

## Integrable Objects for Numerical Computation with H0, H1 and H2 Types

We refer to [Workbook of Applications in VectorSpace C++ Library](#) for a more complete set of features in VectorSpace C++ Library. An over-simplified presentation, if you would, follows.

The inspiration for the object-oriented modeling of an integrable object comes from the observations of various quadrature rules, such as trapezoid rule, simpson's rule, and Gauss quadrature. For example, the analytical integration of

$$\int_1^2 (x-1)^2 dx = \int_1^2 (x^2 - 2x + 1) dx = \left[ \frac{x^3}{3} - x^2 + x \right]_1^2 = \frac{1}{3}$$

Numerical integration such as (extended) trapzoid rule, simpson's rule, ...etc., with "N" equal-sized intervals can all be written as

$$\sum_{i=0}^N hc_i f(x_i)$$

where "f(x)" ( $\equiv (x-1)^2$ ) is the integrand, " $x_i$ " ( $\equiv i * h + x_0$ ) is an array of N+1 evaluation points for the quadrature, "h" is the size of the intervals, and  $c_i$  is an array of coefficients depends on a specific quadrature rule. For exmple, the [extended Simpson's rule](#) with 11 intervals can be implemented in C++ code as

```
const int N = 11;
double c[N] = {1./3., 4./3., 2./3., 4./3., 2./3., 4./3., 2./3., 4./3., 2./3., 4./3., 1./3.},
X[2] = {1.0, 2.0}, h = (X[1]-X[0])/N;
Quadrature qp(c, X[0], X[1], N); // domain [1.0, 2.0], "N" integration points
H0 x(qp);
C0 area = (x.pow(2) - 2 * x + 1) | J(h); // or "C0 area = (x-1).pow(2) | J(h);"
cout << area << endl; // 0.333334
```

The class "Quadrature" is used to pass the essential information for instantiating an integrable H0 type object. The reason for class "J", the Jacobian, will become obvious for Gauss quadrature later. For now, it only gives the size of the equal-size intervals, "h". Gauss quadrature in its basic form is given in a normalized region, e.g., at a interval of  $\xi \equiv [-1, 1]$ . For Gauss quadrature to be performed on an arbitrary interval  $x \equiv [x_0, x_1]$ , we need to define a coordinate transformation rule. For example, the linear interpolation can be used

$$x(\xi) = \frac{1}{2}(1 - \xi)x_0 + \frac{1}{2}(1 + \xi)x_1$$

Without lost of generalization to multi-dimensional cases, the Gauss quadrature rule is

$$\int f(\mathbf{x}) d\mathbf{x} = \int f(\mathbf{x}) \det\left(\frac{\partial \mathbf{x}}{\partial \xi}\right) d\xi = \int f(\mathbf{x}) J d\xi \equiv \sum_{i=0}^N W_i \tilde{f}(\xi_i) J_i$$

where "the Jacobian",  $J \equiv \det\left(\frac{\partial \mathbf{x}}{\partial \xi}\right)$ , and  $W_i$  are the weighting coefficients on Gauss integration points  $\xi_i$ . The [Gauss quadrature](#) in C++ code is simply implemented as

```
double X[2] = {1.0, 2.0};
Quadrature qp(1, 2);
```

```

H0 z(qp), // coordinate  $\xi$  for normalized domain
    x = (1-z)/2*X[0] + (1+z)/2 * X[1]; //  $\bar{x}(\xi) = (1/2)(1-\xi)x_0 + (1/2)(1+\xi)x_1$ 
J omega((X[1]-X[0])/2); //  $J = d\bar{x}/d\xi = (x_1-x_0)/2$ 
C0 area = (x-1).pow(2) | omega;
cout << area << endl; // 0.333334

