solution and test functions. In the proposed weak form language in particular, test functions are assumed to match the defined solution spaces by prepending the corresponding variable with the `Test_` prefix to express a virtual variation. GetFEM provides several tools for the construction of function spaces on a given computational mesh. The main object of the MeshFem module in Figure 3 decorates each element of a mesh with a finite element object, defining a set of degrees of freedom that can either be associated to specific nodes or not. From a software architectural point of view, it is very useful to exploit polymorphism in order to abstract two different mechanisms of either defining the shape functions on an element or as global functions.

The standard finite element method combines simple elementwise solution spaces, typically defined on the reference element, to obtain solution spaces for each problem subdomain Ω_i. Assuming that solution spaces are mapped from the reference to the real element T, a finite element is defined by [7] as a triplet $(T, V_T, \mathcal{L}_T)$, where

- T is the geometric element,

- V_T is a N-dimensional vector space of functions over T,
- $\mathcal{L}_T = \{\ell_1, \ell_2, \dots, \ell_N\}$ is a set of N linear forms over V_T (the degrees of freedom),

such that $\mathcal{L}_T$ is unisolvent with respect to V_T, i.e. each function of V_T is determined by a unique set of degrees of freedom in $\mathcal{L}_T$. Then, the space V_T can be written as $V_T = \text{Span}\{\varphi_1, \varphi_2, \dots, \varphi_N\}$ where φ_i are the so called shape functions satisfying the condition $\ell_i(\varphi_j) = \delta_{ij}$.

The simplest and most common way of constructing the function space V_T is by direct mapping from a corresponding space $\hat{V}_T = \text{Span}\{\hat{\varphi}_1, \hat{\varphi}_2, \dots, \hat{\varphi}_N\}$ on the reference element, obtained by

$$\varphi_i(\tau_T(\hat{X})) = \hat{\varphi}_i(\hat{X}), \tag{14}$$

where τ_T is the geometric transformation from the reference to the real element, which can be affine or not. In GetFEM terminology, finite element types that can be constructed by Eq. (14), like e.g. all Lagrange elements, are denoted as τ-equivalent elements.

More complex elements, such as intrinsically vector elements and Hermite elements, are not τ-equivalent because a transformation of shape functions from the reference element involves a more complex dependence on the geometric transformation τ_T like for example including its derivatives. Extending Eq. (14) with a linear transformation matrix M_T, which may actually depend on the geometric transformation τ_T and hence on the real element, leads to the more general mapping

$$\varphi_i(\tau_T(\hat{X})) = \sum_{j=1}^{N} (M_T)_{ij} \hat{\varphi}_j(\hat{X}), \tag{15}$$

that can facilitate the construction of a wider class of finite elements, still based on a reference element. This mechanism has been the standard way of defining elements like Hermite, Argyris or Raviart-Thomas in GetFEM and has independently been proposed by other authors [10, 17]. The use of Eq. (15) in defining complex elements consists basically in determining the necessary matrix M_T and implementing it efficiently, since it has to be evaluated on each real element. An example of an advanced element type implemented in this manner in GetFEM is the Argyris triangular element that is made compatible with both affine and non-affine geometric transformations, also mapped onto a surface element in 3D.

A large set of pre-implemented finite element types are accessible through a corresponding naming system in GetFEM, with some common of them listed in the appendix Table 8. These also include intrinsically vector elements such as Raviart-Thomas and Nedelec elements, widely used in electromagnetism and mixed formulations, [6]. Otherwise, vector-valued or even tensor-valued fields can also be defined component-wise based on a finite element with scalar shape functions. Apart from the definition of basis functions per element it is also possible to define globally indexed basis functions $\varphi_I(X)$ and assign them to all elements within their support $\varphi_I(X) \neq 0$. This is a useful feature for the implementation of a broad class of numerical methods like XFEM, but beyond the scope of the present work.

3.3 The model and workspace objects

The purpose of the infrastructure presented in the previous subsections is to construct rather arbitrary solution spaces and numerical integration schemes serving the computation of weak form terms, discussed in section 2. The respectively constructed `mesh_fem` and `mesh_im` objects representing different variables

and integration methods in one or multiple domains can nevertheless only be useful as building blocks for an overall model. In the general case of several arbitrarily coupled variables, setting up such a model can become time consuming and error-prone, when programmed manually.

Other high level finite element frameworks such as FEniCS [23] and FreeFEM [13, 14] define problem variables directly as objects in their programming environment. A different paradigm is followed here with the whole model definition encapsulated in the GetFEM object `model`. This object has a very extended functionality including e.g. pre-implemented PDE terms, a Newton solver, methods facilitating the implementation of time integrators, etc. Alternatively, there is also the `ga_workspace` object which is lighter and strictly limited to the assemblage of zero, first and second order terms. Both objects allow the definition of variables and data, which can either be scalar or defined on a finite element space or defined on integration points. The GWFL is implemented as part of these objects, so that any valid GWFL expression can be added to an instance of these classes, provided that the involved variables and data names are previously defined. The superposition of all added expressions to a `model` or `ga_workspace` object is then evaluated upon each request for the residual vector or the Jacobian matrix.

This setup is suitable for a monolithic solution where all unknowns are addressed at once in a single Newton loop, which is a very efficient approach for moderate size problems that can be solved with an efficient direct solver [2]. For problems where a monolithic application of Newton's method is not sufficient, the `model` class facilitates the implementation of staggered solution schemes by allowing to temporarily disable some of the variables and treat them as data until they are re-enabled. In addition, problems with instabilities and bifurcated solutions can be treated with numerical continuation algorithms [8, 21, 22], where the commonly used continuation parameter is simply defined as scalar data in GWFL. In summary, both `model` and `ga_workspace` objects can represent complex multiphysics problems easily, dealing with multiple unknowns on appropriately constructed solution spaces and multiple weak form terms, describing the different physics and couplings of variables.

4 Implementation aspects

The previous sections have focused on problem formulation and software design choices that aim at a high level of freedom, flexibility and universality in the intended numerical modeling. For solving real engineering problems though, performance is also essential because of the often three dimensional and complex geometries involved, inevitably leading to a large number of degrees of freedom. This section will hence mainly focus on implementation aspects that are important for achieving a high computational efficiency.

4.1 Elementary computations and assembly

As the degree of nonlinearities and coupling in a multi-field system of PDEs increases, a computationally efficient residual vector and Jacobian matrix assembly also becomes increasingly important. These assembly operations involve repeated evaluations of first and second order terms respectively according to Eqs. (3) and (4), for a large number of variations δu_I and Δu_I. Here, the computation of the Jacobian matrix based on a second order term will be described as the most general case, with the simplification to first order or even zero order terms being rather obvious.

Let $F_2(u_I; \delta u_I, \Delta u_I)$ be a second order term defined on a geometric entity S according to Eq. (4). One can define the restriction of F_2 to only first variations of variable u_α and second variations of variable u_β,

as

$$F_{2|\alpha,\beta}(u_I;\delta u_\alpha,\Delta u_\beta)=F_2(u_I;\{0,\dots,0,\delta u_\alpha,0,\dots,0\},\{0,\dots,0,\Delta u_\beta,0,\dots,0\}), \tag{16}$$

with zeros denoting zero functions for the variations of the remaining variables. This restriction of F_2 describes only the coupling between variables u_α and u_β,

Let now j_α be the index of one degree of freedom for the discretized variable u_α in the global system and j_β be another global index for a degree of freedom corresponding to variable u_β. Then, let φ_{j_α} and φ_{j_β} denote the corresponding basis functions from the finite element spaces V^h_α and V^h_β used for the approximation of variables u_α and u_β, respectively. Under these definitions, the contribution of the second order term to the (j_α, j_β) entry of the global Jacobian matrix K_G is obtained by substituting δu_α and Δu_β in $F_{2|\alpha,\beta}$ with φ_{j_α} and φ_{j_β}, i.e.

$$F_{2|\alpha,\beta}(u_I;\varphi_{j_\alpha},\varphi_{j_\beta})\to K_G(j_\alpha,j_\beta).$$

From the definition of F_2 in Eq. (4) and its restriction in Eq. (16), the above contribution to $K_G(j_\alpha, j_\beta)$ can be evaluated as

$$\begin{aligned}&F_{2|\alpha,\beta}(u_I;\varphi_{j_\alpha},\varphi_{j_\beta})=\\&\qquad\sum_{T\in\mathscr{T}^h}\int_{T\cap S}G_{2|\alpha,\beta}(u_I,\nabla u_I,\mathbb{H}u_I\ ;\ \ \varphi_{j_\alpha},\nabla\varphi_{j_\alpha},\mathbb{H}\varphi_{j_\alpha},\ \ \varphi_{j_\beta},\nabla\varphi_{j_\beta},\mathbb{H}\varphi_{j_\beta})\ dS,\end{aligned} \tag{17}$$

with $G_{2|\alpha,\beta}$ denoting the restriction of the weak form integrand G_2 to variations of variables u_α and u_β exclusively, equivalent to Eq. (16). The intersection $T\cap S$ is included for generality to cover cases such as fictitious domain methods where T may lie only partially in S.

