ESFLIB: A library to compute the element free Galerkin shape functions

Petr Krysl

Research Assistant Professor, Civil Engineering, Northwestern University, Evanston, IL, USA.

Ted Belytschko

Walter P. Murphy Professor of Civil and Mechanical Engineering, Northwestern University, Evanston, IL, USA.

We present a library for the computation of the shape functions for the element free Galerkin (EFG) method. While the EFG method is in many respects strikingly similar to the finite element method, the construction of the shape functions for the EFG method is much more complicated than in the FEM. Thus, the question arises whether it is possible to encapsulate the complexity of the shape function construction process, while providing a simple interface to the shape function data.
We present a shape function library design which proved unrestrictive (it can be used with almost any programming language without any special requirements on application program data structures), highly configurable, and robust.

Introduction

The element free Galerkin method (EFG) is a relatively recent approach. It was proposed by Belytschko *et al.* [1] as a consistent and improved version of the diffuse element method as introduced by Nayroles *et al.* [2]. It has been since applied to problems in static and dynamic fracture mechanics [3–5], modeling of material interfaces [6], large-strain inelastic deformations [7], thin plates and shells [8,9], and transient coupled problems [10]. The connections of the EFG method to the reproducing kernel particle methods [11], to *hp*-clouds [12] and to the most general method, partition of unity [13,14], have been described; see the review by Belytschko *et al.* [15].

The EFG method has been recently explored from the point of view of essential boundary conditions enforcement [16–18], techniques for speeding up the shape function evaluation [19], approximations in non-convex domains [20,21], and convergence rates [22,21].

The EFG is classified as a meshless method. The reason for this rests on its flexibility in the shape function construction. Let us compare the differences between the finite element method (FEM), and the EFG method (EFGM) in rather general terms.

In the FEM, one constructs a subdivision of the domain into non-overlapping finite elements, and then defines the support of a shape function associated with a node as a union of those finite elements which reference the node. Thus, the supports of the FE shape functions overlap exactly in a single finite element. To guarantee the required smoothness of the shape functions across inter-element boundaries, special consideration must be given to the form of the shape functions and to the topology of the mesh.

To summarize, in order to evaluate the FE shape functions one needs to

(i) construct the mesh topology (difficult and expensive, in general, since there are many constraints on the mesh topology and shape; also, this step is usually done off-line, by separate programs, mesh generators), and
(ii) compute the element-based expressions (easy and inexpensive, because the restrictions of the shape functions to an element are of simple, usually polynomial, form).

The EFG method is very different in this respect. To construct the shape functions, one uses the moving least squares technique [23], in which the influence of a node emanates from a weight function defined on an arbitrary compact subset of the domain. Since the shape functions can be constructed with arbitrary continuity, the boundaries of the node supports do not affect deleteriously the smoothness of the shape function; the smoothness of the shape function is given by the smoothness of the weight function and of the so-called basis functions, and both can easily be constructed in C^∞.

The arbitrary overlaps of the nodal supports lead to a variable connectivity: the number of nodes affecting the approximation varies from point to point arbitrarily, and is usually higher than for the FEM. Furthermore, the shape functions are determined from a least-squares problem, and they are non-polynomial. Therefore, we see that

(i) the topology is easily computed (although it needs to be done at run-time), and
(ii) the shape function expressions are complicated and expensive to evaluate.

The two methods can be thus seen to have very different implementation requirements. On the other hand, when one looks at the formulation of the problem being solved, e.g, data fitting, boundary value problem etc, the formulas are equivalent. Thus, the question arises: Is it possible to provide an equally simple shape function interface in EFG programs as in FE programs? When comparing the above diametrically opposed requirements, the answer would seem to be no, or at least 'not easily'. However, as we intend to show, there is a straighforward way to program the EFG shape function construction in a way which leads to both a simple interface to the application code, and to the implementation of the EFG shape function itself. This goal is achieved by providing an abstract form of the EFG shape function construction algorithm. This form specifies neither how certain operations are to be carried out, nor the data structures to be used. Instead, it leaves these details to be unspecified in the form of a protocol. The abstract algorithm is made concrete and executable by the application programmer, who provides code complying with the protocol.

The outline of the paper is as follows: In Section 1, we briefly review the equations of the EFG shape function construction, to make the paper self-contained. Section 2 discusses the motivation for the design drawing arguments from the similarity of the FEM and the EFGM, and concludes with an exposition of the design goals. The next two sections, Section 3 and 4, discuss two important concepts, renaming and callbacks. The next section, Section 5, introduces the concept of a callback, and describes both the shape function interface, and the programming of the callbacks, on a simple example. Section 6 refines the running example by introducing the blending (coupling) technique to modify the EFG shape functions to satisfy certain constraints, such as interpolation at some points. Section 7 discusses the construction of shape functions for non-convex domains, such as cracked bodies. Performance and how to improve it are discussed in Section 8.

The issues of error handling are crucial to the robustness of any software. They are discussed briefly in Section 9. To make the software useful to a wide audience, it needs to be well documented. While this is widely recognized, programmers rarely find time or the will to provide an adequate documentation. We describe a way to make this task more or less automatic in Section 10.

1 Construction of the EFG shape functions

To construct the EFG shape functions we can proceed by invoking the basic idea of the moving least squares technique, or by modifying the weight functions to satisfy consistency conditions. The latter approach seems to be more instructive, so we adopt it here.

We wish to construct the approximation to a function $u(\boldsymbol{x})$ in Ω such that it can be written as a linear combination of the shape functions ϕ_I, i.e.,

$$u(\boldsymbol{x}) \approx u_h(\boldsymbol{x}) = \sum_I \phi_I(\boldsymbol{x}) u_I \ , \tag{1}$$

where u_I are some coefficients (nodal unknowns) to be determined from the criteria defining the quality of the approximation. Note in particular that the nodal coefficients are not necessarily the values of the function u at the nodes, $u(\boldsymbol{x}_I) \neq u_I$.

We assume that the approximation is to be constructed in a domain Ω embedded in R^n, where $n = 1, 2, 3$. Each node I can affect the approximation within its domain of influence, B_I, which can be conveniently defined as such a subset of R^n where a weight function $w_I(\boldsymbol{x})$ is non-zero (in fact, positive), i.e.,

$$B_I = \{\boldsymbol{x} \in R^n : w_I(\boldsymbol{x}) = w(\boldsymbol{x}, \boldsymbol{x}_I) > 0\} \ . \tag{2}$$

Weight function. The weight function has a compact support, which ensures sparsity of the global matrices. The weight functions that are commonly used are listed below. They differ in both the shape of the domain of influence (parallelepiped centered at the node for tensor-product weights, or sphere), and in functional form (polynomials of varying degrees, or non-polynomial such as the truncated Gaussian weight).

– *quadratic tensor-product weight function*

$$w(\boldsymbol{x}_I, \boldsymbol{x}) = W\left(\frac{x - x_I}{d_{mxI}}\right) W\left(\frac{y - y_I}{d_{myI}}\right) W\left(\frac{z - z_I}{d_{mzI}}\right) \tag{3}$$

where the component function W is defined as

$$W(s) = \begin{cases} (1 - s^2) & \text{for } 1 > s \geq 0, \\ 0 & \text{for } s \geq 1. \end{cases} \tag{4}$$

– *composite quadratic tensor-product weight function*

$$w(\boldsymbol{x}_I, \boldsymbol{x}) = W\left(\frac{x - x_I}{d_{mxI}}\right) W\left(\frac{y - y_I}{d_{myI}}\right) W\left(\frac{z - z_I}{d_{mzI}}\right) \tag{5}$$

where the component function W is defined as

$$W(s) = \begin{cases} (1 - 2s^2) & \text{for } 1/2 > s \geq 0, \\ 2(1 - s)^2 & \text{for } 1 > s \geq 1/2. \\ 0 & \text{for } s \geq 1. \end{cases} \tag{6}$$

– *quadratic spherical weight function*

$$w(\boldsymbol{x}_I, \boldsymbol{x}) = w(r) = \begin{cases} (1 - r^2) & \text{for } 1 > r \geq 0, \\ 0 & \text{for } r \geq 1. \end{cases} \tag{7}$$

– *composite quadratic spherical weight function*

$$w(\boldsymbol{x}_I, \boldsymbol{x}) = w(r) = \begin{cases} (1 - 2r^2) & \text{for } 1/2 > r \geq 0, \\ 2(1-r)^2 & \text{for } 1 > r \geq 1/2. \\ 0 & \text{for } r \geq 1. \end{cases} \tag{8}$$

– *quartic spherical weight function*

$$w(\boldsymbol{x}_I, \boldsymbol{x}) = w(r) = \begin{cases} (1 - 6r^2 + 8r^3 - 3r^4) & \text{for } 1 > r \geq 0, \\ 0 & \text{for } r \geq 1. \end{cases} \tag{9}$$

– *truncated Gaussian spherical weight function*

$$w(\boldsymbol{x}_I, \boldsymbol{x}) = w(r) = \frac{\exp(\beta^2 r^2) - \exp(\beta^2)}{1 - \exp(\beta^2)} \tag{10}$$

In the above expressions, $r = ||\boldsymbol{x}_I - \boldsymbol{x}||/d_{mI}$, with d_{mI} being the radius of a spherical support, and d_{mxI}, d_{myI}, and d_{mzI} are the half the length of the support parallelepiped sides for the tensor product weights. The parameter β affects the shape of the weight function bell, and is usually adopted as $\beta = 4$.