```

In this case, “the Jacobian”,  $J = (x_1-x_0)/2$ , is constant throughout the entire integration domain. A logic next step for integrable object application in one dimension is the [solution of differential equation using Green’s function](#). The following example is taken from [Stakgold, I. 1979. “Green’s function and boundary value problems”](#). Considering a general diffusion problem, e.g., heat conduction or chemical diffusion, in the form of a *differential equation* is

$$\frac{d^2 u}{dx^2} = f(x), \quad \text{with } 0 < x < 1, u(0) = \alpha, u(1) = \beta$$

where  $f(x)$  is the source term. This differential equation can be solved using [Green’s function method](#) with the solution as

$$u(x) = \int_0^1 g(x, \xi) f(\xi) d\xi + (1-x)\alpha + x\beta$$

where  $g(x, \xi)$  is the Green’s function. The physical interpretation of the Green’s function is that the  $g(x, \xi)$  is the *temperature (or concentration) sampling at  $x$  when a unit concentrated point source is located at  $\xi$* . Therefore,  $g(x, \xi)$  satisfies the original differential equation; i.e., with  $g(x, \xi)$  in place of  $u(x)$  in the differential equation. We also require, at the location  $x = \xi$ ,  $g(x, \xi)$  to be continuous. And the net flux of the infinitesimal control line segment at  $x = \xi$  equals to the source intensity; i.e.,  $g'(\xi^+, \xi) - g'(\xi^-, \xi) = -1$ , which is also known as the *jump condition*. From the above conditions, the Green’s function can be solved for as

$$g(x, \xi) = \begin{cases} x(1-\xi), & 0 < x < \xi \\ \xi(1-x), & \xi < x < 1 \end{cases}$$

Therefore, the solution becomes

$$u(x) = (1-x) \int_0^x \xi f(\xi) d\xi + x \int_x^1 (1-\xi) f(\xi) d\xi + (1-x)\alpha + x\beta$$

where at interval  $[0, x]$  of the integration domain  $\xi < x$ , so “ $g(x, \xi) = \xi(1-x)$ ” is applicable, and at the interval  $[x, 1]$  of the integration domain,  $x < \xi$ , “ $g(x, \xi) = x(1-\xi)$ ” is applicable. For a specific case with source distribution as  $f(x) = \sin(\pi x)$ , and homogeneous boundary conditions  $\alpha = \beta = 0$ , we can compute ten point values of the solution  $u(x)$  at interval of “ $h = 0.1$ ” with extended Simpson’s integration rule as

```

double const PI = 3.141592654; double const alpha = 0.0; double const beta = 0.0;
double x = 0.0,
    w[11] = {1.0/3.0, 4.0/3.0, 2.0/3.0, 4.0/3.0, 2.0/3.0, 4.0/3.0, 2.0/3.0, 4.0/3.0, 2.0/3.0,
             4.0/3.0, 1.0/3.0};
for(int i = 0; i < 11; i++) {
    Quadrature q1(w, 0.0, x, 11), q2(w, x, 1.0, 11);
    H0 z1(q1), z2(q2),
        f_1 = sin(PI*z1), f_2 = sin(PI*z2);
    C0 integ_1, integ_2;

```

```

if(i !=0) integ_1 &= (1-x)*( z1*f_1) | J(x/10.0); else integ_1 &= C0(0.0);
if(i !=10) integ_2 &= x*( ((1-z2)*f_2) | J((1-x)/10.0); else integ_2 &= C0(0.0);
double u = (double)(integ_1+integ_2) + (1-x)*alpha+x*beta;
cout << "u(" << x << ")": " << u << endl;
if(i != 10) x += 0.1;
}

```

The point-wise numerical solutions are compared to analytical solution, “ $-\sin(\pi x)/\pi^2$ ”, in the following table (due to symmetry only the first half, from 0.0 to 0.5, are listed).

| x                     | 0.0 | 0.1      | 0.2      | 0.3      | 0.4      | 0.5      |
|-----------------------|-----|----------|----------|----------|----------|----------|
| $-\sin(\pi x)/\pi^2$  | 0   | 0.031310 | 0.059555 | 0.081971 | 0.096362 | 0.101321 |
| <b>Green function</b> | 0   | 0.031311 | 0.059555 | 0.081970 | 0.096361 | 0.101320 |