A single element T will contribute to the coupling term between the two variables u_α and u_β in K_G for several degrees of freedom j_α and j_β. If the corresponding sets of active degrees of freedom in element T are defined as

$$J_{\alpha|T}=\{j_\alpha:\mathrm{supp}(\varphi_{j_\alpha})\cap T\neq\emptyset\}\quad\text{ and }\quad J_{\beta|T}=\{j_\beta:\mathrm{supp}(\varphi_{j_\beta})\cap T\neq\emptyset\},$$

then the contribution of element T to the Jacobian matrix portion related to first variations of variable u_α and second variations of variable u_β can be summarized to an elementary matrix

$$\begin{aligned}K_{\alpha,\beta|T}&=K_G(J_{\alpha|T},J_{\beta|T})\\&=\int_{T\cap S}\Big[G_{2|\alpha,\beta}(u_I,\nabla u_I,\mathbb{H}u_I\ ;\ \varphi_{j_\alpha},\nabla\varphi_{j_\alpha},\mathbb{H}\varphi_{j_\alpha},\ \varphi_{j_\beta},\nabla\varphi_{j_\beta},\mathbb{H}\varphi_{j_\beta})\Big]_{\substack{j_\alpha\in J_{\alpha|T}\\ j_\beta\in J_{\beta|T}}}\ dS\\&\approx\sum_p w_p\Big[G_{2|\alpha,\beta}(u_I,\nabla u_I,\mathbb{H}u_I\ ;\ \varphi_{j_\alpha},\nabla\varphi_{j_\alpha},\mathbb{H}\varphi_{j_\alpha},\ \varphi_{j_\beta},\nabla\varphi_{j_\beta},\mathbb{H}\varphi_{j_\beta})\big|_{X=X_p}\Big]_{\substack{j_\alpha\in J_{\alpha|T}\\ j_\beta\in J_{\beta|T}}}\end{aligned} \tag{18}$$

The last approximate evaluation in Eq. (18) represents the actual numerical integration method, with X_p denoting an integration point and w_p the corresponding weight. For computational efficiency, it is common to calculate the integrand $G_{2|\alpha,\beta}$ in a vectorized manner, i.e. for all active degrees of freedom $(J_{\alpha|T}, J_{\beta|T})$ in the current element T instead of an individual pair (j_α, j_β) at a time. With a slight change in notation, Eq. (18) can be rewritten as

$$K_{\alpha,\beta|T}\approx\sum_p w_p K^{<p>}_{\alpha,\beta|T}(u_I,\nabla u_I,\mathbb{H}u_I\ ;\ \varphi_{u_\alpha|T},\nabla\varphi_{u_\alpha|T},\mathbb{H}\varphi_{u_\alpha|T},\ \varphi_{u_\beta|T},\nabla\varphi_{u_\beta|T},\mathbb{H}\varphi_{u_\beta|T}) \tag{19}$$

with $K^{<p>}_{\alpha,\beta|T}$ denoting the contribution of the integration point X_p to the element tangent matrix. The vectorization has been moved here to the arguments $\varphi_{u_\alpha|T}$ and $\varphi_{u_\beta|T}$, respectively representing all basis functions of the finite element spaces for u_α and u_β that are nonzero on T. For the computation of values, gradients and Hessians of these shape functions use is made of any available precomputations either on the real or the reference element, depending on the finite element type. In total, the global assembly procedure is a quite standard one corresponding to Algorithm 1.

Algorithm 1: Assembly procedure

```
for each sub-domain S do
    for each element T of S do
        for each integration point with index p in T do
            Compute matrices K^{<p>}_{α,β|T} for all available combinations of α and β
            and accumulate the result to K_{α,β|T}
        Apply optional element level transformations on assembled element matrices K_{α,β|T}
        Accumulate all element matrices K_{α,β|T} to the respective indices J_{α|T} and J_{β|T} in K_G
```

One central point regarding the otherwise conventional algorithm 1 is that the innermost computation involves all weak form expressions and problem variables. This is in contrast to the alternative approach of composing stiffness matrices by superposition of pre-implemented PDE terms, which are computed separately. The approach followed here allows for optimizations across all PDE terms added to a model. Moreover, the optional element level transformation before the addition to the global matrix, allows for the implementation of advanced element types, involving local projections depending on the real element, such as locking-free MITC plate elements and hybrid high-order elements [4, 9].

4.2 Compilation of GWFL expressions and optimization

At this point, it is essential to achieve an efficient calculation at each integration point of the elementary matrices $K^{<p>}_{\alpha,\beta|T}$, or corresponding vectors in the assembly of first order terms. Performing any kind of text string interpretation of GWFL expressions at each integration point would of course be very inefficient. The text based description of a weak forms is therefore initially compiled into a sequence of optimized basic instructions that are later repeatedly executed for each integration point. Such a compilation step is implemented in different software projects based on different strategies, as reported e.g. in [13, 14] and [23, 31]. The compilation procedure implemented in GetFEM can be defined in terms of four steps, comprising

- parsing of expressions and transformations into an operation tree,
- semantic analysis and simplifications,
- symbolic differentiation of the term when necessary, and
- compilation into a sequence of basic instructions.

A simple single variable example will be considered in order to illustrate this procedure. Letting $u : \Omega \to \mathbb{R}^3$ denote the displacement field of an elastic solid Ω, the simplest constitutive law for large

deformation elasticity is the Saint Venant-Kirchhoff one, defining the second Piola-Kirchhoff stress tensor S as a linear function of the deformation tensor $\mathcal{E}$, in the form

$$S(\nabla u) = \lambda \mathrm{Tr}(\mathcal{E}) I + 2\mu\mathcal{E} \quad \text{with} \quad \mathcal{E} = (\nabla u + \nabla u^T + \nabla u^T \nabla u)/2,$$

where λ and μ are the Lamé coefficients. The computation of the residual vector for the discretized problem is based on the first order term

$$F_1(u; \delta u) = \int_\Omega ((I + \nabla u) S(\nabla u)) : \nabla \delta u \; d\Omega$$

where, the integrand can be expressed in GWFL as

```
((Id(meshdim)+Grad(u))*
 (0.5*lambda*Trace(Grad(u)+Grad(u)'+Grad(u)'*Grad(u))*Id(meshdim)
                          +mu*(Grad(u)+Grad(u)'+Grad(u)'*Grad(u)))):Grad(Test_u)
```
(E9)

with the special GWFL keyword `meshdim` denoting the problem space dimension d.