Construction from reproducibility conditions. The convergence of approximation techniques often rests on local approximation properties of polynomials. Therefore, it is often required that the shape function should be able to reproduce exactly polynomials of a certain order (so-called "consistency conditions"). For example, in applications to second order partial differential equations, the shape functions are required to be of linear precision, which means that they should be able to reproduce exactly a constant and a linear polynomial. This requirement is also linked to the classical and generalized patch test. See Strang and Fix [24] for a thorough discussion of the classical patch test; Stummel [25] introduced the generalized form of the patch test.

The linear consistency conditions can be written as

$$u_h(\boldsymbol{x}) = u(\boldsymbol{x}) = a + \boldsymbol{b}^T \boldsymbol{x} \ . \tag{11}$$

Collecting the terms corresponding to the different powers and simplifying, we arrive at the following form of the consistency conditions

$$\sum_I \phi_I(\boldsymbol{x}) \quad {}^0u_I = 1 \ , \quad \text{and} \quad \sum_I \phi_I(\boldsymbol{x}) \quad {}^1\boldsymbol{u}_I = \boldsymbol{x} \ , \tag{12}$$

where ${}^0u_I$ and ${}^1\boldsymbol{u}_I$ are some parameters which make it possible for (12) to hold. The conditions (12) can be made more useful by requiring that the approximation (1) should *interpolate* the function u when it is a linear polynomial or a constant. Thus, Equation (12) boils down to

$$\sum_I \phi_I(\boldsymbol{x}) = 1 \ , \quad \text{and} \quad \sum_I \phi_I(\boldsymbol{x})\, \boldsymbol{x}_I = \boldsymbol{x} \ . \tag{13}$$

In general, we should not pick an arbitrary weight function to act as a shape function; the consistency conditions (13) would not be satisfied.

Following [15,26] we will seek the shape function as a "corrected" weight function, i.e., we multiply the weight function by a correction function $C_I(\boldsymbol{x})$, which will be designed so that the resulting shape function satisfies (13)

$$\phi_I(\boldsymbol{x}) = C_I(\boldsymbol{x}) w_I(\boldsymbol{x}) \ , \tag{14}$$

The correction function C_I will be sought in the form

$$C_I(\boldsymbol{x}) = \boldsymbol{a}(\boldsymbol{x})^T \boldsymbol{g}(\boldsymbol{x}_I) \ , \tag{15}$$

where $\boldsymbol{g}(\boldsymbol{x})$ is a column matrix of m linearly independent functions ($m \geq 4$ for 3D domains), and $\boldsymbol{a}$ is a column matrix of m coefficients. The functions g_i should include the linear polynomial and a constant. In addition, they may include also some other (possibly non-polynomial) functions, such as near-field crack tip solutions[27]. Since we need to determine m coefficients a_i to compute the correction function, we complete the linear consistency conditions by a requirement that *all* functions g_i should be reproduced exactly by their EFG approximations

$$\sum_I \phi_I(\boldsymbol{x}) \boldsymbol{g}(\boldsymbol{x}_I) = \boldsymbol{g}(\boldsymbol{x}) \ . \tag{16}$$

Substituting (14) and (15) into (16) gives

$$\sum_I w_I(\boldsymbol{x}) \left[\boldsymbol{g}(\boldsymbol{x}_I) \boldsymbol{g}^T(\boldsymbol{x}_I) \right] \boldsymbol{a}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{x}) \ . \tag{17}$$

Equation (17) can be cast in a form identical with that arrived at by the moving least squares technique (compare, for example, with [15]), i.e.,

$$\sum_I \sum_j w_I(\boldsymbol{x}) g_i(\boldsymbol{x}_I) g_j(\boldsymbol{x}_I) a_j(\boldsymbol{x}) = g_i(\boldsymbol{x}) \ , \tag{18}$$

or

$$\boldsymbol{A}(\boldsymbol{x}) \boldsymbol{a}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{x}) \ . \tag{19}$$

where $\boldsymbol{A}$ is given by

$$A_{ij} = \sum_I w_I(\boldsymbol{x}) g_i(\boldsymbol{x}_I) g_j(\boldsymbol{x}_I) \ . \tag{20}$$

The matrix $\boldsymbol{A}$ is symmetric. Under certain conditions it is expected to be positive-definite (more on this below), and is often called the moment matrix. The matrix $\boldsymbol{A}$ is usually factorized by the pivoting LU, QR factorization or singular value decomposition (the latter two are indicated for ill-conditioned matrices). In the software described in this article, we use the solvers implemented in the linear algebra package Meschach by D. Stewart and Z. Leyk [28].

Shape function derivatives. The derivatives of the shape functions are calculated by an approach described in Reference [19] which speeds up the computations. The required equations are obtained by differentiating (14) and by noting that the differentiation of (19) yields

$$\boldsymbol{A}_{,i}\boldsymbol{a} + \boldsymbol{A}\boldsymbol{a}_{,i} = \boldsymbol{g}_{,i} \ , \tag{21}$$

where $g_{,i}$ denotes $\partial g / \partial x_i$. Thus we can obtain the derivatives of $\boldsymbol{a}_{,i}$ by solving

$$\boldsymbol{A}\boldsymbol{a}_{,i} = \boldsymbol{g}_{,i} - \boldsymbol{A}_{,i}\boldsymbol{a} \ . \tag{22}$$

For this purpose the factorization of $\boldsymbol{A}$ computed when solving (19) can be reused, so the computation of the derivatives involves little extra effort.

Accuracy checks. The moment matrix may be ill-conditioned when (i) the basis functions g_j are (almost) linearly dependent, or (ii) there are not enough nodal supports overlapping at the given point, or (iii) the nodes whose supports overlap at the point are arranged in a special pattern, such as a conic section for a complete quadratic polynomial basis g_j. Provided enough supports overlap at a point and the basis functions are reasonably independent, the matrix $\boldsymbol{A}$ can be factorized with sufficient accuracy. However, a robust algorithm should always check the precision of the solutions of $\boldsymbol{A}$, and the final precision of the shape function values. There are two checks possible. The first one is indirect: one can estimate the condition number of the matrix $\boldsymbol{A}$, using one of the many techniques for doing this, which vary both in reliability and in cost. The second one is direct, and checks the final accuracy of the shape functions by invoking the first of the consistency conditions (13),

$$\sum_I \phi_I(\boldsymbol{x}) = 1 \ . \tag{23}$$

Differentiating this equation with respect to the independent variable, we can check also the accuracy of the computed shape function derivatives by checking that

$$\sum_I \phi_{I,i}(\boldsymbol{x}) = 0 \ , \ \text{for} \ \ i = x, y, z \ . \tag{24}$$

Time dependency of the shape functions. Let us consider the following problem of large deformation of solids. The deformation is described by the mapping $\boldsymbol{x} = \boldsymbol{\chi}(\boldsymbol{X}, t)$, where $\boldsymbol{X}$ is the material point (particle marker), x is the position material point $\boldsymbol{X}$ in the deformed configuration, t is the time, and $\boldsymbol{\chi}$ is the deformation. If the deformation is described in the total Lagrangian setting, i.e. if the boundaries of the node supports are curves deforming with the material, the shape functions are not dependent on time. Therefore, for example the approximations of the displacement and of the velocity are linked through

$$\begin{aligned} \boldsymbol{u}(\boldsymbol{X}, t) &= \sum_I \phi_I(\boldsymbol{X}) \boldsymbol{u}_I(t) \ , \quad \text{and} \\ \boldsymbol{v}(\boldsymbol{X}, t) &= \sum_I \phi_I(\boldsymbol{X}) \boldsymbol{v}_I(t) \\ &= \sum_I \phi_I(\boldsymbol{X}) \dot{\boldsymbol{u}}_I(t) \ . \end{aligned} \tag{25}$$

Let us note in this context that the FE shape functions are always expressed as being inscribed in the material (they are defined on the elements, which deform with the material), and are therefore time-independent.