The numerical results differ from analytical solution on six-th digit after the decimal point. The above three examples only use H0 type objects, which are integrable but not differentiable. H1 and H2 types generate integrable objects which once and twice differentiable, respectively. Back to the Gauss quadrature problem, the C++ code is

```

double x[3] = {1.0, 1.5, 3.0};           // x = [1, 3]
Quadrature qp(1,3);
H1 zai(qp),                             // ξ
    X = -(1-zai)*zai/2.0 * x[0] +
        (1-zai)*(1+zai) * x[1] +
        zai*(1+zai)/2.0 * x[2];

C0 v = (((H0)X)-1).pow(2) | J(d(X));    // analytical integration  $\int_1^2 (x-1)^2 dx = 2\frac{2}{3}$ 
cout << v << endl;                     // 2.66667,

```

where quadratic interpolation functions are used for coordinate transformation,

$$x(\xi) = -\frac{1}{2}(1-\xi)\xi x_0 + (1-\xi)(1+\xi)x_1 + \frac{1}{2}\xi(1+\xi)x_2$$

“the Jacobian”,  $J$ , will not be constant throughout the integration domain. In 1-D  $J = dx(\xi)/d\xi$ . Since the coordinate transformation function “ $x(\xi)$ ” is now a H1 type object, a free function call “ $d(x)$ ” gives the derivative information of “ $x(\xi)$ ”.

Next we show example of **line-integration** on a 2-D curve with (1) **arc-length method** commonly introduced in every Calculus text, and (2) **coordinate transformation method** found in finite element and boundary element methods. We seek the length of a circle in the first quadrant

$$x^2 + y^2 = 1$$

The value of the ordinate “ $y$ ” can be considered as a function of “ $x$ ” as

$$f(x) \equiv y = \sqrt{1-x^2}$$

The arc length, “ $s$ ”, of a function  $f(x)$  is

$$s \equiv \sum_i \sqrt{(\Delta x)^2 + |f(x_{i+1}) - f(x_i)|^2}$$

by applying Pythagorean law to calculate every infinitesimal arc-length (hypotenuse) with its two legs:  $\Delta x$  in x-direction and  $|f(x_{i+1})-f(x_i)|$  in y-direction. For differentiable function  $f(x)$  at the limit of  $\Delta x \rightarrow 0$ , we have the arc-length method formula

$$s = \int \sqrt{1 + f'(x)} dx$$

Next we use the coordinate transformation method. A point in a curved element is represented by a position vector  $\mathbf{r} = \{x, y\}^T$ . we use quadratic interpolation functions for coordinate transformation of position vector,  $\mathbf{r}$ , with three nodal coordinates  $(x_0, y_0)$ ,  $(x_1, y_1)$ , and  $(x_2, y_2)$ .

$$x(\xi) = -\frac{1}{2}(1-\xi)\xi x_0 + (1-\xi)(1+\xi)x_1 + \frac{1}{2}\xi(1+\xi)x_2$$

$$y(\xi) = -\frac{1}{2}(1-\xi)\xi y_0 + (1-\xi)(1+\xi)y_1 + \frac{1}{2}\xi(1+\xi)y_2$$

where  $\xi$  is the parameter for the coordinate transformation of  $x$  and  $y$  coordinates. An infinitesimal length of the curve  $d\mathbf{r}$  can be obtained as

$$d\mathbf{r} = \sqrt{\left(\frac{dx}{d\xi}\right)^2 + \left(\frac{dy}{d\xi}\right)^2} d\xi$$

by applying the Pythagorean law. The integrand in the right-hand-side is

$$\sqrt{\left(\frac{dx}{d\xi}\right)^2 + \left(\frac{dy}{d\xi}\right)^2} = \left\| \frac{d\mathbf{r}}{d\xi} \right\|_2 = \|\mathbf{J}\|_2$$