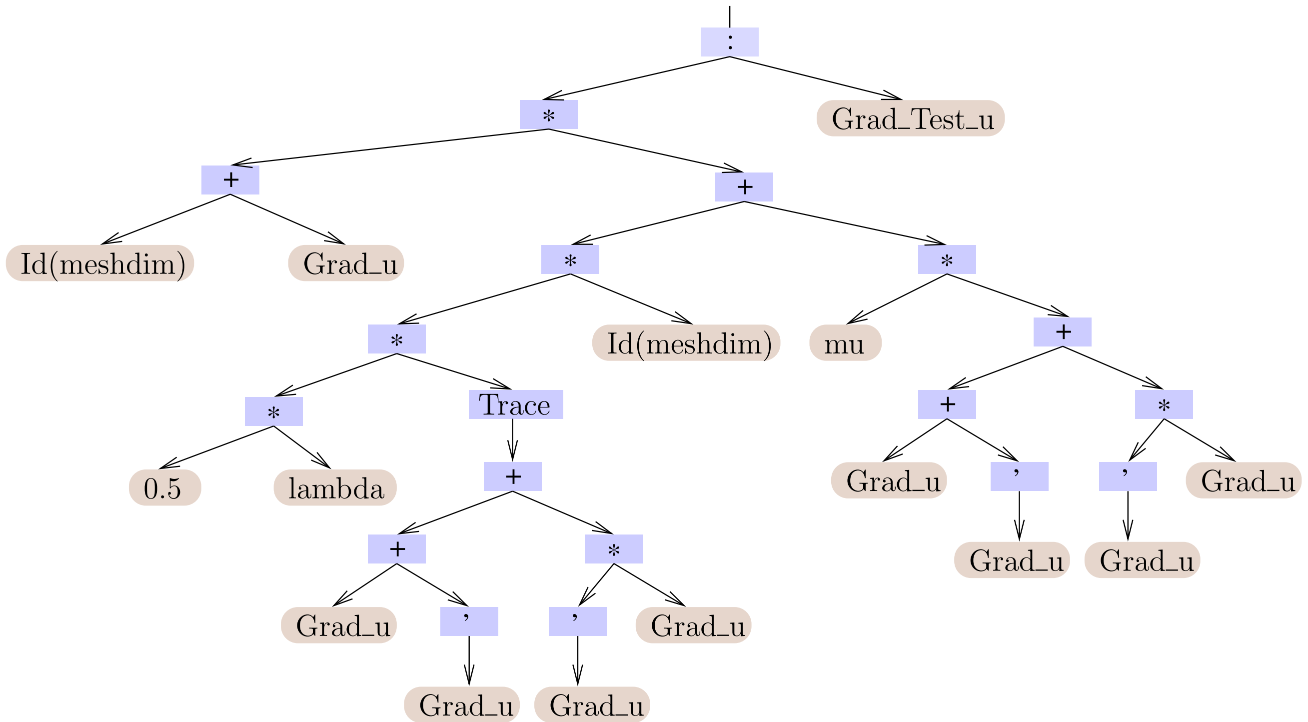


Figure 4: Operation tree for the elastostatic problem with the Saint Venant-Kirchhoff constitutive law.

The parsing step on this expression will then simply result in the operation tree shown in Figure 4. The subsequent semantic analysis step will enrich the tree with information needed for checking the validity of the operations and for carrying out simplifications. The first obvious simplification concerns the precomputation of subexpressions that only involve constants. In the present example, assuming a homogeneous material, the Lamé coefficients do not depend on the current integration point and a repeated evaluation of the product `0.5*lambda` is therefore superfluous. In the simplification phase after

the semantic analysis, all subtrees depending on constant data will be evaluated and substituted with the corresponding numerical result in the optimized tree.

Additionally, a hash value is assigned to each node of the tree, which depends on the node itself and its child nodes. This allows an inexpensive detection of identical parts of the tree based on a simple sorting of their hash values, [23]. For instance, in the tree of Figure 4, there are multiple occurrences of `Grad(u)` and two computations of `Grad(u)+Grad(u)'+Grad(u)'*Grad(u)`. After eliminating all detected redundancies, the processed tree will be converted to a (single sourced) directed acyclic graph (DAG), [20], shown in Figure 5 for the considered example. One optimization not visible in this figure but present in the actual implementation is the use of the basis functions gradients stored in tensor t^{16}_{ijk} by instruction M also in the calculation of the spatial gradient of u, computed and stored in t^2_{ij} by instruction A.

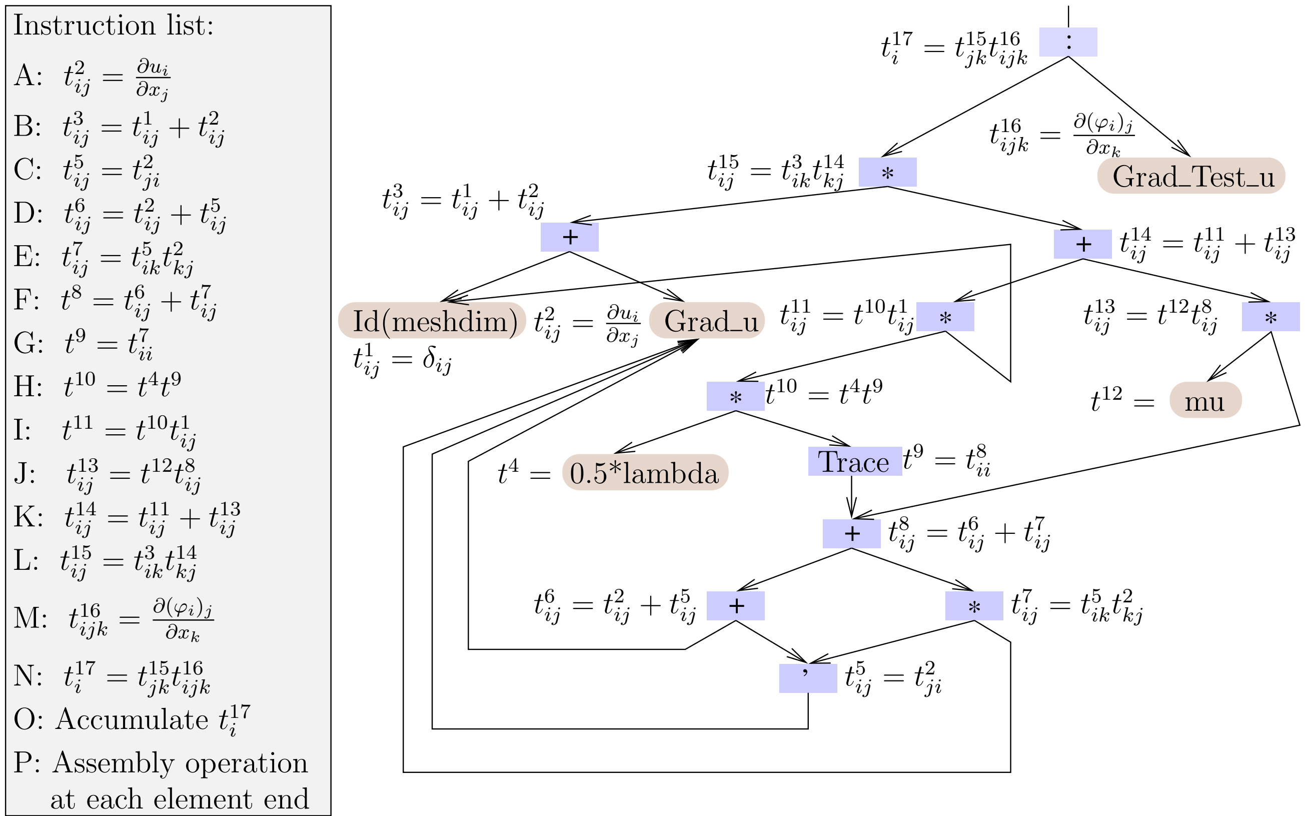


Figure 5: DAG obtained after the elimination of all duplicated subexpressions. The result of each node is a tensor with implicit summation on repeated indices assumed for brevity.

Some further optimizations would be possible by rearranging terms based on the commutative and distributive properties of some operations such as additions and multiplications. In the considered example one could for instance replace the subexpression

```
0.5*lambda*Trace(Grad(u)+Grad(u)'+Grad(u)'*Grad(u))*Id(meshdim)
```
(E10)

with

```
Trace(Grad(u)+Grad(u)'+Grad(u)'*Grad(u))*0.5*lambda*Id(meshdim)
```
(E11)

so that the diagonal matrix `0.5*lambda*Id(meshdim)` could be precomputed as a constant. Such rearranging optimizations are not implemented in GetFEM yet, hence the user is expected to provide all relevant expressions in an adequately factorized form, with constants grouped together, in order to avoid computational losses.

After the construction of the optimized DAG for a given expression, a symbolic differentiation step may be necessary if the order of the provided expression is lower than the order of the assembled term. For example, if a Jacobian matrix has to be assembled but the provided expression is only first order, this will be automatically differentiated to the corresponding second order term. In general, if necessary, the symbolic differentiation rules will be applied once or even twice to the already optimized DAG, representing the provided expression. Most of the GWFL operators include in their definitions either the corresponding symbolic differentiation rule or an equivalent lower level numerical implementation of it. For instance, the implementation of the multiplication operator `*` includes the symbolic differentiation rule $\partial(fg) = \partial f g + f \partial g$ but the derivative of the matrix inverse operator `Inv(A)` is implemented as a C++ function instead of being written out according to the corresponding symbolic rule $\partial(A^{-1}) = -A^{-1}(\partial A)A^{-1}$. Whenever an expression has to be differentiated, the resulting DAG will be again analyzed semantically and optimized, in order to mitigate the fairly high complexity that may arise from the application of the chain rule.

