Object-oriented numerical integration—a template scheme for FEM and BEM applications

Rogério J. Marczak

Mechanical Engineering Department, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS 90050-170, Brazil

Received 3 November 2004; revised 20 December 2004; accepted 20 May 2005

Available online 11 July 2005

Abstract

An object-oriented numerical integration template implementation is presented on the basis of the C++ programming language. Aiming its straightforward application in finite and boundary element methods, the design supports integrand objects of scalar, vector or matrix types, so that a single programming statement is able to integrate element matrices and vectors. The integrand can contain singularities like the ones typically found in boundary element methods, allowing the evaluation of both regular and singular integrals under the same programming structure. The use of the proposed design is illustrated through some elementary applications as well as finite element and boundary element code excerpts.

© 2005 Elsevier Ltd. All rights reserved.

Keywords: Numerical integration; Object-oriented programming; Templates; Singular integrals; Boundary element method; Finite element method

1. Introduction

Numerical integration is a very important step in engineering and scientific calculations. A considerable number of available methods can be used depending on the application and the type of integrand. Among the classic integration rules one can find schemes based on the simple trapezoidal rule, Newton-Cotes rules [8], Gaussian rules [25] and recursive monotone stable (RMS) rules [9,19]. Each one of these methods has its weak and strong points, making the choice of any of them highly application dependent.

The Gaussian quadrature rules seems to be the most used scheme for engineering applications, due its good accuracy and fairly fast convergence for most regular kernel functions. It does not enable recursive reuse of the integration points, but this is not an issue in many engineering applications, where the necessary number of integration points is known a priori. One of the most common examples is the integration of stiffness and mass matrices in the finite element method (FEM).

The implementation of numerical integration procedures can be accomplished by many ways. It is a usual programming practice to incorporate the quadrature loops directly into the subroutine that evaluates the resulting integral, regardless the dimension of the integration domain. This is perfectly acceptable when the required number of integration points (quadrature order) is low. When the quadrature order becomes high and variable, it is advisable to implement another subroutine, which returns the stations and weights to avoid lengthy, repetitive data definitions throughout the code. Simpler cases can be even analytically integrated by symbolic algebra software [31,32].

In many other situations, however, the integrand shows some degree of singularity or there are checks and other operations that must be included inside the quadrature loops. Yet another situation appears in applications with two or three-dimensional integrals, where it may be necessary the use of an integration scheme for one direction and another scheme for the others. In these cases, more elaborated approaches should be used, particularly if the integration routine is called a large number of times. The boundary element method (BEM) is an instance of such cases, as the method commonly deals with improper integrals (singular kernels). Depending on the position of the collocation point and the boundary element being integrated, the integrand may show regular, quasi-singular,
weakly singular or strongly singular behavior [4,5]. This usually drives the researcher towards a very different code structure than the ones employed in FEM codes, particularly for strongly singular integrals. A suitable integration routine able to handle all these cases would be desirable from the implementation point of view.

This work presents a template class designed to support most of the aforementioned situations. It is based on Gaussian quadrature rules, but other rules can be implemented. The approach enables the integration of functions defined with standalone subroutines, but the integration of general objects is possible by subclassing and overriding a single member function. The class also handles weakly and strongly singular integrals, aiming their applications in the BEM. Examples of usage are shown for a number of situations, including the integration of stiffness and mass matrices in the FEM as well as the collocation matrices in boundary element methods.

2. Typical FEM and BEM integrands

Before proceeding to the description of the proposed class, this section reviews shortly some typical applications of numerical integration with the finite and boundary element methods. For the sake of simplicity, only the two-dimensional linear case will be analyzed.

The evaluation of stiffness and mass matrices or load vectors is the most common example of numerical integration usage in finite element methods. In standard displacement based formulations, the corresponding stiffness and mass expressions for a given finite element \( e \) can be written, respectively [2]

\[
K^e = \int_{\Omega} B^T D B J \, d\Omega \tag{1a}
\]

\[
M^e = \int_{\Omega} \rho N^T N J \, d\Omega \tag{1b}
\]

where \( B \) is the strain-displacement matrix, \( D \) is the constitutive matrix and \( N \) is the shape functions matrix, \( \Omega \) stands for the normalized domain of integration (length, area or volume) and \( J \) is the corresponding Jacobian of the transformation. Similar expressions are used for the evaluation of the element load vectors:

\[
f^e = \int_{\Omega} N q \, d\Omega \tag{2}
\]

where \( q \) contains the nodal values of the loading. Eqs. (1a,b) and (2) have different forms in mixed or hybrid approaches.