That is the Euclidean norm of Jacobian of the coordinate transformation rule. Therefore, we have

$$d\mathbf{r} = \|\mathbf{J}\|_2 d\xi$$

This integration formula is simply written consistent with the *coordinate transformation method*.

```

int NODE_NO = 5;
const double r = 1.0; const double PI = 3.141592654;
double X[NODE_NO][2];
for(int i = 0; i < NODE_NO; i++) {
    X[i][0] = r*cos(((double)(i+1))*PI/(2.0*(NODE_NO-1)));
#ifndef __ARC_LENGTH
    X[i][1] = r*sin(((double)(i+1))*PI/(2.0*(NODE_NO-1)));
#endif
}
Quadrature qp(1, 4);
H1 zai(qp),
    N = INTEGRABLE_VECTOR_OF_TANGENT_BUNDLE(
    "int, int, Quadrature", 3, 1, qp),
    x, y;
N[0] = -zai*(1-zai)/2; N[1] = (1+zai)*(1-zai); N[2] = zai*(1+zai)/2;
C0 length = 0.0;
for(int i = 0; i < (NODE_NO-1)/2; i++) {
    x = N[0]*X[i+2][0]+N[1]*X[i+1][0]+N[2]*X[i][0];
#if defined(__ARC_LENGTH)
    y = sqrt(pow(r,2)-x.pow(2));
    length += (sqrt(1.0+(d(y)/d(x)).pow(2)) | J(d(x)));

```

```

#else
    y = N[0]*X[i+2][1]+N[1]*X[i+1][1]+N[2]*X[i][1];
    length += (sqrt(d(x).pow(2)+d(y).pow(2))) | J(1.0);
#endif
}
cout << length << endl;

```

The macro definitions “ARC\_LENGTH” set the code to run the arc-length method. Otherwise, the coordinate transformation is used. The arc-length method gives length = 1.57079, and the coordinate transformation methods gives length = 1.57020. The analytical result for the length of a quadrant of a circle with unit radius is  $2\pi(1)^2/4 \cong 1.570796327$ . The The function “sqrt()” is only defined for C0 and H0 types but that can be extended easily for H1 type by an *ad hoc* implementation of a free function “sqrt()” such as

```

H1 sqrt(const H1& a) {
    return INTEGRABLE_TANGENT_BUNDLE(
        “const H0&, const H0&”, sqrt((H0)a), d(a)/2.0/sqrt((H0)a) );
}

```

Next we show examples of [surface and volume integration in 3-D](#). That is to integrate the *volume* and *surface area* of a sphere with unit radius, numerically. The analytical value of volume of a sphere is  $\frac{3}{4}\pi r^3 \cong 4.188790203$  (where  $r = 1$ ), and surface area of an unit sphere is  $4\pi r^2 \cong 12.56637061$  ( $r=1$ ). The nodal coordinates of any point on an unit sphere can be computed using spherical (polar) coordinates as

$$x_0 = r \sin \phi \cos \theta, \quad x_1 = r \sin \phi \sin \theta, \quad \text{and} \quad z = r \cos \phi$$

where  $r$  is the *radius* of a sphere,  $\phi$  (*latitude*) is the angle between the point to the Z-axis,  $\theta$  (*colatitude*) is the angle obtained by first projecting the point on the  $x_0$ - $x_1$  plane, then, measuring the angle of the projected point and the polar axis on the  $x_0$ - $x_1$  plane.

We use 9-node Lagrangian element in the following figure, which is popular in finite element method. The case presented here with the interpolation functions the same as the functions used for coordinate transformation is known as *isoparametric element*. The algorithm presented here, however, is applicable to 4 to 9 nodes element. With this algorithm, any number of nodes in fifth to ninth nodes can be added or omitted. The order of the node number and the interpolation functions defined in the above are both accepted standard in finite element method. A quarter of a circle is taken as a whole element or subdivided into three quadrilateral elements.