Let us now consider what happens when the EFG node supports are inscribed in their given shape in the deformed state of the material (e.g, spheres on the deformed configuration). The material time derivatives then should take into account also the time-dependence of the shape functions. The shape functions may not depend on time directly, but only through the time-varying connectivity, i.e., through the positions of the nodes $\boldsymbol{X}_M$ in the deformed configuration. For example, let us consider the case where the shape functions are expressed in terms of the positions of the nodes in the deformed configuration, $\phi_I(\boldsymbol{\chi}(\boldsymbol{X}), \boldsymbol{\chi}(\boldsymbol{X}_M)$. The velocity would then be linked to the displacement through a much more complicated expression

$$\begin{aligned} \boldsymbol{v}(\boldsymbol{X}, t) &= \frac{\partial}{\partial t} \sum_I \phi_I(\boldsymbol{\chi}(\boldsymbol{X}), \boldsymbol{\chi}(\boldsymbol{X}_M)) \boldsymbol{u}_I(t) \\ &= \sum_I \left[\frac{\partial\, \phi_I(\boldsymbol{\chi}(\boldsymbol{X}), \boldsymbol{\chi}(\boldsymbol{X}_M))}{\partial\, \boldsymbol{\chi}(\boldsymbol{X})} \frac{\partial\, \boldsymbol{\chi}(\boldsymbol{X})}{\partial\, t} \right. \\ &\qquad \left. + \sum_M \frac{\partial\, \phi_I(\boldsymbol{\chi}(\boldsymbol{X}), \boldsymbol{\chi}(\boldsymbol{X}_M))}{\partial\, \boldsymbol{\chi}(\boldsymbol{X}_M)} \frac{\partial\, \boldsymbol{\chi}(\boldsymbol{X}_M)}{\partial\, t} \right] \boldsymbol{u}_I(t) \\ &\quad + \sum_I \phi_I(\boldsymbol{\chi}(\boldsymbol{X}), \boldsymbol{\chi}(\boldsymbol{X}_M))\, \dot{\boldsymbol{u}}_I(t) \end{aligned} \tag{26}$$

These expressions are truly a nightmare to program, and their implementation may be excessively expensive. Neglecting the term in the square brackets in (26) might help, but its effects are not yet fully explored.

The time-independence of the shape functions in the total Lagrangian formulation seems to be a good reason for using the EFG with this setting. Therefore, the dependence of the shape functions on time is not handled by the software presented here.

2 Design considerations

Comparison of the EFG and FE methods. As shown by Babuška and Melenk [14], both the finite element method and the element free Galerkin method can be viewed as specializations of the partition of unity method. Once an algorithm is described in a discrete form, the difference between using the EFG method or the FE method in the formulas shows only in the properties of the shape functions. Compare, for instance, the way a function may be approximated by an interpolation. In the FE method, we write

$$u_h(\boldsymbol{x}) = \sum_I N_I(\boldsymbol{x}) u_I \ , \tag{27}$$

where the nodal parameters u_I are simply the values of the approximated function at the nodes, $u_I = u(\boldsymbol{x}_I)$ due to the interpolating properties of the FEM, $N_I(\boldsymbol{x}_M) = \delta_{IM}$.

In complete analogy, the approximate function is written using an EFG approximation scheme as

$$u_h(\boldsymbol{x}) = \sum_I \phi_I(\boldsymbol{x}) u_I \ , \tag{28}$$

where the only difference with respect to (27) is the replacement of the finite element shape function N_I by the EFG shape function ϕ_I. Since here we do not have the Kronecker delta property, $\phi_I(\boldsymbol{x}_M) \neq \delta_{IM}$, we seek the nodal parameters u_I by setting up the interpolation conditions at the nodes

$$u_h(\boldsymbol{x}_M) = u(\boldsymbol{x}_M) = \sum_I \phi_I(\boldsymbol{x}_M) u_I \ , \tag{29}$$

and the (sparse) system of linear equations (29) needs to be solved for the unknown u_I's. Plugging the obtained coefficients u_I into (28) we can evaluate the approximation at any point.

Comparing (27) and (28), one can see a complete formal analogy. The difference lies in the evaluation of the values of the shape functions at the given point. Let us look at how it is done for the FEM. For the sake of concreteness, consider an 8-noded hexahedral finite element. The routine to evaluate the shape functions, together with their derivatives at a given point $\widehat{\boldsymbol{x}}$ (corresponding to the parametric coordinates ξ, η, θ) can be called like this

```
hex_shape_f (n, n_x, n_y, n_z,
                xi, eta, theta, x1, y1, z1, x2, ..., z8);
```

The output parameter `n` (`n_x`, `n_y`, `n_z`) is the array of eight shape function values (values of derivatives with respect to the spatial coordinates) evaluated at ξ, η, θ. The input parameters `xi`, `eta`, `theta` are the parametric coordinates of the point $\widehat{\boldsymbol{x}}$, and `x1`, . . . , `z8` are the spatial coordinates of the nodes.

One can see that the interface between the shape function routine `hex_shape_f ()` and the application program is very simple. It requires a small number of input parameters and no tuning of its operations.

On the other hand, the EFG shape function routine can be expected to require not only the similar arguments as the finite element routine, but also a large number of other parameters: shapes and sizes of domains of influence of the nodes, types of the weight functions for each node, data describing the basis functions g_i, etc. Also, the EFG shape function routine can fail, in contrast to its finite element counterpart; consider just the case of a (near) singular moment matrix, when there is no solution of (19). Therefore, the EFG shape function routine and its interface with the application program may turn out to be much more complex than its FE counterpart, when designed in a traditional, structured programming manner.

Design considerations. In the following, the acronym ESFLIB (**E**lement free Galerkin **S**hape **F**unction **LIB**rary) denotes both the concept of an abstract algorithm being encapsulated in a library (in the sense of a repository of software), and the actual code in the form of a library archive.

ESFLIB was designed with several objectives in view:

Flexibility: In order to satisfy varying demands posed by different application programs, the ESFLIB must be very flexible in terms of configurability (different weight functions, basis functions, etc.). We also wish to allow the application software to design and implement its data structures independently of the ESFLIB. This is especially important with respect to the use of the ESFLIB in mixed language environments using a mix of procedural (C, Fortran 77 and Fortran 90), and object-oriented languages.

Robustness: The software should be robust both in terms of implementation (robust linear algebra routines, sufficient number of checks), and in terms of resilience to misuse by the application programmer.

Maintainability: The software should be easily maintanable, especially with respect to its use in different space dimensions (1, 2, and 3), and in many different configurations with different implementation requirements.

Performance: The ESFLIB should provide acceptable performance in the worst case, and it should allow for transparent optimized implementations for some special needs. As an example of this, we can mention parallel evaluation of the shape functions values at several points on multi-threaded machines.

Readability: We wish to enhance the readability of the application programs, and to allow for better structuring. We also strive to make the ESFLIB self documented, with appropriate manuals generated from the source without human intervention.

It is our belief that the technology of a *library module* caters well to all of the above requirements. The software-engineering issues related to the subject of libraries are discussed to considerable depth for example by Plauger [29].

Terminology. In what follows, we use some abbreviations and acronyms to avoid repeating lengthy names.

ESFLIB EFG shape function library. Here we use the term "library" in its generic meaning denoting simply a repository of software.

SFLT Shape function library *template*. The template is a prescription for the creation of a type in the sense that template is an incomplete type, with some specifications intentionally left blank. The unresolved specifications can be viewed as variables of the type, and be substituted for at an appropriate time to create the fully specified types. In our case, the unresolved specifications are related to the number of space dimensions, to the calling conventions specific to the caller's language environment, and to some other issues of tuning, checking etc.

SFLA Shape function library *archive*. This is the full type specification, which is obtained from the template, SFLT, by preprocessing and compilation.

SFLI The concrete instance of the type as specified by SFLA. It is a run-time entity, created and initialized by calling an initialization routine from the library.

TSFV Table of values of the shape functions and of its derivatives. This is a useful abstraction; the term does not necessarily reflect the actual storage of data. The TSFV is valid at a particular point $\widehat{x}$ at which it was evaluated. There is a row in the table for each node whose support overlaps $\widehat{x}$. A j^{th} row of the table stores:

(i) The identifier of the node, I_j; this can be a pointer, or an integer index, or any other suitable key.

(ii) The value of the shape function for the node I_j, $\phi_{I_j}(\widehat{x})$.

(iii) The value of the first derivatives of the shape function for each spatial dimension, $\phi_{I_j,x}(\widehat{x})$ for 1D, $\phi_{I_j,x}(\widehat{x})$, $\phi_{I_j,y}(\widehat{x})$ for 2D, and $\phi_{I_j,x}(\widehat{x})$, $\phi_{I_j,y}(\widehat{x})$, $\phi_{I_j,z}(\widehat{x})$ for 3D.

(iv) In some cases not only the first derivatives are needed, but also the second derivatives of the shape functions. Then the SFLA's are generated with provisions for the second derivatives, which are stored for each combination of spatial dimensions (taking into account the symmetry of differentiation) $\phi_{I_j,xx}(\widehat{x})$ for 1D, ...

SFLA generation. The ESFLIB source is written and maintained in the form corresponding to an SFLT, i.e., it describes the algorithm in a general, parameterized form, which leaves certain decisions, such as the number of space dimensions, inclusion of certain features, open. The rewriting capability of the C-language preprocessor is applied to the SFLT source to remove and adjust all parametric features during the first phase of the compilation, when the SFLT is converted to the SFLA source. The SFLA source is then compiled and collected into a library archive.