In a BEM analysis, the starting point usually is the solution of the associate boundary value problem. The discretized boundary integral identity for the displacement field can be written as [5]

\[
C(p)u(p) + \sum_{e=1}^{\text{NE}} \left( \int_{\Gamma} T(p, q) N(q) J \, d\Gamma \right) u^e(q) = \sum_{e=1}^{\text{NE}} \left( \int_{\Gamma} U(p, q) N(q) J \, d\Gamma \right) t^e(q) + \sum_{d=1}^{\text{ND}} \left( \int_{\Omega} U(p, q) N(q) J \, d\Omega \right) q^d(q) \tag{3}
\]

In Eq. (3) \( p \) is the collocation (load) point, \( q \) is the field point and \( \text{NE} \) is the number of boundary elements. The shape functions matrix \( N \) are used to interpolate the boundary displacements \( (u^e) \) and tractions \( (t^e) \) over each boundary element \( e \). The domain loadings \( (q^d) \) are integrated on each of the ND domain cells. The displacement \( (U) \) and traction \( (T) \) fundamental solution tensors can be found elsewhere for various differential operators, as well as the geometric \( (C) \) factors [4,5]. The last integral is evaluated on a two-dimensional (area) domain partition while the other integrals are evaluated on a one-dimensional (length) partition. Eq. (3) is rewritten (for each collocation point)

\[
Hu = Gt + f \tag{4}
\]

where

\[
H = \int_{\Gamma} T(p, q) N(q) J \, d\Gamma + C(p) \tag{5a}
\]

\[
G = \int_{\Gamma} U(p, q) N(q) J \, d\Gamma \tag{5b}
\]

\[
f = \int_{\Omega} U(p, q) N(q) J \, d\Omega \tag{5c}
\]

The geometric factor \( C \) must be added to the right side of (5a) only when \( p \) belongs to the element being integrated [5]. Eq. (4) is evaluated for each boundary node and superposed in the global system of equations. After the imposition of the boundary conditions, the global system is solved for the unknowns on the boundary.

Eq. (5a,b,c) are valid for most direct boundary element formulations. The Galerkin (symmetric) version of the BEM leads to different forms for them, but the framework developed here remains valid. The main difficulty in the numerical implementation of (5a,b,c) is related to the eventual singularities contained in tensors \( U \) and \( T \). Fundamental solutions are usually written in terms of the distance \( r = |q-p| \), and singular kernels of the type \( \ln(r) \) and \( r^{-\alpha} \) are commonly found. Clearly, these integrands become unbounded if the load point is inside the element being integrated. In the former case (and in the latter one when \( \alpha \) is
smaller than the dimension of the integration domain) the integral is called weakly singular. The latter case is called strongly singular if the surface element being integrated. However, as the load point approaches the element being integrated, the singular behavior of the kernels becomes more conspicuous, and the number of required integration points grows for a given accuracy, in comparison to the fully regular situation. This case is known as quasi-singular. It thus becomes clear that the numerical integration procedures in the BEM are not of straightforward implementation when compared to the FEM.

Many approaches have been devised to handle BEM integrals, with varying degree of success and simplicity. A popular scheme used for the evaluation of the weakly and quasi-singular cases is a cubic transformation of the stations and weights of Gauss quadrature \cite{28,29}. In essence, Telles’ method is a transformation that concentrates the stations around the singular pole, where the Jacobian vanishes. The method is quite effective to integrate improper but convergent (weakly singular) kernels \cite{28}, and it is easily adaptable for quasi-singular cases \cite{29}. Because it is based on Gaussian rules, the implementation is straightforward in existing codes. For singular integrals, a large number of approaches have been proposed in the last decades, most of them based on finite part and Cauchy principal value (CPV) definitions. In spite of a few successful special quadrature rules \cite{14,15}, regularization techniques \cite{21,27} are more appealing for general implementations as they can be applied over the \([-1,+1]\) interval without further transformations, and employ Gaussian stations and weights \cite{10,11}.

Since the normalized domain \([-1,+1]\) (in each direction) is generally used for the integration of Eqs. (1a,b), (2) and (5a,b,c), they are recovered simply as

\[
K^\prime = \int_{-1}^{+1} \int_{-1}^{+1} B^\prime DB \, d\xi_1 d\xi_2 \quad (6a)
\]

\[
M^\prime = \int_{-1}^{+1} \int_{-1}^{+1} \rho N^T N \, d\xi_1 d\xi_2 \quad (6b)
\]

\[
f^\prime = \int_{-1}^{+1} N q \, d\xi \quad (6c)
\]

and

\[
H = \int_{-1}^{+1} T(p,q) N(q) J \, d\xi + C(p) \quad (7a)
\]

\[
G = \int_{-1}^{+1} U(p,q) N(q) J \, d\xi \quad (7b)
\]

\[
f = \int_{-1}^{+1} \int_{-1}^{+1} U(p,q) N(q) J \, d\xi_1 d\xi_2 \quad (7c)
\]