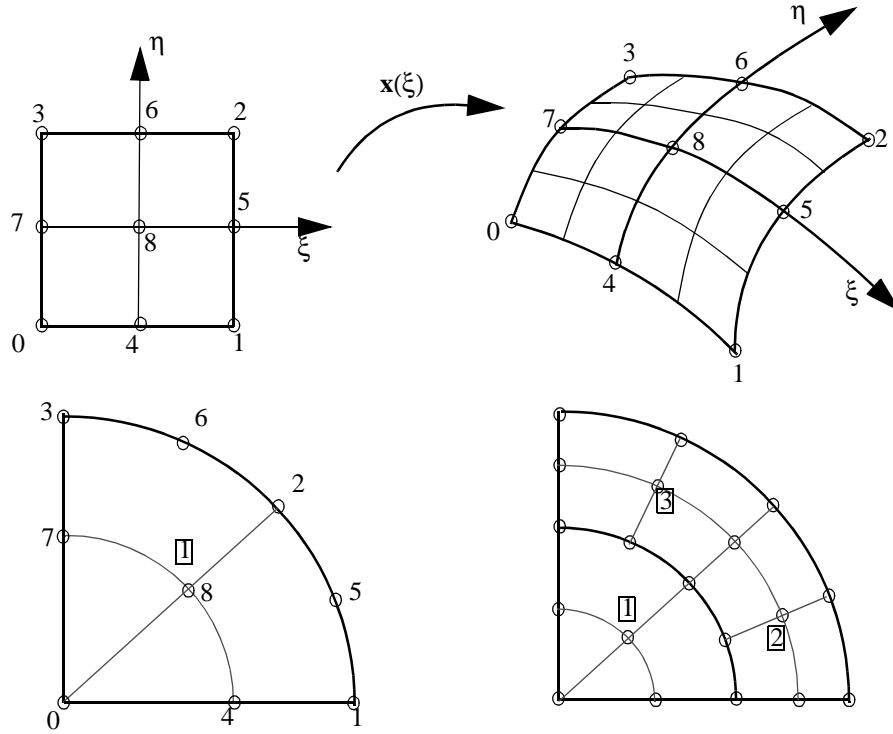
The *coordinate transformation rule* for a nodal point  $\mathbf{x} = (x_0, x_1)$  is now

$$\mathbf{x}(\xi, \eta) \equiv \mathbf{N} \bullet \bar{\mathbf{x}}$$

where the components of vector  $\mathbf{N}$  are the shape (interpolation) functions, and  $\bar{\mathbf{x}} = (\bar{x}_0, \bar{x}_1)$  are nodal coordinates. The *interpolation function* for the z-coordinate of a point on the surface of a sphere can be expressed similarly as

$$\mathbf{z}(\xi, \eta) \equiv \mathbf{N} \bullet \bar{\mathbf{z}}$$

where  $\bar{\mathbf{z}}$  is the height of the spherical surface above a node. For geometrical entities that are simple enough, as in this case with the unit sphere, their algebraic expression can be found in *analytical geometry*. Instead of the the z-coordinate transformation rule, the height of the unit sphere can be written as



**Step 1:** Define interpolation functions  $N_i$  for four corner nodes ( $i = 0, 1, 2, 3$ ) with

$$\begin{aligned} N_0(\xi, \eta) &= \frac{1}{4} (1-\xi)(1-\eta), & N_1(\xi, \eta) &= \frac{1}{4} (1+\xi)(1-\eta) \\ N_2(\xi, \eta) &= \frac{1}{4} (1+\xi)(1+\eta), & N_3(\xi, \eta) &= \frac{1}{4} (1-\xi)(1+\eta) \end{aligned} \quad \text{Eq. 3•1}$$

**Step 2:** If no center node, skip the Step 2. (1) add center node by

$$N_8(\xi, \eta) = (1-\xi^2)(1-\eta^2) \quad \text{Eq. 3•2}$$

Then, (2) modification of four corner nodes due to the presence of the center node

$$N_0 \text{ -} N_8/4, N_1 \text{ -} N_8/4, N_2 \text{ -} N_8/4, N_3 \text{ -} N_8/4, \quad \text{Eq. 3•3}$$

**Step 3:** For each four edge nodes ( $i = 4, 5, 6, 7$ ), do the followings. If the edge node present,

(1) add the edge node by the corresponding interpolation function

$$\begin{aligned} N_4(\xi, \eta) &= (1-\xi^2)(1-\eta) - N_8/2, & N_5(\xi, \eta) &= (1-\eta^2)(1+\xi) - N_8/2 \\ N_6(\xi, \eta) &= (1-\xi^2)(1+\eta) - N_8/2, & N_7(\xi, \eta) &= (1-\eta^2)(1-\xi) - N_8/2 \end{aligned} \quad \text{Eq. 3•4}$$