An application program may link in a single SFLA for each space dimension. However, the source of the SFLT names all variables and functions in a generic way. Some of the variables are private to the compiled modules; those do not need to be renamed, since the linker is able to distinguish between private and public symbols. On the other hand, the names SFLT source uses for public functions need to be changed (mangled) so that different SFLA's use different names. Otherwise the linker would run into a name clash (two or more public symbols with the same name), and the linking phase would fail.

Finally, during the processing of the SFLA source, the documentation is automatically generated, so that it reflects the appearance of the code the application programmer needs to use.

Instantiation of a SFLI. While the SFLA as a type is a persistent entity, the SFLI as the instantiation of the SFLA is a run-time object. The technique of instantiation of a model (type) is well known from the object-oriented (OO) literature. The instances maintain different state data, are configured to work slightly differently, but are still subject to the same protocol. The observance of the protocol guarantees that the software does what it is supposed to.

The application program can create just a single SFLI from a given SFLA, or it can create more. Consider, for example, an implementation of the Petrov-Galerkin method, where there is a need to work with two different sets of shape functions in the test and trial spaces. Thus, a Petrov-Galerkin program with 1D, 2D and 3D capabilities would need three SFLA's (for each dimension), and six SFLI's (two for each SFLA; one for trial and one for test functions). Alternatively, one could use only three SFLI's, and reconfigure them when the context is switched between test and trial space shape functions. The former solution is, of course, preferable from the maintainability and readability viewpoint.

State data. The notion of an SFLI rests on the notion of an object (in the sense commonly used in the OO literature). The object consists of some state data, it complies with some interface specifications, and responds to requests sent to it. How these requests are satisfied is immaterial as long as the results are consistent with the specification of the interfaces. The evaluation of the TSFV requires a considerable amount of state data – linear algebra data structures, configuration parameters, etc. The state data should be private to each SFLI, otherwise one could compromise parallelization, as discussed below.

Implementation language. We could have used an object-oriented language, such as C++. However, designing the ESFLIB around some classes, which the application would then be required to use in order to be able to work with the ESFLIB, is overly restrictive. It would force the programmer to program in an object-oriented language, and to structure the application program to suit the ESFLIB. What we wanted was the exact opposite.

Writing the ESFLIB in Fortran 90 was also out of the question, since the instantiation in the form envisioned above is awkward in this language, if not impossible.

If we do not consider some obscure and rarely used languages, the choice is restricted essentially to Ada, Eiffel, and the C language. Practical considerations (availability, inter-language mixing) and convenience led us to adopt C as our implementation language.

3 Renaming scheme and space dimensions

As indicated previously, the source of the ESFLIB corresponds to the template form, the SFLT. Here we discuss the scheme which we use to cope with the conversion of the SFLT into separate SFLA's.

To rename some symbols (types of variables, function prototypes) we use the C-preprocessor capability to expand code. Thus we define

```
#define GLUE(A, B)   A##B
#define GL(A, B)     GLUE(A, B)
#define ESFLIBID(ID) GL(GL(GL(ESFL,ESFLIB_SPACE_DIM),_),ID)
```

The above macros embelish the identifier `ID` by prefixing it with the string `ESFL`n`_`, where n is the space dimension (1, 2, or 3). Thus, the given the following macro definition,

```
#define ESFLIB_weight_func ESFLIBID(weight_func)
```

all occurrences of the string `ESFLIB_weight_func` are replaced in the source files during the compilation of, for example, the 3D library, by `ESFL3_weight_func`. The compiler thus "sees" the mangled symbol name, not the generic one, and the finished library archive defines the symbol `ESFL3_weight_func` as it should.

The ESFLIB needs to describe the algorithm of computing the shape function values by invoking different functions, whose signature, of course, is different for the one-, two-, or three-dimensional SFLI's. Therefore, we use to following technique to allow for automatic adjustement of the calling sequences during code generation. We define a series of macros for each spatial dimension

```
#if ESFLIB_SPACE_DIM == 2
#   define REF_LIST(X, Y, Z) X, Y
...
#endif
```

Then, during the code generation (assuming an SFLA for two space dimensions is generated), the function name `ESFLIB_shape_func_at` is transformed by renaming into `ESFL2_shape_func_at`, and the code fragment

```
double REF_LIST(x, y, z);
REF_LIST (x = rcx - atx, y = rcy - aty, z = rcz - atz);
curr_lib->efg_node_coords (efgp, REF_LIST (&x, &y, &z));
```

is transformed by adjustment to the number of space dimensions into

```
double x, y;
x = rcx - atx, y = rcy - aty;
curr_lib->efg_node_coords (efgp, &x, &y);
```

Therefore, the compiler can perform the full range of compile-time type checking. Since the code generation produces also header files for the different space dimensions, the compiler can also check the application code for type consistency.

4 Callbacks

The SFLI can obtain the input data needed to compute the shape function table at a given point in several (at least two) ways.

– The first approach is to assemble input data outside the library and pass the information as argument lists at each invocation of a library routine. Thus, one would for example search for all nodes whose supports overlap the given point, store them in a list, compute for each of them the values of the weight function and its derivatives, and pass them to the library routines. However, this would be not only unwieldy and difficult to read, it would also be very error prone and a nighmare to maintain.
– Another way, and we believe it is the better solution, is to let the library call the application code with requests for data. This technique is well established in the form of callbacks in one of the best known graphical user interfaces, the X Window System, and its conceptual form is also well established in the object-oriented programming paradigm.

Let us explain the concept of a callback on an analogy with phone numbers. Consider the following algorithm: An agent is asked to perform certain actions in certain circumstances, for example in case of an emergency, or if one wants to find out about the up-to-date weather forecast. The actions consist in both cases in making a phone call to obtain/tranmit the needed information.

```
emergency_phone_# = $\boxed{\qquad\qquad}$
weather_info_phone_# = $\boxed{\qquad\qquad}$
...
if      (emergency)
    dial emergency_phone_#
else if (need to know the weather forecast)
    dial weather_info_phone_#
```

As can be seen, the boxes which should hold the numbers to call are empty. That means that the requested actions cannot be performed. Now, imagine we tell the agent which number to use to obtain help in an emergency. The code it would execute could look like this

```
emergency_phone_# = $\boxed{911}$
weather_info_phone_# = $\boxed{\phantom{911}}$
...
if      (emergency)
   dial emergency_phone_#
else if (need to know the weather forecast)
   dial weather_info_phone_#
...
```

One could augment the above further to take an alternative action when the phone number is not available, and to require that the algorithm cannot start without having some particular phone number. For instance like this

```
emergency_phone_# = $\boxed{911}$
weather_info_phone_# = $\boxed{\phantom{911}}$
...
require emergency_phone_#
if      (emergency)
   dial emergency_phone_#
else if (need to know the weather forecast)
   if (weather_info_phone_# available)
      dial weather_info_phone_#
   else
      try to find it in newspapers
...
```

Note that if there follows a conversation after the dialing, it usually involves some information exchange, in a single direction or bidirectionally.

The callbacks used by the library are direct equivalents of the phone numbers; invocation of the callback is the equivalent of dialing, and so forth. Some of the callbacks are optional (the algorithm can perform some alternative computation, or simply do nothing), others are required; all of them are passed some context data. There may be also some data computed by the callback and passed back to the library.

One note regarding the C-language implementation. The ESFLIB callbacks are technically handled as pointers to functions, which can be for the present purposes thought of as "function names". Passing, storing, or invoking a function by pointer can be done by simply referring to its name.

5 Example of ESFLIB use

Understandably, we cannot reproduce a user's manual and describe every function provided by the library. However, we can illustrate our design by introducing the most important concepts in action. Therefore, we describe a typical use of the ESFLIB in a simple program for approximation of functions of a single variable by interpolation. Later on, we also briefly discuss applications in two or three dimensions, where an additional issue of non-convex boundaries comes into play.