which are now in a suitable form for the application of Gaussian quadrature for a generic tensor-type kernel \(f\)

\[
I_{1D} = \sum_{i=1}^{K} f(\xi_i) w_i \quad (8a)
\]

\[
I_{2D} = \sum_{i=1}^{K} \sum_{j=1}^{K} f(\xi_i, \xi_j) w_i w_j \quad (8b)
\]

for one- and two-dimensional cases, respectively. The order of the quadrature \(K\) is usually determined experimentally, taking into account the regularity of the kernel in each direction. It is worth to stress that the kernel \(f\) may be a scalar, a vector or a matrix function. In a general implementation, it would be interesting to have a procedure able to carry out the calculation of (8a,b) regardless the algebraic type of the integrand, or its regularity.

Regarding the evaluation of Eq. (7a,b,c) in the singular case, the direct method \cite{10} is among a number of integration methods able to handle strongly singular integrals in the same programming framework of the regular ones, i.e. using Eq. (8a,b). Regular integrals can be conveniently handled by coordinate transformations applied to the Gauss stations and weights \cite{28}. Therefore, it is possible to design a unified environment to compute Eqs. (6a,b,c) and (7a,b,c), provided both can be evaluated using primarily Gaussian-type rules.

3. A simple template-based class proposal for numerical integration

The use of object-oriented programming (OOP) \cite{26} has become a common paradigm in many software development fields, including scientific and engineering software. The diffusion of the use of object-oriented (OO) languages in engineering is due to a number of factors, most of them deeply rooted in the demand for extensibility and reusability of the codes (or part of them) without demanding the costs associated to the development of new software or to unwanted changes in codes successfully tested and used \cite{22}.

The objective of this section is to illustrate a simple numerical integration class developed in C++ language which is particularly adaptable for use in FEM and BEM source codes. Besides being extensible and reusable, the need to handle different algebraic types of integrand with the same class must be taken into account, i.e. the class must be able to integrate scalar, vector or matrix functions with the same functionality. An object-oriented language is the obvious choice for such task. In C++ language, this level of generality is possible through templates, which enables the definition of argument types and return types during
runtime. All calculations are to be performed on the normalized space \([-1, +1]\) in each direction.

There is a large number of numerical integration procedures developed in Fortran language [17]. However, quite often they are very dependent of the type of integrand making its choice and implementation cumbersome. Even a recommended system was developed to help the selection of suitable algorithms [20]. Early attempts aimed elementary calculus tools [6], and improper integrals were dealt with using RMS methods, which is very time consuming in BEM applications. Very few approaches succeeded in developing an OO numerical integration class with sufficient generality degree to encompass FEM and BEM applications. Reference [18] presents an interesting approach to integrate typical BEM kernels, but it is directed to scalar functions only.

Thus, the main goal of the present class proposal is twofold: integrate any algebraic type of integrand and handle singular integrals. The former can be accomplished by using fully template-based definitions. This leads to the concept of numerical integration of objects [1], which would allow a member function of an arbitrary object to be integrated (provided it overloads the basic algebraic operations) as a Riemann sum

\[
\text{Result} = \int_{-1}^{+1} \text{Object}(x)\,dx = \sum_{i=1}^{K} \text{Object}(\xi_i)w_i
\]

where Result is of the same type of Object.

A number of methods can be applied to achieve the second goal. The aforementioned Telles’ transformation [28] was chosen here as a primary method to integrate weakly singular integrals, while the direct method [10] was used for the strongly singular ones. In this way, only Gaussian stations and weights need to be used. Special

---

**Fig. 1.** Basic UML diagram of mcQuadrature class.

```cpp
1: + static mcDictionary mcQuadratureTypes
2: + enum __mcQuadType__
3: + static const int nGaussLeg
4: + static const int nGaussCub
5: # mcArray order,type
6: # mcCoord singpole
7: # void setStackVectors()
8: # Vector getValues(int dir, char c);
9: + mcQuadrature(int dir, int num, int typ)
10: + virtual ~mcQuadrature()
11: + void setOrder(int n1, int n2, int n3)
12: + int getOrder(int dir)
13: + void setType(int t1, int t2, int t3)
14: + int getType(int dir)
15: + Vector getStations(int dir)
16: + Vector getWeights(int dir)
17: + void setIntegrand(mcIntegrand1D userFunc)
18: + void setIntegrand(mcIntegrand2D userFunc)
19: + void setSingPole(mcCoord& sing)
20: + mcCoord& getSingPole()
21: + T NIntegrate(double acc)
22: + T NIntegrate(mcIntegrand1D userFunc, double acc)
23: + T NIntegrate(mcIntegrand2D userFunc, double acc)
24: + T BEMIntegrate(mcBElement& be, mcBEInt1 userFunc, char* id, double acc)
25: + T BEMIntegrate(mcDCell& dc, mcDCInt2 userFunc, char* id, double acc)
26: + T FEMIntegrate(mcFElement& fe, mcFEInt2 userFunc, double acc)
```
quadratures also can be included, like Gaussian quadrature with logarithmic weights [25] and finite part rules [15]. However, they are generally defined for integration domains other than the $[-1, +1]$ one, and scaling of the integrand may be needed.