Once the optimized DAG of the term to be assembled has been generated, with or without any intermediate differentiation, the last step consists in compiling a sequence of basic instructions corresponding to each node of the DAG. A tensor of appropriate dimensions is also associated to each of these instructions for storing the expected result. For the earlier discussed example for instance, the generated instruction sequence is illustrated in Figure 5. It should be noted that the described assembly procedure includes the vectorization with respect to the active degrees of freedom in the current element, discussed in the previous subsection. The tensor t^{16}_{ijk} in Figure 5, for example, is a rank three tensor just because of this vectorization, with the first tensor index i actually denoting the position in the set of active basis functions $\varphi_{u|T}$ in the current element T. This extra tensor dimension propagates to the result tensor t^{17}_{i}, the elementary vector, which holds residual values to be transferred to all degrees of freedom in the global residual vector that are active in the current element. An extra instruction O performs the weighted summation over all integration points in the current element, and a final instruction P, executed only at the last integration point per element, performs the actual assembly, adding the computed elementary vector to the correct indices in the global residual vector.

All tensors involved in the compiled instruction list are in principle of constant size, independent of the current element and integration point. This allows to allocate all tensors involved in the assembly procedure only once and avoid any memory reallocation during the actual assembly. This is in general true for uniform finite element meshes with a constant number of degrees of freedom per element. If different finite element types are mixed in the same mesh, an additional instruction is added for resizing any tensors that depend on the number of degrees of freedom in the current element. In that sense, for meshes with multiple finite element types it is computationally favorable to number the elements clustered in groups of similar element types in order to minimize the need for reallocations.

It is important here to briefly describe the actual C++ implementation of the final compilation of the processed expression into a sequence of basic instructions. The compilation algorithm produces a list of function calls which correspond to each node crossed when transversing the DAG from the leafs (sinks) to the root (source). To achieve a computationally efficient implementation, the function calls are without passing of arguments of any kind. Instead, the necessary memory allocation and wiring of input and

output quantities of these function calls occurs in an appropriate structure generated by the expression compilation algorithm. The building block for this structure is the instruction class conceptually shown in Figure 6, which implements the function call in its `exec()` method and occupies memory only for the output tensor and possibly some internal data. It will typically also include C++ references to output tensors of other instructions in the DAG for giving access to all required input. The compilation procedure consists in constructing a list with instances of pre-implemented instruction classes corresponding to each node in the DAG and initialize these instances appropriately. Once the list is produced, the `exec()` method of all instruction instances in the list will be successively called at each integration point. The DAG structure ensures that when the `exec()` method of an instruction is called, all necessary input is already available from the prior execution of its dependencies.

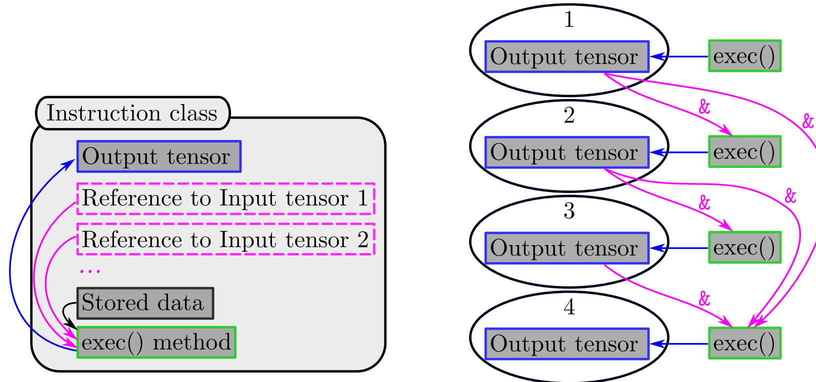


Figure 6: Structure of a typical instruction class implementing a node in the GWFL DAG and example of compiled list of instruction class instances.

Based on the previous description, the term compilation in this context is to be understood in the sense of producing a sequence of function calls to actually precompiled basic instructions acting on an appropriately initialized data structure. This is in contrast to generation and compilation of C or C++ code involving a call to an external compiler, as in FEniCS [23]. Being able to do the compilation of an expression at runtime offers great flexibility and allows to easily and efficiently update the expression between load steps to account for example for possibly evolving terms. The use of C++ references for achieving the necessary flow of data between instructions avoids any redundant memory allocations or copying of values from one instruction to another. A single call per instruction to its `exec()` method is actually the only well defined computational penalty of this approach, compared to the low level compilation approach. The cost of calling a C++ virtual method without arguments is comparable to a simple memory access operation and it is hence negligible compared to the time spent inside the function. This is is especially the case for the assembly of second order terms, where the computations performed inside each instruction are comparably heavier. A GWFL-based reimplementation of a large collection of PDE terms in GetFEM, has led to significant performance gains against most of the manual implementations. Minor losses were only observed for a combination of very simple PDE terms and linear elements.

The example DAG of Figure 5 represents a very simple situation with a single variable and a single assembly term. For coupled problems and more advanced constitutive laws, DAGs can become much more complicated. In order to approach the absolutely minimum number of necessary computations, operation

trees of different terms to be evaluated on the same elements and with the same integration method are combined together and the search for repeated subexpressions and corresponding simplifications are made on the combined set of operation trees. The resulting single DAG can be very complex, but this treatment ensures that no subexpression repeated in different terms, will be evaluated more than once. Obviously, the more complex the considered problem, the largest the gain from this optimization procedure compared to a traditional implementation where contributions of different PDE terms are assembled independently.

As a final comment, we need to underline the importance of the simplicity of the proposed generic weak form language. Although the language includes a very comprehensive set of operators, its syntax is extremely simple, much closer to a mathematic language than a programming language. The lack for instance of if-conditions, loops or local variables as part of the language is a conscious choice, very essential not only for maintaining a certain simplicity in the language and its implementation but also for achieving high efficiency based on conceptually rather simple optimizations.

4.3 Performance

High level and very universal modeling environments are normally associated with inferior performance both in terms of memory usage and computational efficiency. However, the above described optimizations show a potential for computational gains through a high level automated modeling approach, which are hard to match through a manual low level implementation. Of course, such gains from avoiding repeated calculations of intermediate quantities will only become decisive for the overall computational performance if 1) the cost of parsing and high-level compilation of GWFL expressions is small and scalable with increasing expression size, and 2) the remaining parts of the implementation have similar performance to more specific low level implementations.

Parsing of GWFL strings is implemented in GetFEM based on standard C++ strings and a usual recursive algorithm of approximately linear complexity. Recorded timings of the expression parsing, generation of the computation DAG, c.f. Figure 5, and eventual derivation of higher order weak forms, have demonstrated that the computational cost of the GWFL operations, performed once per assembly, is far from becoming a bottleneck.

Regarding the non-GWFL part of GetFEM, the underlying semi-automated finite element infrastructure relies heavily on lazy, i.e. on demand, computations and caching of intermediate results, to achieve similar performance with purely manual implementations. To mention geometric transformations as an example, GetFEM uses a memory pool to store and retrieve values, gradients or Hessians of relevant shape functions, evaluated only once at all involved integration points. As a general comment, caching of intermediate results favors computational efficiency at the cost of memory usage. On the other side, reusing objects by means of a memory pool is a mechanism which both favors memory and computational efficiency and it can actually lead to computational gains also compared to manual implementations, in which case reusing of results between very remote parts of the code is harder to achieve.

For strongly nonlinear and highly coupled problems that the presented solution is meant for, residual vector and Jacobian matrix assemblage can comprise a quite significant computational load, compared to the solution of linear systems. Fortunately, assemblage is trivially parallelizable. GetFEM includes both OpenMP and MPI parallelizations of the assembly of GWFL expressions, based on partitioning of the computational mesh with METIS [16] and storage of these partitions in mesh regions, c.f. subsection 3.1.