The first thing an application code should do to be able to use the ESFLIB, is to create a SFLI. The parameters passed to the initialization function are

(i) the error handler function name, `err_handler`,
(ii) the data needed by the search callbacks, `lobs1d`, (in this example, we are using a bounding box search library, which is not part of ESFLIB),
(iii) the callbacks for the search of nodes whose support overlaps a given point (`find_overlaps`), and for iteration through the nodes found (`first` and `next`),
(iv) the callback to obtain the node location, `node_loc`,
(v) then we pass two `NULL`'s to indicate that we want a polynomial basis, and that an inlined, optimized code, should be used,
(vi) the next parameter (`2`) is the number of the basis terms (complete linear polynomial, 1 and x; thus two terms),
(vii) the names of the functions to compute the value and the derivative of the weight, `ESFL1_gh_w` and `ESFL1_gh_D1` (in this case, the functions are bundled with the SFLI, but it is of course possible to use one's own functions),
(viii) the callback to obtain the support size of a node, `supp_size`,
(ix) and finally, we pass some `NULL`'s to indicate that we do not wish to configure these callbacks.

```
esflib1 = ESFL1_init (
          err_handler, /* error handler */
          lobs1d,      /* data for the search routine */
          find_overlaps, first, next, /* search callbacks */
          node_loc,    /* callback to get node location */
          NULL, NULL,  /* use in-lined code for basis g_j */
          2,           /* # of linear basis terms, 1 and x */
          ESFL1_gh_w, ESFL1_gh_D1,
          supp_size,   /* callback to get support size */
          NULL, NULL, NULL, 0, NULL /* do not configure */
          );
```

The function to search for the nodes whose supports overlap a given point `x` simply calls a function from a bounding box library, and returns a boolean outcome (successful search or a failure). The nodes found are collected in a buffer by the library, and can be iterated over when needed as shown in what follows.

```
static int
find_overlaps (ESFLIB_search_struct_t ss, double x)
{
  lobs_1d_front_end_t *lobs = (lobs_1d_front_end_t *)ss;
  return lobs_1d_search_for_boxes_overlapping_pt (lobs, x);
}
```

The two functions used to iterate the nodes found by `find_overlaps()` use the search library functions for access to the found objects. The functions are rather simple and do not need further elaboration, with the exception of the value returned. It is either the node found, or `NULL`, when there is nothing to return, i.e., when there was no node found (`first`), or the iteration reached the end (`next`).

```
static ESFLIB_efg_node_id_t
first (ESFLIB_search_struct_t ss)
{
  lobs_1d_front_end_t *lobs = (lobs_1d_front_end_t *)ss;
  return lobs_1d_first_overlapping_object (lobs);
}

static ESFLIB_efg_node_id_t
next (ESFLIB_search_struct_t ss)
{
  lobs_1d_front_end_t *lobs = (lobs_1d_front_end_t *)ss;
  return lobs_1d_next_overlapping_object (lobs);
}
```

The next two functions provide a means for the SFLI to obtain information about the nodes. The functions access fields in the data structure of a node. Note that they rely on the input parameter, `efgp`, to be a pointer to a node. This is quite safe, however, since the SFLI calls these callbacks passing to them whatever `first` and `next` returned, unless they returned `NULL`, in which case the functions do not get called at all.

```
static void
node_loc(ESFLIB_efg_node_id_t efgp, double *x)
{
  *x = ((node_t *)efgp)->x;
}

static void
supp_size (ESFLIB_efg_node_id_t efgp, double *ssize)
{
  *ssize = ((node_t *)efgp)->d_m;
}
```

Once the SFLI has been created, one can use it to create a TSFV, `sft`. This table is initially empty (no rows).

```
sft = ESFL1_new_shape_func (esflib1);
```

To evaluate the TSFV at a given point, `x`, we call the function `ESFL1_shape_func_at()`. It is passed the TSFV, `sft`, and the location `x`. The remaining two parameters are on this occasion unused. One could be used when constructing the TSFV near a reentrant corner of the domain Ω (which does not make sense in the current, one-dimensional, setting), and the other would indicate data used when blending the EFG shape functions with another basis.

```
ESFL1_shape_func_at (sft, x, NULL, NULL);
```

Once the TSFV has been constructed, one can access any information in it, and use it in the application program. One comment is in order here: both the library type and the TSFV type are opaque to the application writer, i.e., the application cannot access any information from the private data of these types directly; the access functions provided by ESFLIB are the only way. This measure promotes maintainability, enhances robustness, and its run-time cost is rarely significant.

To compute the approximation at the point $\widehat{x}$, $u_h(\widehat{x})$, we need to evaluate the sum $\sum_I \phi_I(\widehat{x}) u_I$. One way to do this is by looping over the number of rows in the TSFV `sft`, retrieving from them the identifier of the node I (a pointer, in this case), and the value of the shape function ϕ_I.

```
u_h = 0;
nn = ESFL1_num_of_entries (sft);
for (I = 0; I < nn; I++) {
 an = ESFL1_efg_node_at (sft, I);
 ESFL1_inspect_at (sft, ESFLIB_ORDER_0, ESFLIB_X, I, &phi_I);
 u_h = u_h + phi_I * an->u_I;
}
```

While this approach is quite straighforward, there is a much more elegant and readable way. One can use the library function `ESFL1_f_eval()`, which is passed the TSFV, `sft`, the name of the callback to access the nodal value associated with a particular node from the TSFV, the function order (`ESFLIB_ORDER_0` means function value itself, `ESFLIB_ORDER_1` would mean the first derivative). The result of the sum is returned in the variable `u_h`.

```
ESFL1_f_eval (sft, get_u_I, ESFLIB_ORDER_0, ESFLIB_X, &u_h);
```

The callback `get_u_I ()` is very simple. It just returns the value of the nodal parameter stored in the node data structure.

```
static double
get_u_I (ESFLIB_efg_node_id_t efgp)
{
  return ((node_t *)efgp)->u_I;
}
```

This way is clearly preferable to the explicit looping in the application program. While we might not save on the total of lines coded, we make the flow of the algorithm more explicit, and consequently the program becomes more readable.

6 Blending

The EFG shape functions do not produce interpolations, but rather approximations. This property complicates the imposition of the essential boundary conditions. Therefore, a blending technique was proposed by Belytschko *et al.* [16] and by Krongauz and Belytschko[17] to couple finite element and EFG trial spaces. It was also applied by Fleming *et al.* [27] to blend together asymptotic crack tip fields with a regular EFG approximation. It is a very general technique, which allows essentially any set of basis functions to be blended with another set over arbitrarily shaped transition regions. The blending function is also sometimes called a *ramp*, because it is 0 on one side of the transition region, and 1 on the opposite, varying monotonously in between.

ESFLIB allows for construction of EFG shape functions in the transition regions. Since the most useful applications of blending so far have been in coupling finite element type functions with EFG approximations, we describe here how to implement the callbacks which allow us to make the EFG approximation computed by the sample program to interpolate at the end nodes.

The transition region (cell) is defined as the span between the end-point nodes (`e`), and their interior neighbors (`i`). The finite element produces interpolations, and it is therefore natural to establish a linear FE basis in the transition cell using the two nodes, `e` and `i`, and enforce the interpolation conditions at the end nodes by the blending. Note that in order to blend two sets of shape functions and still preserve their precision both sets need to be complete over the transition region. That is why the nodes in the transition cell are associated both with the EFG approximation and with the linear basis.

First we define the transition region (cell) data package type. This package is used to transfer data to the callbacks described next, and it is passed to the library when evaluating a shape function.

```
typedef struct blend_cell_data_t {
   node_t *end;
   node_t *interior;
} blend_cell_data_t;
```

The transition cell package stores the pointers to the two nodes of the cell.

We assign the two transition cell nodes the numbers 0 (the end node) and 1 (the interior node). The `blend_cell_node_num` callback returns a local number of an EFG node, i.e. 0 for the end node, and 1 for the interior node, and it returns -1 to indicate that the node on input is not one of the cell nodes. This numbering is used by the ESFLIB routines to keep track of which node is associated with a transition region node shape function of the linear type.

```
static int
blend_cell_node_num (ESFLIB_generic_ptr_t if_data,
                          ESFLIB_efg_node_id_t efgn)
{
  blend_cell_data_t *
              tcell = (blend_cell_data_t *) if_data;
  node_t *n = (node_t *)efgn;
  if      (n == tcell->end)      return  0; /* end node */
  else if (n == tcell->interior) return  1; /* interior node */
  else                           return -1; /* it is neither */
}
```

The procedure `linear_shape_f` computes the shape function and its derivative for a transition cell node.

```
static void
linear_shape_f (ESFLIB_generic_ptr_t if_data, int node_num,
                double *n, double *nx, double x)
{
  blend_cell_data_t *
             tcell = (blend_cell_data_t *) if_data;
  node_t *i = tcell->interior, *e = tcell->end;
  double L  = (i->x - e->x);
  if (node_num == 0) {
     *n  = (i->x - x) / L;
     *nx = -1 / L;
  } else {
     *n  = (x - e->x) / L;
     *nx = 1 / L;
  }
}
```

The `construct_ramp` callback computes the value of the ramping function at the current point, `x`. The ramp function should be equal to one at the interior node (i.e., at the EFG boundary of the transition cell), and zero at the end nodes. Therefore, the ramp is in fact identical with that portion of the linear function associated with the interior node which is inside the transition region (compare with the "else" part of `linear_shape_f()`).

```
static void
construct_ramp (ESFLIB_generic_ptr_t if_data,
                double *R, double D_R[1], double x)
{
  blend_cell_data_t *
             tcell = (blend_cell_data_t *) if_data;
  node_t *i = tcell->interior,
         *e = tcell->end;
  double L  = (i->x - e->x);
  *R        = (x - e->x) / L;
  D_R[0]    = 1 / L;
}
```

To be able to use the blending technique, we need to configure the SFLI so that it can call our callbacks `linear_shape_f`, `construct_ramp`, and `blend_cell_node_num`.

```
ESFL1_set_construct_ramp_func (esflib1, construct_ramp);
ESFL1_set_fe_shape_func (esflib1, linear_shape_f);
ESFL1_set_elem_node_num_func (esflib1, blend_cell_node_num);
ESFL1_set_base_elem_num (esflib1, 0);/* number from zero */
```

In order to compute the TSFV at a point outside the transition regions we can stil use the same code fragment as before (passing `NULL` as the transition package data signals no blending is required). To evaluate the TSFV *inside* the blending cell, we need to pack a transition cell package to be used by the callbacks, and pass it as an additional argument to `ESFL1_shape_func_at()`:

```
static blend_cell_data_t cd;
cd.end      = end;
cd.interior = interior;
ESFL1_shape_func_at (sft, x, NULL, &cd);
```

Note that when one constructs the TSFV inside a transition cell only the invocation of the `ESFL1_shape_func_at()` function changes; the TSFV computed there is indistinguishable from a TSFV computed anywhere else.