The basic UML diagram [3] of the proposed class is shown in Fig. 1, and some of its methods are briefly described below:

- **void setOrder (int n1, int n2, int n3);**

  Used to change the order of the quadrature in each integration direction.

- **void setType(int t1, int t2, int t3);**

  Specifies the type of quadrature to be used in each direction. Table shows the quadratures implemented so far. Note that Kutt’s rules [13] and Gaussian quadrature with logarithmic weights are included to exemplify the use of non-Gaussian rules. Analytical integration of specific integrands can be dealt with through a special option (typ = 0).

- **Vector getStations(int dir);**
- **Vector getWeights(int dir);**

  These members return vectors containing the coordinates of the stations and the corresponding weights for each direction, respectively. These methods are included to enable the user to implement customized subroutines for special applications.

- **void setIntegrand(mcIntegrand& obj, double acc);**

  Changes the current integrand.

- **void setSingPole(Vector sing);**
- **Vector& setSingPole();**

  These members informs (or retrieve) the position of the singularity in the integration domain, which is necessary for the integration of singular kernels (this is obviously ignored for regular kernels).

1: mcQuadrature<double> Func(1,3,1);
2: mcQuadrature<Vector> Vect(2,2,1);
3: mcQuadrature<Matrix> Mat(2,4,1);

Dimension of the integration domain
Number of integration points
Type of quadrature

4: Ke = Mat.NIntegrate( my_stiffness );
5: Fe = Vect.NIntegrate( my_load );
6: f = Func.NIntegrate( my_func );

Names of the subroutines which evaluate the integrands

Fig. 2. Usage of mcQuadrature objects.

- **T NIntegrate(mcIntegrand& obj, double acc);**

  Performs the numerical integration and returns a result of type T. Variations derived from this member can be implemented for specific purposes.

Being the class fully template based, the user do not need to take care of the algebraic type of the integrand. In addition, the members invoked to perform the numerical integration remain the same regardless the regularity of the kernel. The constructor (Fig. 2) sets the quadrature parameters. The user specifies the dimension of the integration domain (dim), the order of the quadrature (nip) and the rule to be used (according to Table 1). The order and the type of quadrature are initially assumed the same in all directions by the constructor, but this can be changed later for each direction.

The type of singularity in the integrand defines the rule to be used. FEM integrands are always regular, but BEM kernels may be not. However, this information is generally known from the fundamental solutions. In a more elaborated OO approach, one could store the order and type of singularity in a fundamental solution class, which is used to set up the appropriate mcQuadrature rule. This method was successfully used in a general BEM code [16], where the singular character of each fundamental solution entry is identified by an integer

<table>
<thead>
<tr>
<th>Code (TYP)</th>
<th>Quadrature type</th>
<th>Available orders (nip)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Analytic integration</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>Gauss–Legendre</td>
<td>1(1)10, 12(2)20, 24, 32(8)48, 64</td>
</tr>
<tr>
<td>2</td>
<td>Gauss–Legendre with cubic transf.</td>
<td>Same as above</td>
</tr>
<tr>
<td>3</td>
<td>Gauss–Legendre with log weights</td>
<td>2(1)8</td>
</tr>
<tr>
<td>4</td>
<td>Kutt’s quadrature</td>
<td>2(1)14, 17, 18</td>
</tr>
</tbody>
</table>

Table 1: Quadrature rules implemented
The current integrand is defined by pointers to functions, enabling the integration of any kernel provided it is defined in a standalone subroutine or in an object member function. Since the mcQuadrature object is informed of the subroutine by pointers, there is no need to recompile the code when the integrand is changed. In addition, a single object can be reused as many times as necessary. Fig. 2 shows an elementary FEM application excerpt. In lines 1–3, the quadrature objects Func, Vect and Mat are created to integrate scalar, vector and matrix functions, respectively. In line 4, the statement returns a stiffness matrix \( K_e \) integrated by four points in each direction. In line 5, the vector \( F_e \) is returned after being integrated using two points in each direction, while line 6 exemplifies the integration of a real function with three integration points. It is worth to note that there is no need to loop over the entries of the matrices/vectors, since this is implicitly done (vector and matrix classes are abundant in the literature [7,24], and will not be reviewed here). In this example, the word passed as arguments of NIntegrate are the actual names of the subroutines that evaluate each integrand, and then the objects Func, Vect and Mat can be reused for other integrands simply by invoking the setIntegrand member.