4.4 Interpolate transformations

Interpolate transformations were presented in subsection 2.7 as a major feature that endows the GWFL with considerably extended expressive capabilities. Apart from the possibility of defining interpolate transformations symbolically using GWFL expressions, explained in subsection 2.7, the GWFL implementation in GetFEM allows to define custom interpolate transformations programmatically by overloading a C++ abstract base class described in Figure 7.

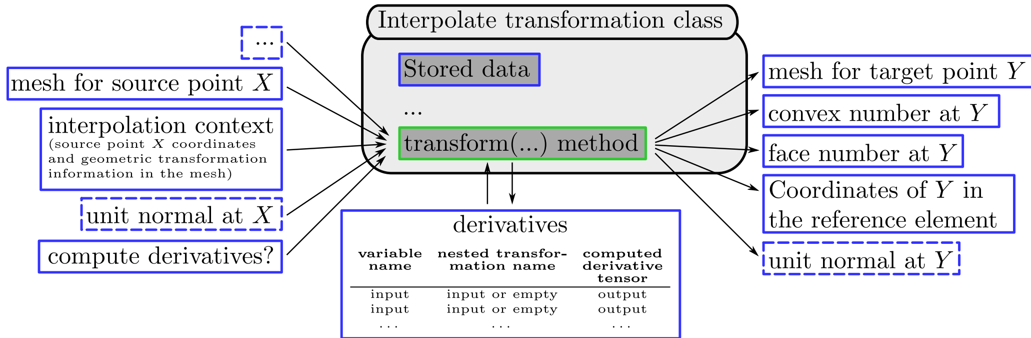


Figure 7: Interpolate transformation class archetype (`virtual_interpolate_transformation` base class).

In the most general case, the implementation of the `transform` method of this class will for a given point X return information about the transformed point $\mathcal{Y}(X)$, but it will also upon request return numerical results for all derivatives of the transformation required in Eq. (10). Derivatives $\partial\mathcal{Y}/\partial u_i$ with respect to any variable u_i can be requested by including the name of the variable u_i in the derivatives input/output argument of the `transform` method. Upon execution of the method, appropriate tensors will be evaluated and stored in the same argument for each requested derivative. Moreover, if a variable u_i is used in the considered interpolate transformation through another interpolate transformation, the application of the chain rule for nested transformations can be requested by a corresponding entry in the passed derivatives argument with a non-empty name for the nested transformation.

The programming of the computation of derivatives for a transformation is usually the most work-intensive task in implementing a custom interpolate transformation. The GWFL expression based transformations, raytracing and neighbor element transformations, discussed in subsection 2.7, are all derived from the archetype shown in Figure 7.

5 Examples

This section presents two representative numerical models implemented with the help of GWFL, that demonstrate modeling techniques with relevance for multiphysics problems. First, a relatively simple continuum mechanics problem is solved, mainly for showing the expressive power and compactness of the proposed language. Subsequently, a more advanced multiphysics example with several coupled variables demonstrates the versatility of GWFL in accounting for complex couplings between different physics. Code excerpts are provided for the two models implemented in Python in a total of approximately 110 and 160 lines, respectively.

5.1 Hyperelastic membrane and follower loads

For the first example, a circular membrane is considered, fixed at its circumference and subjected to an incremented pressure on one side. The heavily stretched membrane at increased pressure, will also become thinner, leading to significant thickness variations. Beyond some point, deformations will localize to the most thinned region causing the membrane to burst. The maximum pressure before this instability occurs defines the pressure capacity of the membrane.

This is a rather simple problem for moderate loads, but significant challenges occur when the onset of localization is approached. In such high load situations, a very accurate membrane element is required, linked to a robust underlying material model suitable for very large strains. Moreover, a numerical continuation scheme is required for tracking the process of localization.

In its reference configuration, the membrane of this example is considered planar and perpendicular to the z-axis. A displacements field u is considered to describe the overall shape of the deformed membrane with respect to its reference configuration. Moreover, if the initial normal unit vector on the membrane is mapped to a vector n in the deformed configuration, it is easy to show that the deformation state of each point of the membrane can be represented by a deformation gradient matrix in the form

$$F = \begin{bmatrix} 1+u_{x,x} & u_{x,y} & n_x \\ u_{y,x} & 1+u_{y,y} & n_y \\ u_{z,x} & u_{z,y} & n_z \end{bmatrix}$$

The vector field n is considered here as an additional field with 3 components to solve for, apart from the displacements field u. Its magnitude essentially expresses the actual thickness of the membrane relative to its initial thickness

Given the aforementioned expression for the deformation gradient F as a function of ∇u and n, any hyperelastic material law can be defined in terms of F. In the present example, a neo-Hookean material is considered, according to the strain energy density

$$W(\nabla u, n) = \frac{\kappa}{2}\left(\ln|F|\right)^2 + \frac{\mu}{2}\left(|F|^{-2/3}\,\|F\|^2 - 3\right),$$

with κ and μ respectively denoting the initial bulk and shear moduli. By using W as the integrand G_0 of a zero order term F_0 according to Eq. (1), involving the two field variables u and n, the kinematics and constitutive behavior of the considered membrane are fully defined. The deformed state of the membrane in equilibrium can then be easily found by solving the strain energy minimization problem on the functional F_0.

It only remains to provide a weak form term for the applied load on the membrane. The work conjugate tractions to the displacements field u for different kinds of follower loads are summarized in Table 4. In the present case, the applied pressure on one side of the membrane acts normal to the deformed membrane and it is defined per area of the deformed membrane, so that the expression in the upper right corner of the table will be used. For a membrane initially lying in the xy-plane with the overpressure on its bottom side, one can use

$$N = (0, 0, -1)^T \quad \text{and} \quad q = \lambda\, p_{\max},$$

where λ is a scalar load multiplier for performing numerical continuation and $p_{\max}$ a user defined maximum pressure. A first order term according to Eq. (3) can then be used for representing the applied load, with the integrand

$$G_1(\nabla u, n\ ;\ \delta u) = \left(\lambda p_{\max}|F|F^{-T}N\right)\cdot\delta u$$

Table 4: Follower load q on a surface as a function of the deformation gradient F, and the normal and tangent vectors N and T in the reference configuration.

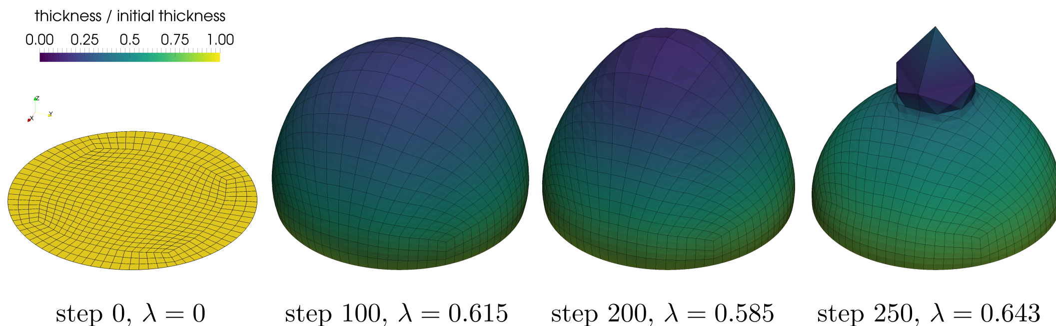
	q per undeformed area	q per deformed area
q acting normal to the surface	$q\dfrac{F^{-T}N}{\|F^{-T}N\|}$	$q\|F\|F^{-T}N$
q acting tangent to the surface	$q\dfrac{FT}{\|FT\|}$	$q\|F\|\,\left\|F^{-T}N\right\|\,\dfrac{FT}{\|FT\|}$

Figure 8 presents simulation results at different load steps, showing the actual thickness distribution over the deformed membrane. The numerical continuation algorithm has tracked the maximum applied load of the membrane corresponding to $\lambda = 0.6912$ at load step 50 and could further simulate the localization phase of the deformation up to the final bursting of the membrane.