7 Handling of reentrant corners in domains

The moving least squares technique produces arbitrarily smooth shape functions. However, this is sometimes undesirable. One of these cases occurs near discontinuity surfaces in domains, such as cracks, or near non-convex boundaries, such as reentrant corners. As a consequence, the supports of nodes affected by these discontinuities need to be modified to incorporate the proper behaviour of the shape functions or of its derivatives.

Algorithms for dealing with non-convex boundaries have been discussed by Krysl and Belytschko [21]. For the reader's convenience, we summarize the findings here.

Let us denote the list of nodes which can affect the approximation at $\widehat{\boldsymbol{x}}$ by $\mathcal{A}(\widehat{\boldsymbol{x}})$. We shall denote this set also as the *active set*. Further, let us denote by Ω the spatial domain of interest, by Γ the surface of discontinuity, and by B_I the domain of influence (support set) of the node I.

Node inclusion criterion. The first step in the treatment of discontinuities is to decide whether a node should be included in the list $\mathcal{A}(\widehat{\boldsymbol{x}})$ of a given point $\widehat{\boldsymbol{x}}$. Reference [21] lists two criteria: Provided the original (unmodified) domain of influence of the node I covers the point $\widehat{\boldsymbol{x}}$, the node is included in the set $\mathcal{A}(\widehat{\boldsymbol{x}})$:

Visibility criterion: if node I is visible from the point $\widehat{\boldsymbol{x}}$ (assuming the discontinuity surface is "opaque").

Contained-path criterion: if node I can be reached from $\widehat{\boldsymbol{x}}$ by a path C without leaving the intersection of Ω and B_I, and without crossing the discontinuity Γ.

Modification of the domain of influence. If a node I qualifies for inclusion in $\mathcal{A}(\widehat{\boldsymbol{x}})$, the next action depends on the criterion used.

Visibility criterion The most important implication of this criterion is that the domains of influence are truncated by the exclusion of the "shadows" of boundaries. The resulting shape functions are then not even in C^0 in the domain. However, as shown in Reference [21], the theory of non-conforming approximation then, under certain conditions, implies that the EFG approximation is still convergent. In fact, for some classes of problems, the non-conforming EFG method gives very good results, even better than a conforming one. While the reason is not yet fully theoretically understood, one can expect some error cancelling to be at play.

Contained-path criterion There are two possible courses of action. In the first, the domain of influence of the node is left unchanged. The shape functions are then as smooth in the vicinity of the discontinuity surface as anywhere else. The second approach modifies the domain of influence of nodes, for which the supports overlap the edges of the discontinuity surface. The reason is that for some applications, the approximation is required to display certain singularity at the edge of the discontinuity (e.g, crack front). However, the original supports may constrain the approximation to be excessively smooth near the edge. Thus, a possible cure of this phenomenon is to make the supports reach only just round the discontinuity, but still preserve their smoothness to some degree (usually to produce C^0 shape functions).

Implementation of the node inclusion criteria. The node inclusion/exclusion is decided by ESFLIB using the following algorithm:

```
if (    node_qualifies_by_weight
     OR  node_is_at_interface_cell_vertex)
   if (node_is_at_interface_cell_vertex)
      node is included in the active set
   else
      if (have_callback_to_check_for_exclusion_by_boundary)
         if (NOT  excluded_by_boundary)
             node is included in the active set
      else
         node is included in the active set
```

The following comments apply:

– The `node_qualifies_by_weight` predicate is expressed simply as $w_I(\boldsymbol{x}) > 0$, where $w_I(\boldsymbol{x})$ is the value of the weight function associated with node I at point $\boldsymbol{x}$.
– Provided the TSFV is being evaluated inside a blending region, the `node_is_at_interface_cell_vertex` predicate is TRUE if the node I is located at one of the vertices of the blending region; otherwise it is set to `FALSE`.
– The predicate `excluded_by_boundary` is the value returned by the callback supplied by the application program. Therefore, it is up to the application writer to decide which criterion is used to check for inclusion/exclusion of nodes.

The second step, domain of influence modification, is again implemented by callbacks. We shall not delve into the subject deeper due to space constraints; the reader can look up additional information in References [30,20].

8 Performance issues

Connectivity computation. As already noted, the connectivity in the EFG method is determined at run-time, for each point at which the shape functions need to be evaluated. The connectivity at a given point $\widehat{\boldsymbol{x}}$ is established by searching for all nodes whose supports overlap (cover) $\widehat{\boldsymbol{x}}$. (The connectivity computation is actually more involved for non-convex boundaries.) Let us denote the active set of nodes which can affect the approximation at $\widehat{\boldsymbol{x}}$ by $\mathcal{A}(\widehat{\boldsymbol{x}})$. The speed with which the set $\mathcal{A}(\widehat{\boldsymbol{x}})$ is computed depends crucially on the search strategy adopted. Sequential searches are of $O(N)$ complexity for each evaluation point (N is the total number of EFG nodes). Therefore, one should always try to use a search technique which employs localization, since such techniques can perform the search at a given point in an optimal time $O(\log N)$; see, for example, References [31,32].

Optimization by not computing. One of the ways of making the EFG shape functions less costly is in fact the simplest: do not compute things you do not need. For example, for certain domains one knows that they are either convex or insignificantly non-convex. In that case not checking for boundary/support interference may save considerable amount of time. Another way to save is not to compute derivatives when they are not needed.

Checks vs. optimal performance. ESFLIB offers a series of verifications and checks to provide robust and fool-proof computation. However, at times it might be preferable to limit the number of checks performed, so that a higher performance could be achieved. This is possible by a run-time configuration of a SFLI.

Access to shape function data. Some applications, such as explicit dynamics programs, avoid the cost of shape function recomputation by computing once and storing the TSFV's. The access to shape function data may appear to be unnecessarily expensive when effected through function invocations as it is demonstrated in the sample code fragments above. The solution may be to copy the TSFV's to local storage.

Local storage may also be a solution to the following problem: ESFLIB stores all shape function data as double precision values. The memory to store hundreds of thousands of TSFV's may be prohibitive, though, especially for large active sets. To cut the storage cost in half with respect to TSFV's is to store them in local variables as floats (`real*4`) instead of as doubles (`real*8`) (this may be expected to work on only those machines which store fewer bits for floats than for doubles).

Parallelization. The evaluation of the EFG shape functions is inherently more costly than it is the case with the FEM. (Or rather, while the cost is in the EFG all concentrated to run-time, the FEM delegates part of it to separate computations, which does not necessarily mean lower overall computation cost, but definitely leads to smaller run-time cost of the solution run.) Therefore, it is worthwhile to consider parallel evaluation of the shape functions for the EFG programs. Especially with respect to the increasing availability of symmetric multi-processing machines, since essentially no synchronization is necessary.

While the computation of the shape functions is theoretically in the class of "embarrassingly parallel" algorithms, care must be taken to allow for parallel invocation of the ESFLIB. ESFLIB does not use any static data, not even read-only ones, which could be a source of synchronization bottlenecks. However, since each SFLI uses some private data structures, a single SFLI cannot be used by two concurrent threads of computation. The solution is to create as many SFLI's as there are threads (processors).

Another possible contention area is, for instance, the search for the active set $\mathcal{A}(\widehat{\boldsymbol{x}})$. The search can be performed by storing each node in a search structure (tree, cell array, etc), and by performing a search customized to the search structure used. However, the search algorithm may store the information about the set $\mathcal{A}(\widehat{\boldsymbol{x}})$ as it is collected in a storage area private to the search structure, which would, of course, preclude its concurrent use by two or more threads of execution. The solution might be to use different storage for each search thread. Similarly, when constructing the active set $\mathcal{A}(\widehat{\boldsymbol{x}})$ for non-convex boundaries, a search needs to be performed to check for possible interference of boundaries with the supports. Also these searches might use private data, which would need to be re-assigned to parallel threads of computation.

Some performance measurements. We provide an illustrative selection of performance measurements. All of them were carried out on an HP/9000 715/100XC workstation. The reported times are CPU time consumed by the process.