The case shown in Fig. 2 is quite simple for integrand functions of global scope. However, object member functions have limited visibility and proper interfaces of IntegrandnD have to be provided when the integrand is an object member function.

### 4. Examples

In order to illustrate the usage of the proposed class, this section presents some elementary source codes written in C++ language. Many variations of the basic codes shown here can be implemented. The main goal is to show how the use of this type of class can simplify common programming tasks related to numerical integration.

The code in Fig. 3 illustrates a driver program used to integrate two scalar functions, namely \( f_1(x) = x^{10} \) and \( f_2(x) = -\ln|x| \) over the \( \xi = [-1,1] \) domain. A single mcQuadrature object (funcInt) is created in line 5 as a one-dimensional, single variable integrator. In line 6 a vector of integers is defined containing four orders of

```
Driver program:
1. #include "mc_quadrature.hpp"
2. double sfunc1(double);
3. double sfunc2(double);
4. int main() {
5.   mcQuadrature<double> funcInt(1,2,1);
6.   mcArray ord(4); ord(1) = 2; ord(2) = 4; ord(3) = 8; ord(4) = 16;
7.   cout << endl << "K      Result f1(x)      Result f2(x) " << endl;
8.   for (int i=1; i<=4; i++) {
9.     funcInt.setOrder(ord(i)); cout << ord(i);
10.    funcInt.setIntegrand(sfunc1);
11.    funcInt.setType(1);
12.    cout << "       " << funcInt.NIntegrate() << endl;
13.    funcInt.setIntegrand(sfunc2);
14.    funcInt.setType(2);
15.    cout << "       " << funcInt.NIntegrate() << endl;
16. }
17. return 0;
18. }

19. double sfunc1(double x) { return pow(x,10); }
20. double sfunc2(double x) { return -log(fabs(x)); }

Output:

<table>
<thead>
<tr>
<th>K</th>
<th>Result f1(x)</th>
<th>Result f2(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8.23045e-03</td>
<td>3.29584e+00</td>
</tr>
<tr>
<td>4</td>
<td>1.56035e-01</td>
<td>2.15800e+00</td>
</tr>
<tr>
<td>8</td>
<td>1.81818e-01</td>
<td>2.02144e+00</td>
</tr>
<tr>
<td>16</td>
<td>1.81818e-01</td>
<td>2.00286e+00</td>
</tr>
</tbody>
</table>
```

Fig. 3. Code example used to integrate \( x^{10} \) and \(-\ln|x|\) functions.
quadrature: 2, 4, 8 and 16 integration points. The program loops over these quadrature orders (lines 8–16) printing the corresponding results of the integration. The integrands \( f_1 \) and \( f_2 \) are defined in lines 19–20.

It is interesting to note that a single mcQuadrature object was used to integrate both functions. Pointers to functions were used in lines 10 and 13 to inform the object which one is the current integrand before invoking the NIntegrate member-function (which actually carries out the integration). The function \( f_1 \) is a regular polynomial, but function \( f_2 \) presents a weak singularity at the origin of the integration domain. The only necessary step necessary to integrate it is to ask the funcInt object to use the cubic transformation before performing the integration (the singular pole was not informed because \( \xi = 0 \) is the default value). This is done in line 14. The output of the program is also shown in Fig. 3.

This elementary example asserts the reusability of the proposed class. By switching the current integrand to the desired subroutine through the setIntegrand member, one can integrate any kernel with no need of further mcQuadrature objects. In addition, the change in the integrand is made during the runtime (no recompiling needed).

### Table 2
Integration results obtained for weakly singular functions

| \( K \) | \( \int_{-1}^{1} \ln(\xi + 1) \, d\xi \) | \( \int_{-1}^{1} \tan|\xi| \, d\xi \) | \( \int_{-1}^{1} K_1(\xi) \, d\xi \) |
|--------|-----------------|-----------------|-----------------|
| Gauss  | Telles          | Gauss  | Telles          | Gauss  | Telles          |
| 4      | -0.550778       | 1.27131 | 1.16672 | -0.328347       | -0.303465 |
| 8      | -0.596184       | 1.24269 | 1.23049 | -0.318003       | -0.305413 |
| 16     | -0.669664       | 1.23653 | 1.23125 | -0.309431       | -0.305101 |
| 24     | -0.611601       | 1.23262 | 1.23125 | -0.307790       | -0.305095 |
| 32     | -0.612510       | 1.23203 | 1.23125 | -0.305920       | -0.305093 |
| 48     | -0.613169       | 1.23160 | 1.23125 | -0.305499       | -0.305093 |
| Analytic| -0.613706       | 1.23125 |        |                 |            |