For the implementation of this model in GetFEM, the mesh shown in Figure 8 was used with 9-node quadratic quadrilateral elements for both unknown fields u and n. The model was implemented in Python and the most essential parts of the implementation are provided in the code listings below. The mesh generation, the definition of a finite element space and the definition of an appropriate integration method are done with

```
mesh = gf.Mesh("import", "structured_ball",
               "GT='GT_QK(2,2)';ORG=[0,0];SIZES=[50];NSUBDIV=[10,3];SYMMETRIES=0")
mesh.transform([[1,0],[0,1],[0,0]]) # convert the 2D mesh to 3D
mesh.set_region(DIR_BOUNDARY, mesh.outer_faces(2))
mf = gf.MeshFem(mesh, 3) # vector FEM with 3 components per node
mf.set_classical_fem(2)  # second order Lagrangian FEM
mim = gf.MeshIm(mesh, 5) # degree 5 integration method, i.e. 3x3 points
```

Then a model object is created, all relevant variables and problem constants are defined and the two weak form terms representing the aforestated hyperelastic strain energy function and the follower load, are added to the model. Last, a homogeneous Dirichlet condition is imposed on the displacements at the external circumference of the membrane.



step 0, $\lambda = 0$ | step 100, $\lambda = 0.615$ | step 200, $\lambda = 0.585$ | step 250, $\lambda = 0.643$

Figure 8: Deformation of a circular membrane under pressure, with diameter of 100 mm, initial thickness of 2 mm and, Young's modulus 50 MPa, Poisson ratio 0.4, and $p_{\max} = 1$ MPa.

```
md = gf.Model("real")
md.add_fem_variable("u", mf)            # displacements variable
md.add_fem_variable("n", mf)            # deformed normal vector variable
md.set_variable("n", md.interpolation("[0,0,1]", mf, -1))
md.add_initialized_data("gamma", 0.)    # numerical continuation load multiplier
md.add_initialized_data("pmax", pmax)   # maximum pressure
md.add_initialized_data('kappa', kappa) # initial bulk modulus
md.add_initialized_data('mu', mu)       # initial shear modulus
md.add_initialized_data('H', H)         # membrane thickness

md.add_macro("F", "[1,0,0;0,1,0;0,0,0]+Grad(u)+n@[0,0,1]") # deformation gradient
md.add_nonlinear_term(mim, "H*0.5*kappa*sqr(log(Det(F)))+"
                           "H*0.5*mu*(pow(Det(F),-2/3)*Norm_sqr(F)-3)")
md.add_nonlinear_term(mim, "gamma*pmax*Det(F)*((Inv(F)'*[0,0,-1]).Test_u)")
md.add_Dirichlet_condition_with_multipliers(mim, "u", mf, DIR_BOUNDARY)
```

The subsequent call to the numerical continuation solver within a corresponding loop is omitted here in the interest of space, but these remaining steps are also implemented in a comparably compact manner. Just to give an impression about the use of GWFL for preparing post-processing output, Von Mises stresses are calculated and exported with

```
VM = md.local_projection(mim, "sqrt(1.5)*Norm(mu*pow(Det(F),-5./3.)"
                              "*Deviator(Left_Cauchy_Green(F)))", mfout)
mfout.export_to_vtk("membrane_VM.vtk", mfout, VM, "Von Mises Stress")
```

where `mfout` is a discontinuous Langrangian finite element space, used only for post-processing.

5.2 Phase field model of hydrogen assisted crack propagation

Hydrogen assisted crack propagation is a typical multiphysics problem, involving elasticity with fracture damage and diffusion of hydrogen. There is a strong bidirectional coupling between the two subproblems with the elastic stresses affecting the diffusion of hydrogen and at the same time the hydrogen concentration having an impact on the damage behavior. Here, we present a reimplementation and extension of a model from the literature, with the help of GWFL.

Considering a solid with bulk modulus κ and shear modulus μ, subjected to small strains due to a displacements field u, its elastic energy density function is

$$\psi_0(\nabla u) = \frac{\kappa}{2} \left(\nabla \cdot u\right)^2 + \mu \left\|\mathrm{Dev}\left(\nabla_{\mathrm{s}} u\right)\right\|^2 .$$

The fracture phase field model proposed by [28], introduces a damage field variable d, which leads to a degradation in stiffness as it increases from zero to one. The evolution of the damage variable is governed by the critical energy release rate parameter $\mathcal{G}_c$ and the length scale parameter l. The model proposed by [27] additionally introduces the hydrogen concentration C as an unknown field, and accounts for a dependence of the fracture parameter $\mathcal{G}_c$ on C. The diffusion of hydrogen in the material is coupled to the mechanical stresses, as hydrogen is attracted to regions of lower hydrostatic pressure p.

In order to avoid the occurrence of second order spatial derivatives with respect to the displacement field u in the hydrogen diffusion equation, which are difficult to treat numerically, the hydrostatic pressure p is considered as an additional unknown field. This results in a model with the four unknown fields u, d, p and C, in contrast to the three fields used in [27]. Moreover, we add an inertia term based on the material density constant ρ.

Following the time discretization proposed in [28], the governing equations for the time step $dt_n = t_n - t_{n-1}$ can be cast into a weak form consisting of four first order terms

$$\int_\Omega G_{1a}(d, \nabla u \; ; \; \nabla \delta u) \; + \; G_{1b}(d, \nabla u, \nabla d \; ; \; \delta d, \nabla \delta d) \\ + \; G_{1c}(p, d, \nabla u \; ; \; \delta p) \; + \; G_{1d}(C, \nabla p, \nabla C \; ; \; \delta C, \nabla \delta C) \; d\Omega = 0 \quad \forall \; \delta u, \delta p, \delta d, \delta C.$$

The three integrands G_{1a}, G_{1b} and G_{1d}, are adopted from [27], with only small modifications. By including inertia forces, with acceleration approximated recursively in a backward Euler sense, the first integrand becomes

$$G_{1a} = \rho\big((u - u_{n-1})/dt_n - \dot{u}_{n-1}\big)/dt_n \cdot \delta u + g(d) \left(\kappa \, (\nabla \cdot u) \, I + \mu \, \mathrm{Dev}(\nabla_{\mathrm{s}} u)\right) : \nabla \delta u, \tag{20}$$

where u_{n-1} and $\dot{u}_{n-1} = (u_{n-1} - u_{n-2})/dt_{n-1}$, are displacements and velocities at the previous time instant t_{n-1}, and $g(d) = (1-d)^2 + k_1$ is degradation function with a small positive constant k_1.

Following the model from the literature, the second integrand is

$$G_{1b} = -2(1-d)\mathcal{H}_n(\nabla u) \, \delta d + \frac{\mathcal{G}_c(C)}{l} \left(d \, \delta d + l^2 \, \nabla d \cdot \nabla \, \delta d\right), \tag{21}$$

with

$$\mathcal{H}_n = \max\left(\psi_0(\nabla u), \mathcal{H}_{n-1}\right) \quad \text{and} \quad \mathcal{G}_c = \left(1 - \chi \frac{C}{C + c_1}\right) \mathcal{G}_{c0},$$

where $\mathcal{G}_{c0}$ is the critical energy release rate in the hydrogen free material and χ and c_1 are additional material parameters.

The pressure variable p can be defined as equal to $-g(d) \, \kappa \, \nabla \cdot u$ through the weak form integrand

$$G_{1c} = (p + g(d) \, \kappa \, \nabla \cdot u) \, \delta p, \tag{22}$$

and finally, the steady state form of the hydrogen diffusion equation is expressed through

$$G_{1d} = (\nabla C + c_2 \, C \, \nabla p) \cdot \nabla \delta C, \tag{23}$$

where c_2 is a material parameter describing the attraction of hydrogen due to pressure gradients.