Then, (2) modification of the four corner nodes due to the presence of the four edge nodes

$$N_0 \text{ -} (N_4+N_7)/2, N_1 \text{ -} (N_4+N_5)/2, N_2 \text{ -} (N_5+N_6)/2, N_3 \text{ -} (N_6+N_7)/2, \quad \text{Eq. 3•5}$$

$$z(\xi, \eta) = \sqrt{1 - (x_0)^2 - (x_1)^2}$$

The *volume* of one-eighth of a sphere can be obtained by

$$\text{volume} = \int_{\mathbf{x}} z(\xi) d\mathbf{x}$$

where  $\xi$  is a vector =  $\{\xi, \eta\}^T$ . Either, the *isoparametric interpolation* or *analytical expression* of can be used for “z” to evaluate the volume in the last equation. A three dimensional *surface area* of a two dimensional integration domain has its formula as a multiple dimensional generalization to that of the *arc length* method; i.e.,

$$\text{surface area} = \int_{\mathbf{x}} \sqrt{1 + \left(\frac{\partial z}{\partial x_0}\right)^2 + \left(\frac{\partial z}{\partial x_1}\right)^2} d\mathbf{x}$$

where

$$\frac{dz}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial z}{\partial x_0} \\ \frac{\partial z}{\partial x_1} \end{bmatrix} = \frac{dz}{d\xi} \left( \frac{d\mathbf{x}}{d\xi} \right)^{-1}$$