### Driver program:

```c
1. #include "mc_quadrature.hpp"
2. Matrix mfunc(double,double);
3. int main() {
4.   mcQuadrature<Matrix> matInt(2,2,1);
5.   mcArray ord(3); ord(1) = 1; ord(2) = 2; ord(3) = 3;
6.   cout << endl << "K   Result  " << endl;
7.   for (int i=1; i<=3; i++) {
8.     matInt.setOrder(ord(i));
9.     cout << ord(i) << "  " << matInt.NIntegrate(mfunc) << endl;
10.   }
11. return 0;
12. }
13. }
```

Output:

<table>
<thead>
<tr>
<th>K</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.00   16.00</td>
</tr>
<tr>
<td></td>
<td>16.00 -4.00</td>
</tr>
<tr>
<td>2</td>
<td>4.00   17.33</td>
</tr>
<tr>
<td></td>
<td>17.33 -16.00</td>
</tr>
</tbody>
</table>

Fig. 4. Code example used to integrate matrix kernels.
The example of Fig. 3 was also tested for three illustrative types of weakly singular kernels

\[ f(x) = \log(x + 1) \]

\[ f(x) = \tan|x| \]

\[ f(x) = K_1(x) \]

where \( K_1(x) \) is the first order Bessel function of second kind. Table 2 presents the results obtained for the integration of these functions. All three kernels were integrated by both the standard Gauss–Legendre rules and the cubic transformation for comparison purposes.

Another case is shown in Fig. 4, which presents a variation of the code in Fig. 3, this time being used to integrate the 2x2 matrix

\[ F(\xi, \eta) = \begin{bmatrix} 1 & (\xi + 2)^2(\eta + 1) \\ (\xi + 1)(\eta + 2)^2 & (\xi - 1)^2(\eta + 2)^3 \end{bmatrix} \]

over a \( \xi \times \eta = [-1, +1] \times [-1, +1] \) domain. The code is the same, but the constructor defines a mcQuadrature object (matInt) of matrix type, with two-dimensional entries. F is defined in lines 14–19. Note that the dimensions of the matrix in the integrand are never used by the driver program or by the integration object, releasing the user from looping over each entry. The corresponding output is also shown in the figure.

In FEM implementations, the programming work can be significantly simplified by using the proposed mcQuadrature object, as repetitive tasks like the implementation of Eq. (6a,b,c) can be reduced to a single programming line. Once defined the subroutine, which evaluates the necessary integrands, one can obtain an extremely compact, readable and maintainable code. Fig. 5 shows an extract of an actual FEM code, where the stiffness matrix (Ke), the mass matrix (Me) and the load vector (Fe) are evaluated like according to Eq. (6a,b,c). The statements in lines 2 and 3 create two objects of mcQuadrature type; the first is to integrate

```cpp
1. // Definition of Matrix & Vector integrators (2x2 Gauss rule):
2. mcQuadrature<Matrix> MatIntegrator(2, 2, 1);
3. mcQuadrature<Vector> VecIntegrator(2, 2, 1);
4. // Get the finite elements iterator:
5. fe = FEMModel->getFElements();
6. // Loop over the elements:
7. while (fe) {
8.   // Evaluate the stiffness and mass matrices:
9.   Ke = MatIntegrator.NIntegrate(fe.current->Stiffness);
10.  Me = MatIntegrator.NIntegrate(fe.current->Mass);
11.  Fe = VecIntegrator.NIntegrate(fe.current->Load);
12.  // Assemble:
13.  SysOfEq->AssembleK(Ke, FEMModel->mapDOF(fe.current));
14.  SysOfEq->AssembleM(Me, FEMModel->mapDOF(fe.current));
15.  SysOfEq->AssembleF(Fe, FEMModel->mapDOF(fe.current));
16.  fe++;
17. }
```

Fig. 5. Code excerpt showing the integration and assembly of typical FEM matrices.

```cpp
1:   be = BEMModel->getBElements();
2:   nd = BEMModel->getBVPcollocationPoints();
3:   while (be) {  // Loop over the elements:
4:     while (nd) { // Loop over the collocation nodes:
5:       fundamental_solution = be.current->getFundSol();
6:       fundamental_solution.setLoadPoint(nd.coor());
7:       sing_node = BEMModel->detectSingularNode(*be,*nd);
8:       be->formMatrix(H,"T",sing_node);
9:       be->formMatrix(G,"U",sing_node);
10:      theConstraint->applyBC(*be, *nd, H, G);
11:     nd ++;
12:   }
13:   be ++;
14: }
```