The following code excerpts assume a 2D mesh defined in GetFEM, along with appropriate finite element spaces `mfu`, `mfd`, `mfp` and `mfC` corresponding to the four unknown fields. Quadratic, 9-node quadrilateral elements are used for `mfu` and `mfd`, while linear 4-node elements are used for `mfp` and `mfC`. Moreover, `mim4` and `mim9` represent integration methods with 4 and 9 integration points per element, respectively, while `mimd4` and `mimd9` are so called `mesh_im_data` objects for storing scalar data on the integration points of the two aforementioned integration methods. In this context, all problem unknowns, data and state variables can be added to a GetFEM model `md` with

```
md.add_fem_variable("u", mfu)    # displacements field
md.add_fem_variable("d", mfd)    # fracture phase field
md.add_fem_variable("p", mfp)    # hydrostatic pressure field
md.add_fem_variable("C", mfC)    # hydrogen concentration field
md.set_variable("C", C0*np.ones(mfC.nbdof()))
md.add_fem_data("u_prev", mfu)
md.add_fem_data("v_prev", mfu)
md.add_im_data("psi0_max", mimd9)
# Definition of constants (Dt, kappa, mu, rho, Gc0, l, C0, chi, c1, c2)
md.add_initialized_data(...name, ...value)
```

The state variables `u_prev` and `v_prev`, corresponding to u_{n-1} and $\dot{u}_{n-1}$, are stored in the same finite element space as u, while the state variable `psi0_max`, corresponding to the maximum reference energy $\mathcal{H}_{n-1}$, is stored on all relevant integration points. The constant `C0` is simply used as an initial and boundary condition value for the hydrogen concentration variable C. The presented governing equations can then be implemented in GWFL as compact as listed in the following code excerpt.

```
md.add_linear_term(mim9, "rho/Dt*((u-u_prev)/Dt-v_prev).Test_u")
md.add_macro("degradation", "sqr(1-d)+1e-7")
md.add_macro("deveps", "Sym(Grad(u))-Div(u)/3*Id(2)")
md.add_macro("psi0", "(0.5*kappa*sqr(Div(u))+mu*Norm_sqr(deveps))")
md.add_macro("Gc", "(1-chi*C/(C+c1))*Gc0")
md.add_nonlinear_term(mim9, "degradation*(kappa*Div(u)*Id(2)+2*mu*deveps):Grad(Test_u)")
md.add_nonlinear_term(mim9, "(-2*(1-d)*max(psi0_max,psi0)*Test_d"
                            "+Gc*(d/l*Test_d+l*Grad(d).Grad(Test_d)))")
md.add_nonlinear_term(mim4, "(p+degradation*kappa*Div(u))*Test_p")
md.add_nonlinear_term(mim4, "(Grad(C)+c2*C*Grad(p)).Grad(Test_C)"
                            "+1e3*pos_part(2*d-1)*(C-C0)*Test_C")
```

The deviatoric strain definition in the macro `deveps`, assumes a 2D problem domain and plane strain conditions. It should also be noted that the provided implementation extends the diffusion Eq. (23) from the literature, by imposing a reference hydrogen concentration value C_0 in all damaged regions of the domain, characterized by $d > 0.5$, through the extra penalization term

$$k_2 \left\langle 2d - 1 \right\rangle (C - C_0)\, \delta C,$$

with k_2 being a moderately large positive penalization factor.

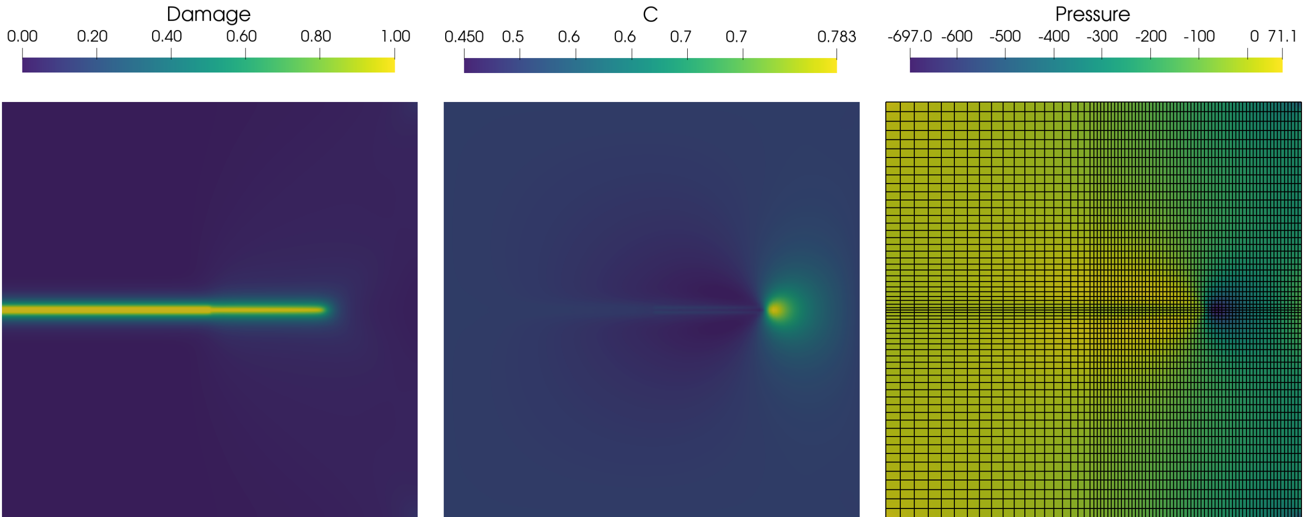


Figure 9: Obtained fracture phase field (left), hydrogen concentration [ppm$_\mathrm{w}$] (center) and hydrostatic pressure [MPa] (right), at an imposed average vertical strain of 0.00266. Reference hydrogen concentration of 0.5 [ppm$_\mathrm{w}$], strain rate of $2 \cdot 10^{-4}$ [1/s], density $\rho = 8 \cdot 10^{-9}$ [t/mm^3], and remaining model parameters as in [27].

The definition of Dirichlet boundary conditions on u and C is skipped here, as it is rather trivial as

shown in the first example. To complete the presentation of all essential parts of the implementation, the code for updating the three state variables for the next time step is listed below.

```
md.set_variable("u_prev", md.variable("u"))
md.set_variable("v_prev", md.interpolation("(u-u_prev)/Dt", mfu))
md.set_variable("psi0_max", md.interpolation("max(psi0_max,psi0)", mimd9))
```

To close this example, Figure 9 shows representative results for the fields d, C and p in a single edge notched specimen under mode I loading. Instead of defining the initial crack as a discontinuity in the computational mesh, it is represented by initializing the state variable psi0_max, corresponding to $\mathcal{H}_{-1}$, to some large value in all elements in a given region. Representative timings for single residual vector and Jacobian matrix assemblies as well as for a single non-symmetric linear system solution with MUMPS [2] are given in Table 5.

Table 5: Assembly and linear solution timings for example 2 on Intel® Xeon® CPU E5-2660 v3 (2.60GHz).

Mesh	Degrees of freedom		Assembly		Linear
size	Displacement	Total	Residual	Jacobian	solution
80×60	38962	69249	80 ms	350 ms	1700 ms
160×120	154722	272889	320 ms	1400 ms	8500 ms

6 Concluding remarks

Numerical modeling for scientific applications follows a general trend in software towards a less error-prone development through less code duplication and a higher level of automation. In this context, this paper has provided software design and implementation details for a highly automated finite element modeling framework. As the main ingredient for this automation, a proposed generic weak form language has been presented both from a theoretical and an implementation perspective, focusing on its potential for combining a high level of automation with computational efficiency.

The value of the introduced ASCII text based language as a suitable universal format for formulating arbitrarily coupled systems of partial differential and algebraic equations has been discussed. More specifically, the interpolate transformation mechanism, incorporated in the language, has received special attention due to its significant contribution to the achieved expressive power of the proposed language. A wide range of numerical methods such as mortar and unilateral contact as well as discontinuous Galerkin methods can be implemented with the help of this mechanism.

At a more technical level, the runtime compilation of the generic weak form language expressions has been explained and conceptually compared to alternative solutions such as just-in-time compilation, highlighting the possible computational efficiency gains. Other relevant innovations and software architecture decisions regarding the overall GetFEM framework that implements the proposed language have also been presented and justified.

The included examples have indicated the potential of the proposed automation for gains in coding and debugging time. Certainly, different types of models and modeling needs exist beyond the presented examples and language functionalities. However, there is also a set of more advanced features already

integrated in GetFEM/GWFL but not covered here. These features, including levelset and XFEM capabilities as well as static condensation of internal variables based on local equations described in GWFL, will in the future demonstrate the extendibility of GWFL to even more complex modeling scenarios.

Appendix

Table 6: Naming of common geometric transformations and corresponding reference elements in GetFEM.