The measurements were done for a uniform, rectangular, three-dimensional grid of N nodes in a unit cube. The spacing of the nodes was a, and was the same in all directions. We set the size of domains of influence to a multiple of $d_m = \kappa\sqrt{3}a$. We denote by $c_{\mathcal{A}}$ the number of nodes in the set $\mathcal{A}(\widehat{\boldsymbol{x}})$, i.e., $c_{\mathcal{A}} = \operatorname{card}\{\mathcal{A}(\widehat{\boldsymbol{x}})\}$. The shape functions are evaluated E times at randomly generated points inside the interval $\{0.1; 0.9\} \times \{0.1; 0.9\} \times \{0.1; 0.9\}$ to avoid irregular connectivities for the regions close to the boundary. The search for connectivities was conducted by a bounding box library, which stores the nodes as boxes, and the search formulation is "find all boxes overlapping a given point". A single search can be done in $O(\log N)$ time.

For all measurements given, we list the *total* time t needed to evaluate the shape functions at the given number of points.

(i) In this experiment, we determine how the time to evaluate the shape functions changes with $c_{\mathcal{A}}$. We use the following settings: tensor-product quadratic weight, $N = 10000$, $E = 10000$. The results are given in Table 1. The dependency can be observed to be approximately linear.

(ii) In the next measurement, we explore how costly the various weight functions are. We use the quadratic, composite quadratic and quartic spherical, and quadratic and composite quadratic tensor-product weights. The multiplier $\kappa = 1.05$ for the spherical weights, and $\kappa = 0.85$ for the tensor-product weights, to achieve approximately the same $c_{\mathcal{A}} \approx 25$ for all support shapes. The results are given in Table 2.

(iii) In this measurement, we test the relative cost of linear, quadratic, and cubic polynomial basis. We use $N = 10000$, $E = 1000$, tensor-product quadratic weight, and $\kappa = 1.5$ which corresponds to $c_{\mathcal{A}} = 139$. The results are given in Table 3.

(iv) In this measurement, we test the relative cost of the different solvers which are used to factorize the moment equation (19). We use $N = 10000$, $E = 100000$, tensor-product quadratic weight, and $\kappa = 0.85$ which corresponds to $c_{\mathcal{A}} = 25$. The results are given in Table 4.

(v) It might be of interest to know the relative expense connected to each separate part of the shape function evaluation in ESFLIB. For the above grid with $N = 10000$, $\kappa = 0.85$, quadratic tensor-product weight, and

$c_{\mathcal{A}}$	κ	t [s]
25	0.85	7.58
34	0.94	9.32
45	1.03	11.19
60	1.13	14.02
80	1.24	17.58
107	1.37	23.60
142	1.50	28.97
189	1.65	35.86
249	1.82	46.33
326	2.00	61.28
428	2.20	79.58
556	2.42	100.36
719	2.66	132.93
928	2.93	168.38

Table 1
Cost of different support sizes

linear polynomial basis, with a bounding box search library, the topology-related computations account for 33% of the total time, evaluation of the weight function used up approx. 6%, and 56% of the total time is actually spent in the linear algebra related to the equations of Section 1. The rest is spent in various bookkeeping routines and other minor consumers of time.

(vi) How expensive are the checks that the consistency conditions are reasonably well satisfied? It depends on a number of factors, but in the worst case less than 5% of the total time is spent in checking that equations (23), and (24) hold with sufficient accuracy. Therefore, it does not seem wise to turn off these checks, especially because they can be invaluable when using graded meshes with variable support sizes.

Performance comparison of FEM and EFGM. We can distinguish two cases of shape function construction for the FEM. The first is used in Galerkin procedures, when the shape functions are evaluated at *integration points*. In that case, the connectivity in the FEM is known beforehand, since one knows in which element, and at which parametric coordinates in that element the shape functions are being evaluated. In the second case, the shape functions

weight	shape	t [s]
quadratic	spherical	9.2
composite quadratic	spherical	10.0
quartic	spherical	10.0
truncated Gaussian	spherical	10.2
quadratic	tensor-product	8.2
composite quadratic	tensor-product	8.6

Table 2
Cost of different weight functions

basis g_j	terms	t [s]
linear	4	3.3
quadratic	10	7.5
cubic	20	20.7

Table 3
Cost of different number of terms in a polynomial basis

Solver characteristics	t [s]
LU with partial pivoting	95.1
QR	105.2
Choleski	97.1

Table 4
Cost for different solvers

are evaluated at an *arbitrary point* inside the domain. This case corresponds to re-interpolation from mesh to mesh, post-processing etc. It is then necessary to find the element enclosing the given point (and probably while testing, also evaluate the parametric coordinates of the point in the element).

To put things into perspective, let us quote here one measurement of time needed to construct the isoparametric shape functions, both values and derivatives, for a hexahedral, eight-noded finite element: calling the shape function routines 10000-times costs approximately 0.15 second. Comparing with Table 1, we can see that the EFG shape functions are 50-times more expensive to compute. However, let us examine these numbers in the light of the above classification.

While in the first case, the FEM relegates part of the effort to separate mesh generators, so the run-time of the analysis program is not burdened by the need to determine the topology, in the second case, the cost is all carried by a single program. In the EFGM, these two cases are essentially identical as far as evaluation of the shape functions goes; the topology is computed at run-time. (However, note that it is also possible to precompute the topology for the EFGM for a Galerkin procedure. In that case, a cell structure is used to evaluate the weak-form integrals, so it is possible to precompute connectivities for each integration point, or for each integration cell, by a separate, preprocessing step.)

Now we can discuss the implications of the above observations. In the first case, the FEM is clearly much faster than EFGM when evaluating the shape functions, by an order of magnitude, roughly speaking. However, should we include also the cost of mesh generation, with triangulation speed of arbitrary meshes on the same HP workstation on the order of 2000 triangles (1500 tetrahedra) per second, the difference could be substantially reduced. Take, for example, a 3D unstructured FE mesh with 10000 nodes, 50000 tetrahedra; the mesh generation time could be on the order of 50 seconds.

In the second case, namely for the random-point evaluation, the FEM can be still expected to be faster due to the simple character of per element expressions, but not by orders of magnitude.

9 Error handling

The computation of the EFG shape functions can fail when the moment matrix is (nearly) singular. Also, it is possible to configure the SFLI erroneously, or to access non-existent data in a shape function table, etc. In these cases, errors need to be handled in a consistent and reliable manner. We chose to provide both boolean return values to signal success or failure, and an error handler to be invoked by the library in case an error was detected. It is the combination of these two means which proves most satisfactory.

The error handler can also be invoked from the user callbacks so that a uniform way of error handling can be provided for all functions associated with the EFG shape function evaluation.

The error handler we provide for our sample program presented here recognizes two kinds of error conditions. One is the precision loss found during the check of the EFG shape functions (one should get $\sum_I \phi_I(x) = 1$, and $\sum_I \phi_{I,x}(x) = 0$ for any x): if the mismatch is small (errors $< 10^{-6}$, in this case), only a warning is reported; otherwise an error exit is made. The other group of errors includes any other circumstance, and an error exit from the program follows.

```
static void
err_handler (ESFLIB_lib_t lib)
{
  if (ESFL1_status (lib) == ESFLIB_BAD_PREC) {
    if (ESFL1_prec_loss (lib) < 1.e-6)
      fprintf (stderr, "%s\n", ESFL1_explain_error (lib));
    else
      errexit (ESFL1_explain_error (lib));
  } else {
    errexit (ESFL1_explain_error (lib));
  }
}
```

10 Documentation

We document the ESFLIB routines in the source code (purpose, arguments and return values), and we use the automatic documentation extraction program, `c2man` by G. Stoney [33], to produce both HTML browsable documents, and a LaTeX documentation. This approach has the obvious advantage that the documentation can be kept up-to-date with the source. We show an example of a manual page for the function `ESFL2_shape_func_at()`. First we show the source code fragment from the SFLT. This is the only place where the ESFLIB maintains documentation.

```
/* Evaluate shape function table. */
ESFLIB_bool_t /* returns TRUE if all went OK;
            otherwise FALSE indicates an error */
ESFLIB_shape_func_at (
    ESFLIB_sf_table_t sft, /* shape function table */
    double atx /* x-coordinate of the evaluation point */,
#if ESFLIB_SPACE_DIM >= 2
    double aty /* y-coordinate of the evaluation point */,
#endif
#if ESFLIB_SPACE_DIM == 3
    double atz /* z-coordinate of the evaluation point */,
#endif
    ESFLIB_generic_ptr_t eval_for,/* data which is passed
       to the callback node_hidden_by_bdry(); as it is not interpreted
       by the library in any way, it may be passed in as NULL */
    ESFLIB_generic_ptr_t if_data /* data which is passed
       to the callbacks implementing the blending (ramping) for
       interface regions; as it is not interpreted by the library in any
       way, it may be passed in as NULL */);
```

And this is the manual page, generated completely without human intervention (for this journal, however, manually edited to save space).

NAME

ESFL2_shape_func_at — Evaluate shape function table.