Fig. 6. Code excerpt showing the integration and assembly of typical BEM matrices.
matrices while the second is to integrate vectors. Both stand for two-dimensional Gauss–Legendre quadratures using 2 integration points in each direction. This particular example employs iterators to go through the finite element list during the loop defined in lines 7–17, but suffices to understand that fe.current is merely a pointer pointing to the current finite element. Conventional loop counters can be used as well. The matrices $K_e$ and $M_e$,

```cpp
1: int mcBEElement::formMatrix(Matrix& A, char* id, int const sing) {
2: typedef Matrix (mcBEElement::*mcBEInt3)(char*, double, double, double);
3: mcBEInt3 integrand;
4: mcBESubregion* SR = &this->getOwner();
5: mcFundSolution* FS = &SR->getFundSolution();
6: mcArray typ   = this->getOwner().getFundSolution().getSingType(id);
7: integrand = &(mcBEElement::elemMatrix);
8: mcQuadrature<Matrix> Integrator(ncoor, nip);
9: ngp(i) = SR->estimateNIP(i);
10: Integrator.setOrder(ngp(REGULAR),ngp(REGULAR),ngp(REGULAR));
11: switch (sing) {
12: case 0:
13:  A = Integrator.BEMIntegrate(*this,integrand, id);
14:  break;
15: default: {
16:   for (int i=1; i<= nrows; i++)
17:    for (int k=1; k<= nrows; k++) {
18:      Integrator.setOrder(ngp(REGULAR),ngp(REGULAR),ngp(REGULAR));
19:      switch (typ(i,k)) {
20:        default : {
21:          Integrator.setType(REGULAR,REGULAR,REGULAR);
22:        } break;
23:      Weakly singular entries – Use the Telles’ transformation [28]:
24:      case WEAK: {
25:        Integrator.setType(WEAK,WEAK,WEAK);
26:      } break;
27:      Strongly singular entries – Use the direct method [11]:
28:      case STRONG: {
29:        Integrator.setType(STRONG,STRONG,STRONG);
30:        A(i,j) += F1(i,j) * CorrTerm1 + F2(i,j) * CorrTerm2;
31:      } break;
32:    }
33:  }
34:  A(i,j) += Integrator.BEMIntegrate(*this,integrand,id)(i,j);
35:  return 0;
36: }
37: if (sing) this->addJumpTerm(A, id, sing, angle);
38: return 0;
39: }
```

Fig. 7. Code showing a more elaborate integration procedure to handle different singularities in BEM matrices.
and the vector \( \mathbf{F} \) are returned in lines 9–11, with the numerical integration already accomplished. One should note that the same quadrature object was used to integrate both the \( \mathbf{K} \) and the \( \mathbf{M} \) matrices, since they are of the same algebraic type (but not necessarily of the same size). The \textit{Stiffness}, \textit{Mass} and \textit{Load} words stand for the actual names of the subroutines which evaluates the integrands of the mentioned matrices (i.e. \textit{Stiffness} evaluates the product \( \mathbf{B}^T \mathbf{DB} \) of Eq. (6a), etc.)

A similar structure can be used in BEM applications, as shown in Fig. 6. This piece of code is used to generate the global system of Eq. (4) and any domain loads were disregarded for simplicity’s sake. Following a classical BEM implementation two loops are used, one run through the boundary collocation points, and another for the boundary elements.

Inside these loops, the Eqs. (7a) and (7b) are evaluated by a member called \texttt{formMatrix} (lines 8–9). This member integrates a general kernel of the form \( \mathbf{A}(p,q) \mathbf{N}(q) J \), where \( \mathbf{A} \) is a fundamental solution tensor. The second argument in the \texttt{formMatrix} call indicates which one to use (‘\( T \)’ for the traction fundamental solution \( \mathbf{T} \), etc.). The member \texttt{applyBC} invoked in line 10 enforces the boundary conditions to generate the final system of equations.

Unfortunately, not all fundamental solution tensors behave with the same degree of singularity for all its components. For instance, in the traction fundamental solution of the Mindlin plate model all the components are either regular or weakly singular except for \( T_{12} \) and \( T_{21} \) which are strongly singular [30]. In such cases, a single call to the \texttt{NIntegrate} member function in order to integrate the whole (matrix) kernel at once would not work properly.

In such cases, the \texttt{formMatrix} member of Fig. 6 has to be implemented in a more elaborate way. Fig. 7 illustrates an excerpt of an actual code, where each entry of \( \mathbf{A}(p,q) \) is verified against its singularity type, and then a suitable type of quadrature is chosen for each component.

As mentioned in the end of the previous section, object member functions may have limited visibility or be vulnerable to changes. Since it is interesting to require little commonality between the present class design and the code under development, function wrappers [12] can be used to fit the proposed class into independently developed parts. When using a function wrapper, any changes in the interface of the user’s class is reflected only in the definition of its corresponding function wrapper. Other parts of the program are not affected by the change. This is the approach suggested here, particularly for BEM codes like the one shown in Fig. 7. Lines 2 and 3 of the code are used along with line 24 of Fig. 1 to wrap an independent (boundary element class) member function which evaluates BEM matrices like the ones in Eq. (7a,b,c).