`GT_PK(p,k)`	Simplicial element in $\mathbb{R}^p$ of degree k. E.g. `GT_PK(3,1)` is a linear tetrahedral element.
`GT_QK(p,k)`	Hypercube element in $\mathbb{R}^p$ of degree k. E.g. `GT_QK(3,2)` is a 27-node hexahedral element.
`GT_PRISM(p,k)`	Prismatic element in $\mathbb{R}^p$ of degree k. E.g. `GT_PRISM(3,1)` is a 6-node wedge element.
`GT_PYRAMID(k)`	Quadrilateral base pyramids in 3D, either linear or quadratic ($k = 1, 2$).
`GT_Q2_INCOMPLETE(p)`	Quadratic serendipity parallelepiped elements in 2D (p=2) and 3D (p=3), respectively corresponding to 8-node quadrilateral and 20-node hexahedral elements.
`GT_PRODUCT(a,b)`	Tensor product of transformations. E.g. the product of a linear triangular element and a line element `GT_PRODUCT(GT_PK(2,1),GT_PK(1,1))` is equivalent to `GT_PRISM(3,1)`.

Table 7: Naming of common numerical integration methods in GetFEM.

`IM_GAUSS1D(k)`	Gauss-Legendre quadrature rule on a 1D element with $k/2 + 1$ points, integrating polynomials of degree k exactly.
`IM_TRIANGLE(k)`	Integration method of order k (up to 13) on a triangle.
`IM_QUAD(k)`	Integration method of order k (up to 17) on a quadrilateral.
`IM_TETRAHEDRON(k)`	Integration method of order k (up to 8) on a tetrahedron.
`IM_PYRAMID(IM)`	Transforms a hexahedron into a pyramid integration method.
`IM_STRUCTURED_COMPOSITE(IM,s)`	Refines the integration method `IM` using `s` subdivisions.

Table 8: Naming of common finite element types in GetFEM.

Name	τ-equiv.	Vector	Element description
`FEM_PK(n,k)`	Yes	No	Lagrange of degree k on a n-dimensional simplex (segment, triangle, tetrahedron, ...).
`FEM_QK(n,k)`	Yes	No	Lagrange of degree k on a segment, quadrilateral, hexahedron, ...
`FEM_HERMITE(n)`	No	No	Hermite on a n-dimensional simplex.
`FEM_ARGYRIS`	No	No	Argyris on a triangle. Conformal C^1-element, polynomial of degree 5.
`FEM_PYRAMID_QK(k)`	Yes	No	Lagrange of degree $k = 1$ or 2 on a pyramid with rational shape functions on the reference element.
`FEM_RT0(n)`	No	Yes	Raviart-Thomas vector element of lowest order on a n-dimensional simplex.
`FEM_NEDELEC(n)`	No	Yes	Nedelec vector element of order 1 on a n-dimensional simplex.
`FEM_HTC_TRIANGLE`	No	No	Hsieh-Clough-Tocher on a triangle. Composite element, piecewise polynomial of degree 3.
`FEM_PRODUCT (FEM1,FEM2)`	Yes	-	Tensor product of two (τ-equivalent) elements. E.g. `FEM_QK(2,1)` can be written as `FEM_PRODUCT(FEM_PK(1,1),FEM_PK(1,1))`.

References

[1] G. Amberg, R. Tönhardt, and C. Winkler. Finite element simulations using symbolic computing. *Math. Comput. Simulat.*, 49(4):257 – 274, 1999.

[2] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. A.*, 23(1):15–41, 2001.

[3] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM T. Math. Software*, 33(4):24/1–27, 2007.

[4] K.-J. Bathe and F. Brezzi. A simplified analysis of two plate bending elements — the mitc4 and mitc9 elements. In G. N. Pande and J. Middleton, editors, *Numerical Techniques for Engineering Analysis and Design*, pages 407–417, Dordrecht, 1987. Springer Netherlands.

[5] M. Bergot, G. Cohen, and M. Duruflé. Higher-order finite elements for hybrid meshes using new nodal pyramidal elements. *J. Sci. Comput.*, 42(3):345–381, 2010.

[6] D. Boffi, F. Brezzi, and M. Fortin. *Mixed finite element methods and applications*. Springer Series in Computational Mathematics. Springer, 2013.

[7] P. G. Ciarlet. The finite element method for elliptic problems. *Classics in Applied Mathematics*, 40:1–511, 2002.

[8] A. Dhooge, W. Govaerts, and Y. A. Kuznetsov. Matcont: A matlab package for numerical bifurcation analysis of odes. *ACM T. Math. Software*, (31):141 – 164, 2003.

[9] D. Di Pietro and A. Ern. A hybrid high-order locking-free method for linear elasticity on general meshes. *Computer Methods in Applied Mechanics and Engineering*, 283:1 – 21, 2015.

[10] V. Domínguez and F.-J. Sayas. Algorithm 884: A simple Matlab implementation of the Argyris element. *ACM T. Math. Software*, 35(2):16/1–11, 2008.

[11] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. Numer. Meth. Eng.*, 79(11):1309–1331, 2009.

[12] R. D. Graglia and I.-L. Gheorma. Higher order interpolatory vector bases on pyramidal elements. *IEEE T. Antenn. Propag.*, 47:775–782, 1999.

[13] F. Hecht. C++ tools to construct our user-level language. *ESAIM-Math. Model. Num.*, 36(5):809–836, 2002.

[14] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[15] M. Heil and A. L. Hazel. oomph-lib – An object-oriented multi-physics finite-element library. In Hans-Joachim Bungartz and Michael Schäfer, editors, *Fluid-Structure Interaction*, pages 19–49. Springer Berlin Heidelberg, 2006.

[16] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[17] R. Kirby. A general approach to transforming finite elements. *SMAI Journal of Computational Mathematics*, 4:197–224, 2018.

[18] J. Korelc. Automatic generation of finite-element code by simultaneous optimization of expressions. *Theor. Comput. Sci.*, 187(1):231 – 248, 1997.

[19] J. Korelc. Automation of primal and sensitivity analysis of transient coupled problems. *Comput. Mech.*, 44(5):631–649, 2009.

[20] D. C. Kozen. *The design and analysis of algorithms.* Springer, 1992.

[21] T. Ligurský and Y. Renard. A continuation problem for computing solutions of discretised evolution problems with application to plane quasi-static contact problems with friction. *Comput. Method. Appl. M.*, (280):222 – 262, 2014.

[22] T. Ligurský and Y. Renard. Bifurcations in piecewise-smooth steady-state problems: abstract study and application to plane contact problems with friction. *Comput. Mech.*, 56(1):39 – 62, 2015.

[23] A. Logg, K. A. Mardal, and G. N. Wells. Automated solution of differential equations by the finite element method. *Lect. Notes Comp. Sci.*, 84:1–736, 2012.

[24] A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM T. Math. Software*, 37(2):20/1–28, 2010.

[25] K. Long, P. T. Boggs, and B. G. van Bloemen Waanders. Sundance: High-level software for PDE-constrained optimization. *Sci. Programming-Neth.*, 20(3):293–310, 2012.

[26] K. Long, R. Kirby, and B. G. van Bloemen Waanders. Unified embedded parallel finite element computations via software-based frechet differentiation. *SIAM J. Sci. Comput.*, 32(6):3323–3351, 2010.

[27] E. Martínez-Pañeda, A. Golahmar, and C. F. Niordson. A phase field formulation for hydrogen assisted cracking. *Comput. Method. Appl. M.*, 342:742 – 761, 2018.

[28] C. Miehe, M. Hofacker, and F. Welschinger. A phase field model for rate-independent crack propagation: Robust algorithmic implementation based on operator splits. *Comput. Method. Appl. M.*, 199(45-48):2765–2778, 2010.

[29] K. Poulios and Y. Renard. An unconstrained integral approximation of large sliding frictional contact between deformable solids. *Computers & Structures*, 153:75–90, 2015.

[30] C. Prud'homme. A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations. *Sci. Programming-Neth.*, 14(2):81–110, 2006.

[31] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T.T. McRae, G. T. Bercea, G. R. Markall, and P. H.J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *Acm Transactions on Mathematical Software*, 43(3):2998441, 2016.

[32] K. Rupp. Symbolic integration at compile time in finite element methods. *Proceedings of the International Symposium on Symbolic and Algebraic Computation ISSAC*, pages 347–354, 2010.

[33] P. S. Wang. Finger - a symbolic system for automatic-generation of numerical programs in finite-element analysis. *J. Symb. Comput.*, 2(3):305–316, 1986.

[34] T. Zimmermann and D. Eyheramendy. Object-oriented finite elements I. principles of symbolic derivations and automatic programming. *Comput. Method. Appl. M.*, 132(3-4):259–276, 1996.