SYNOPSIS

```
ESFLIB_bool_t ESFL2_shape_func_at
(
    ESFLIB_sf_table_t sft,
    double atx,
    double aty,
    ESFLIB_generic_ptr_t eval_for,
    ESFLIB_generic_ptr_t if_data
);
```

PARAMETERS

ESFLIB_sf_table_t sft
Shape function table.

double atx
X-coordinate of the evaluation point.

double aty
Y-coordinate of the evaluation point.

ESFLIB_generic_ptr_t eval_for
Data which is passed to the callback node_hidden_by_bdry(); as it is not interpreted by the library in any way, it may be passed in as NULL.

ESFLIB_generic_ptr_t if_data
Data which is passed to the callbacks implementing the blending (ramping) for interface regions; as it is not interpreted by the library in any way, it may be passed in as NULL.

DESCRIPTION

Evaluate shape function table.

RETURNS

Returns TRUE if all went OK; otherwise FALSE indicates an error.

SEE ALSO

ESFL2_init(), ESFL2_new_shape_func(), . . .

Conclusions

The finite element method (FEM) and the element free Galerkin method (EFGM) share a commonality as partitions of unity. This relationship shows in many ways, especially when dealing with abstract formulas. However, the paradigm the EFGM uses to construct the shape functions is radically different from the traditional FEM approach. This equips the EFGM with flexibility and the ability to control the absolute error, while at the same time making the construction of the shape functions appreciably more expensive. One of the reasons is the need to compute the topology on the fly, the other is the non-polynomial, implicit definition of the shape functions.

In constrast to the FEM, the algorithm of EFG shape function construction is rather complicated, and requires considerable number of input parameters. Therefore, it makes good sense to use a software component, which would make it as easy to design applications with the EFGM as easily as with the FEM. The above observations led us to hide the complexity of the EFG shape function construction, and to provide an interface to the shape function construction which matches the FEM in simplicity.

In this paper, we have described the EFG shape function library, ESFLIB. We have discussed the design goals (flexibility, robustness, maintainability, solid performance, ease of use), and the concepts we have applied. The library interface has been documented on a sample program.

With the ESFLIB, the application programmer is supplied a flexible tool, allowing for experimentation and customization. The implementation is also robust and thoroughly debugged, providing a consistent exception and error handling component. The library implements state-of-the-art algorithms, and its performance has been tuned. The library source is heavily parameterized, so that library instances, customized and specialized to required space dimensions, can be generated from a single source. This contributes to maintainability of the source. The interface of the library used by the application programmer hides the implementation, is very simple, and allows for its use in mixed language environments.

The library was implemented in the ANSI C language for portability. It consists of +5000 lines of C-language header and source files. The ESFLIB software has been used successfully on a number of architectures (SGI Indy, Indigo, Power Challenge, IBM RS/6000, HP/9000 7xx, CRAY C-90), and both in the C-language environment, and in conjuction with a purely Fortran 77 analysis code.

Given the size and complexity of the ESFLIB, the benefits of its constant reuse with application programs become obvious. In many cases the amount of programming and debugging needed by an analysis program is much lower than that invested in the shape function library alone.

Acknowledgments

The support of Office of Naval Research and U.S. Army Research Office is gratefully acknowledged. J. Dolbow and R. Chudoba are thanked for valuable comments.

References

[1] T. Belytschko, Y. Y. Lu, and L. Gu. Element-free Galerkin methods. *International Journal of Numerical Methods in Engineering*, 37:229–256, 1994.

[2] B. Nayroles, G. Touzot, and P. Villon. Generalizing the finite element method: diffuse approximation and diffuse elements. *Computational Mechanics*, 10:307–318, 1992.

[3] T. Belytschko, L. Gu, and Y. Y. Lu. Fracture and crack growth by element-free Galerkin methods. *Modelling Simul. Mater. Sci. Eng.*, 2:519–534, 1994.

[4] T. Belytschko, Y. Y. Lu, L. Gu, and M. Tabbara. Element-free Galerkin methods for static and dynamic fracture. *International Journal of Solids and Structures*, 17/18:2547–2570, 1995.

[5] Y. Y. Lu, T. Belytschko, and M. Tabbara. Element-free Galerkin methods for wave propagation and dynamic fracture. *Computer Methods in Applied Mechanics and Engineering*, 126:131–153, 1995.

[6] L. W. Cordes and B. Moran. Treatment of material discontinuity in the element-free Galerkin method. *Computer Methods in Applied Mechanics and Engineering*, submitted.

[7] T. Belytschko, P. Krysl, and Y. Krongauz. A three-dimensional explicit element-free Galerkin method. *International Journal for Numerical Methods in Fluids*, 24:1253–1270, 1997.

[8] P. Krysl and T. Belytschko. Analysis of thin plates by the Element-Free Galerkin method. *Computational Mechanics*, 17:26–35, 1996.

[9] P. Krysl and T. Belytschko. Analysis of thin shells by the Element-Free Galerkin method. *Int. J. Solids & Structures*, 33:3057–3080, 1996.

[10] H. Modaressi and P. Aubert. A diffuse element – finite element technique for transient coupled analysis. *International Journal of Numerical Methods in Engineering*, 39:3809–3838, 1996.

[11] W. K. Liu, Sukky Jun, S. Li, J. Adee, and T. Belytschko. Reproducing kernel particle methods for structural dynamics. *International Journal of Numerical Methods in Engineering*, 38:1655–1679, 1995.

[12] C. A. Duarte and J. T. Oden. *Hp* clouds – A meshless method to solve boundary-value problems. Technical Report 95-05, Texas Institute for Computational and Applied Mathematics, Austin, 1995.

[13] I. Babuška and J. M. Melenk. The partition of unity finite element method. Technical Report BN-1185, Inst. for Phys. Sc. and Tech., University of Maryland, Maryland, 1995.

[14] I. Babuška and J. M. Melenk. The partition of unity method. *International Journal of Numerical Methods in Engineering*, 40:727–758, 1997.

[15] T. Belytschko, Y. Krongauz, D. Organ, M. Fleming, and P. Krysl. Meshless methods: An overview and recent developments. *Computer Methods in Applied Mechanics and Engineering*, 139:3–47, 1996.

[16] T. Belytschko, D. Organ, and Y. Krongauz. A coupled finite element–element-free Galerkin method. *Computational Mechanics*, 17:186–195, 1995.

[17] Y. Krongauz and T. Belytschko. Enforcement of essential boundary conditions in meshless approximations using finite elements. *Computer Methods in Applied Mechanics and Engineering*, 131:133–145, 1996.

[18] T. Belytschko, D. Organ, and Y. Krongauz. A coupled finite element–element-free Galerkin method. *Computational Mechanics*, 17:186–195, 1995.

[19] T. Belytschko, Y. Krongauz, M. Fleming, D. Organ, and W. K. Liu. Smoothing and accelerated computations in the element-free Galerkin method. *Journal of Computational and Applied Mathematics*, 74:111–126, 1996.

[20] D. J. Organ, M. Fleming, and T. Belytschko. Continuous meshless approximations for nonconvex bodies by diffraction and transparency. *Computational Mechanics*, 18:225–235, 1996.

[21] P. Krysl and T. Belytschko. Element-free Galerkin method: Convergence of the continuous and discontinuous shape functions. *Computer Methods in Applied Mechanics and Engineering*, 148:257–277, 1997.

[22] W. K. Liu, S. Li, and T. Belytschko. Reproducing least square kernel Galerkin method. (i) Methodology and convergence. *Computer Methods in Applied Mechanics and Engineering*, 1996. in preparation.

[23] P. Lancaster and K. Salkauskas. *Curve and surface fitting: an introduction.* Academic Press, London, Orlando, 1986.

[24] W. G. Strang and G. J. Fix. *An analysis of the finite element method.* Prentice-Hall, Englewood Cliffs, N.J., 1973.

[25] F. Stummel. The generalized patch test. *SIAM J. Numer. Analysis*, 3:449–471, 1979.

[26] W. K. Liu, S. Jun, and Y. F. Zhang. Reproducing kernel particle methods. *International Journal for Numerical Methods in Engineering*, 20:1081–1106, 1995.

[27] M. Fleming, Y.A. Chu, B. Moran, and T. Belytschko. Enriched element-free Galerkin methods for crack tip fields. *International Journal for Numerical Methods in Engineering*, 40:1483–1504, 1997.

[28] D. Stewart and Z. Leyk. *Meschach Library, Version 1.2b.* Netlib Repository, http://www.netlib.org/c/index.html, 1994.

[29] P. J. Plauger. *The standard C library.* Prentice Hall, Englewood Cliffs, N.J., 1991.

[30] R. Franke and G. Nielson. Surface approximation with imposed conditions. In R. E. Barnhill and W. Boehm, editors, *Surfaces in computer aided geometric design*, pages 135–147, New York, 1983. North Holland.

[31] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, Mass., 1988.

[32] F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction.* Springer-Verlag, New York, 1990.

[33] G. Stoney. *c2man, Version 2.* Canon Information Systems Research Australia, ftp://dnpap.et.tudelft.nl/pub/Unix/Util, 1996.