Solutions like those illustrated in Fig. 7 will loose some of the versatility of the proposed class. However, one must remember that most fundamental solutions presents the same type of singularity in all its components, and in these cases the code of Fig. 7 will be greatly simplified.

### 5. Efficiency

It is interesting to address some points about the efficiency of the proposed approach. As aforementioned, the \texttt{mcQuadrature} class demands the use of an embedded or a third party class library to handle non-native types (type \( T \) in Fig. 1) like vectors and matrices. Therefore, it is obvious that any issues related to the efficiency of the design are highly dependent of the implementation of such classes. Unfortunately, the present implementation used a well-known class library for matrices, which is reportedly slow for very small matrices/vectors [7]. Even so, a number of tests were carried out to sample the elapsed time to evaluate three integrals

\[
\begin{align*}
I_1 &= \int_{-1}^{+1} \xi^{10} d\xi \\
I_2 &= \int_{-1}^{+1} \left( \xi^{3} (\xi + 5)^{-1} \right) d\xi \\
I_3 &= \int_{-1}^{+1} \int_{-1}^{+1} A d\xi_1 d\xi_2
\end{align*}
\]

where \( A_{ij} = (i-1)^{2} \varepsilon_{2}^2 + (j-1)^{2} \varepsilon_{2}^2 \). In order achieve significant elapsed time digits, the integrals \( I_1 \) and \( I_2 \) were evaluated 100,000 times, while \( I_3 \) was evaluated 200 times. The number of integration points used in \( I_3 \) was the same in both directions. Three methods were used:

- \textit{Method A}. The quadrature stations and weights are returned from a function and used in a cumulative loop, which calls the integrand function using the current stations as parameters. The driver code used was very similar to ordinary C or Fortran procedures, i.e. loops were used to access each matrix/vector entry individually, but the stations and weights (as well as the integrands in cases \( I_2 \) and \( I_3 \)) were

![Fig. 8. Normalized elapsed time to evaluate \( I_1 \) 100,000 times.](image-url)
stored using the aforementioned matrix/vector class library. The stations and weights used in the loops were called only once and reused. This situation simulates cases, where the required quadrature order is fixed and known a priori.

Method B. Similar to method A, but in this case the stations and weights were called every time the integrals were evaluated. This situation simulates cases, where the required quadrature order is variable, and estimated during the run time.

Method C. A single mcQuadrature object was created and used repeatedly through its NIntegrate method. This situation simulates cases, where the required quadrature order is variable, and estimated during the run time (typical in BEM codes).

The results obtained for these tests are shown in Figs. 8–10, for $I_1$, $I_2$ and $I_3$, respectively. In all cases, the elapsed time of method A was used to normalize the results, since this method resembles the usual programming style found in many engineering software. Fig. 8 shows that the use of the proposed class can be competitive with method A for scalar functions, particularly for higher quadrature orders. It is worth to note it can perform up to three times faster than method B. On the other hand, the efficiency of the design to integrate a simple three rows vector ($I_2$) resulted disappointing in comparison to the other methods. This is shown in Fig. 9 and is related to the nature of the matrix/vector class library used, as mentioned before. It was expected a better performance to integrate vectors/matrices or larger orders, as shown in Fig. 10. The use of the mcQuadrature class resulted in the shortest elapsed times among all methods. This behavior was consistently observed in matrix integrands of orders larger than $4 \times 4$ and vectors with more than 8 rows. Although efficiency issues always show some degree of dependence on the way the compiler generates the code, similar behaviors have been observed in other high-performance object-oriented frameworks (see, for instance, [23]). The lower performance for small vectors/matrices integrands can be improved by using matrix class libraries that perform better in these cases. Yet, the maintainability and readability of the code can compensate this aspect in applications, where the integration time is not important.

6. Conclusions

This work presented a basic template class for general numerical integration implementation. The approach supports many types of integrand objects, and is able to perform both regular and singular integrations under the same code structure, suggesting a fair solution for an old BEM implementation drawback. More importantly, a single object can be used throughout a FEM or BEM code to perform most numerical integration tasks using one statement for vector or matrix type integrands. Additional cases and special quadrature methods can be included as subclasses with little effort. The proposed class provides a unified numerical integration implementation allowing a high degree of flexibility and shortening the duration of the software development process. The elementary examples shown illustrate straightforward applicability for existing FEM and BEM codes, but general engineering and physics software can use the idea as well. Some issues related to the efficiency of the approach were also addressed.

Acknowledgements

The author wishes to express his gratitude to both reviewers for suggesting relevant changes in the original manuscript.

References