The C++ code is implemented as

```
#include "include\vs.h"
double PI = 3.141592654, deg = PI/180.0, theta = 90.0, phi = 90.0;
void sphere(double* X, double& Z, double phi, double theta) {
X[0] = sin(phi*deg)*cos(theta*deg); X[1] = sin(phi*deg)*sin(theta*deg); Z = cos(phi*deg); }
void define_element() {
#if defined(__THREE_ELEMENTS)
    const int ELEMENT_NO = 3; double X[ELEMENT_NO][9][2], Z[ELEMENT_NO][9];
    sphere(X[0][0], Z[0][0], 0.0, 0.0); sphere(X[0][1], Z[0][1], 1.0/2.0*phi, 0.0);
    sphere(X[0][2], Z[0][2], 1.0/2.0*phi, 1.0/2.0*theta); sphere(X[0][3], Z[0][3], 1.0/2.0*phi, theta);
    sphere(X[0][4], Z[0][4], 1.0/4.0*phi, 0.0); sphere(X[0][5], Z[0][5], 1.0/2.0*phi, 1.0/4.0*theta);
    sphere(X[0][6], Z[0][6], 1.0/2.0*phi, 3.0/4.0*theta); sphere(X[0][7], Z[0][7], 1.0/4.0*phi, theta);
    sphere(X[0][8], Z[0][8], 1.0/4.0*phi, 1.0/2.0*theta); sphere(X[1][0], Z[1][0], 1.0/2.0*phi, 0.0);
    sphere(X[1][1], Z[1][1], phi, 0.0); sphere(X[1][2], Z[1][2], phi, 1.0/2.0*theta);
    sphere(X[1][3], Z[1][3], 1.0/2.0*phi, 1.0/2.0*theta); sphere(X[1][4], Z[1][4], 3.0/4.0*phi, 0.0);
    sphere(X[1][5], Z[1][5], phi, 1.0/4.0*theta); sphere(X[1][6], Z[1][6], 3.0/4.0*phi, 1.0/2.0*theta);
    sphere(X[1][7], Z[1][7], 1.0/2.0*phi, 1.0/4.0*theta);
    sphere(X[1][8], Z[1][8], 3.0/4.0*phi, 1.0/4.0*theta);
    sphere(X[2][0], Z[2][0], 1.0/2.0*phi, theta);
    sphere(X[2][1], Z[2][1], 1.0/2.0*phi, 1.0/2.0*theta);
    sphere(X[2][2], Z[2][2], phi, 1.0/2.0*theta); sphere(X[2][3], Z[2][3], phi, theta);
    sphere(X[2][4], Z[2][4], 1.0/2.0*phi, 3.0/4.0*theta);
    sphere(X[2][5], Z[2][5], 3.0/4.0*phi, 1.0/2.0*theta);
    sphere(X[2][6], Z[2][6], phi, 3.0/4.0*theta); sphere(X[2][7], Z[2][7], 3.0/4.0*phi, theta);
    sphere(X[2][8], Z[2][8], 3.0/4.0*phi, 3.0/4.0*theta);
#else
```

```

const int ELEMENT_NO = 1;
double X[ELEMENT_NO][9][2], Z[ELEMENT_NO][9];
sphere(X[0][0], Z[0][0], 0.0, 0.0);           sphere(X[0][1], Z[0][1], phi, 0.0);
sphere(X[0][2], Z[0][2], phi, 1.0/2.0*theta); sphere(X[0][3], Z[0][3], phi, theta);
sphere(X[0][4], Z[0][4], 1.0/2.0*phi, 0.0);   sphere(X[0][5], Z[0][5], phi, 1.0/4.0*theta);
sphere(X[0][6], Z[0][6], phi, 3.0/4.0*theta); sphere(X[0][7], Z[0][7], 1.0/2.0*phi, theta);
sphere(X[0][8], Z[0][8], 1.0/2.0*phi, 1.0/2.0*theta);
#endif
}
int main() {
    define_element();
    Quadrature qp(2, 9);
    H1 ZAI(2, (double*)0, qp),
    N = INTEGRABLE_VECTOR_OF_TANGENT_BUNDLE(
        "int, int, Quadrature", 9, 2, qp), Zai, Eta;
    Zai &= ZAI[0]; Eta &= ZAI[1];
    N[0] = (1-Zai)*(1-Eta)/4; N[1] = (1+Zai)*(1-Eta)/4;
    N[2] = (1+Zai)*(1+Eta)/4; N[3] = (1-Zai)*(1+Eta)/4;

    N[8] = (1-Zai.pow(2))*(1-Eta.pow(2));
    N[0] -= N[8]/4; N[1] -= N[8]/4; N[2] -= N[8]/4; N[3] -= N[8]/4;

    N[4] = ((1-Zai.pow(2))*(1-Eta)-N[8])/2; N[5] = ((1-Eta.pow(2))*(1+Zai)-N[8])/2;
    N[6] = ((1-Zai.pow(2))*(1+Eta)-N[8])/2; N[7] = ((1-Eta.pow(2))*(1-Zai)-N[8])/2;
    N[0] -= (N[4]+N[7])/2; N[1] -= (N[4]+N[5])/2;
    N[2] -= (N[5]+N[6])/2; N[3] -= (N[6]+N[7])/2;

    C0 vol(0.0), area(0.0);
    for(int i = 0; i < ELEMENT_NO; i++) {
        C0 X_(9, 2, X[i][0]);
        H1 x = N*X_;
        C0 Z_(9, Z[i]);
    #if defined(__SURFACE_AREA)
        H1 z = N*Z_;
        H0 dz_dx = d(z) * d(x).inverse(),
        da = sqrt((dz_dx[0]).pow(2)+(dz_dx[1]).pow(2)+1.0);
        area += da | J(d(x).det());
    #endif
    }
    cout << (8.0*90.0/theta*area) << endl;
#else
#if defined(__ANALYTICAL_GEOMETRY)
        H0 z = sqrt(1-((H0)x[0]).pow(2)-((H0)x[1]).pow(2));
#else
        H0 z = ((H0)N)*Z_;
#endif
    vol += z | J(d(x).det());
}

```

```

    }
    cout << (8.0*90.0/theta*vol) << endl;
#endif
    return 0;
}

```

The macro definition “\_\_THREE\_ELEMENTS”, if defined, the program discretizes one-quarter of a circular domain into three nine-nodes quadrilateral elements, otherwise, only one element is used. The macro definition “\_\_ANALYTICAL\_GEOMETRY”, use analytical instead of coordinate transformation. The macro definition “\_\_SURFACE\_AREA”, if defined, compute the surface area instead of the volume. The results of this computation are shown in the following table. Three elements discretization does improve the accuracy of the integration as compared to that of the one element discretization.

|            | <b>Volume</b> | <b>Surface Area</b> |
|------------|---------------|---------------------|
| 1 Element  | 3.99991       | 12.3264             |
| 3 Elements | 4.17283       | 12.5465             |
| Analytical | 4.18879       | 12.5664             |