Serdica J. Computing **14** (2020), No 1–2, 1–12

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

NUMERICAL INTEGRATION OF TWO-DIMENSIONAL COMPLEX-VALUED FUNCTIONS FOR THE NEEDS OF BIEM*

Lasko M. Laskov

Abstract. Numerical integration is a key problem with numerous applications, including the boundary integral equation method (BIEM). GSL [7] provides powerful implementations of a broad variety of well-known numerical methods and algorithms. However, its routines have certain limitations: the numerical integration based on QUADPACK library [2] is implemented only in the case of one-dimensional functions.

In this paper we present an extension of the GSL numerical integration routines for the special case of two-dimensional complex-valued functions. The presented approach is part of our effort to build a BIEM software system for solving a 3D elastodynamics problem for wave propagation in a continuously inhomogeneous half-space.

ACM Computing Classification System (1998): G.1, G.4.

Key words: numerical integration, two-dimensional complex-valued functions, GSL C++ wrapper.

*This work is supported by research Grant IB-RA2014-178-EnTranEmiss from the Federal Ministry of Education Research in Germany.

1. Introduction. Numerical integration is a well-known problem in numerical analysis [8], however the practical approach to the implementation of algorithms, even in the case of standard methods, is not a straightforward problem [6]. Especially in the case of "badly-behaved" integrands [1], and functions that contain singularities, special care must be taken in the implementation of numerical methods. In this way, the investigation of such algorithms and the implementation of programs turns to be an extensive project on its own. For this reason, in many practical applications, reliable software systems are adopted when numerical calculation of integrals are needed, especially when the integrands are actually derived from real-world applications.

Depending on the needs of the experiment, such a software system can be the environment for technical computing and numerical analysis MATLAB or its open-source alternative GNU Octave [10]. Another powerful approach is relying on a system for symbolic calculations, such as Wolfram Mathematica [11, 54–62]. The latter is extremely appropriate when solving problems that are mathematically formulated, and can perform the needed experiments for rather complex models, such as in the work of Marinov and Rangelov [9]. The use of systems for technical computing and symbolic calculations is also an irreplaceable tool for method verification and prototyping, and is often the preliminary stage prior to the implementation of the software that provides the final tool needed for the numerical experiment.

In some cases the use of computational systems like Mathematica is not enough, and specific programs need to be implemented. These are applications that need specific methods and algorithms, and also that have to process bigger amounts of data, or applications that are designed to serve specific user needs. In the latter case, the software is supposed to answer all the requirements of a complete software system.

The implementation of scientific software using a programming language like Fortran, C/C++, Java or Python can be a complicated and time-consuming task. For this reason, when standard methods and algorithms need to be implemented, it is better to choose an appropriate scientific library that provides a set of routines. The implementation of such a library is reliable, robust and sufficiently tested, so that actually the effort to re-implement its functionalities is justified only if these concrete methods are the subject of the investigation. A good example is the FFTW ("Fastest Fourier Transform in the West") [5] that is even used by commercial software systems like MATLAB.

Numerical integration itself completely falls into this category of standard algorithms, and a number of numerical computational libraries exist that provide

such a set of routines. For our project, the most appropriate choice is GSL [7], for a number of reasons. First of all, GSL is well-known and reliable tool. It is written in the programming language C, while our project is written in C++, which makes them compatible. Also, GSL provides a broad set of tools besides numerical integration, which are also needed in our implementation. And lastly, GSL is part of the GNU Project and is available under GNU General Public License.

Numeric integration in GSL is based on the QUADPACK library [2], which was originally written in FORTRAN 77. GSL provides the C re-implementation of these routines that include algorithms for both non-adaptive and adaptive integration of general functions. The library covers the following special cases: quadrature over semi-infinite ranges, integration of functions that contain singularities, computation of Cauchy principal values, and integrands with an oscillatory factor [7]. Regardless of the wide range of provided routines, there is the significant restriction that all of them calculate the quadrature of one-dimensional functions.

For many applications, and for our particular application as well, the integration of much more complicated integrands is needed. Besides, the GSL routines are written in the language C, and there is the additional restriction that they work only with global and static functions, when an address of a function has to be passed as an input of a routine. In this paper we describe an approach to extend the functionalities of the GSL integration routines to the case of complex-valued two-dimensional functions. The same approach is applicable if the integrand is a function of higher dimension. Also, we show an approach to extend the functionalities of the GSL integration routines in a C++ project.

The presented approach is a part of our implementation of boundary integral equation method (BIEM) for solving a 3D elastodynamics problem for wave propagation in a continuously inhomogeneous half-space.

2. GSL Integration of a One-dimensional Function.

We start with the basic example of numeric integration of a function $f(x)$ over $[a, b]$ which is the standard GSL one-dimensional function quadrature calculation. We will use the QAGS adaptive integration with singularities, but any other routine in the library can be applied in absolutely analogous way. In the next sections we will extend this example first to integration of two-dimensional function, and then to integration of two-dimensional complex-valued function.

First of all, let us define the function used for the example. We will use a simple function that allows us to verify the result of the integrator:

$$f(x) = \frac{\log \alpha x}{\sqrt{x}}. \tag{1}$$

The integration routine takes as a parameter the address of a structure that contains the address of a global function that we would like to integrate. For that reason the signature of the function to integrate has a predefined shape:

Listing 1. Definition of the one-dimensional integrand

```
double func(double x, void* params)
{
    double alpha = *(double*)params;

    return log(alpha * x) / sqrt(x);
}
```

The function in Listing 1 must return a `double` value, and must have two parameters: a variable of type `double` that is the argument of the function x, and a `void` pointer to store the address of the parameter, in the case of (1) is $\alpha = 1$.

Then in a program scope after the definition of the integrand, we must set the needed parameters for the integrator. For this basic example it can be done in the `main()`.

We prepare the structure that represents the function to be integrated:

Listing 2. Definition of the GSL function structure

```
double alpha = 1;                   // parameter variable
gsl_function intgr_func;            // GSL function
intgr_func.function = &func;        // address of the function
intgr_func.params = &alpha;         // parameters
```

Allocate memory for the integration algorithm workspace that will contain the sub-interval ranges.

```
gsl_integration_workspace* wrkspc =
        gsl_integration_workspace_alloc(WRKSPC_SIZE);
```

And after that, invoke the integration algorithm using the GSL routine:

Listing 3. Invoke the GSL integration routine

```
gsl_integration_qags(
        &intgr_func,        // function structure
```

```
        0,                  // left boundary of the interval
        1,                  // right boundary of the interval
        0,                  // absolute error limit
        ACC,                // relative error limit
        WRKSPC_SIZE,        // workspace size
        wrkspc,             // workspace
        &result,            // estimated result
        &error              // estimated error
    );
```

where `intgr_func` is the function structure to be integrated, the integration is over $[0, 1]$, absolute error limit is 0, `ACC` is a constant that defines the relative error limit, `WRKSPC_SIZE` is a constant the defines the workspace size, `wrkspc` is a pointer to the workspace, `result` will contain the estimated quadrature, and `error` will contain the estimated error.

Finally, we must free the allocated memory:

```
gsl_integration_workspace_free(wrkspc);
```

Running this example will give us the following result:

```
Result: -4.00000000000008527
Exact result: -4
Estimated error: 1.35447209004269098e-13
Actual error: -8.52651282912120223e-14
Intervals: 8
```

which as expected for this input function shows a very good accuracy of the GSL implementation of the integration algorithm. The challenging task in our case is to find a way to apply the routine to functions of type $g(x, y)$, where $x, y \in \mathbb{R}$, and $g \in \mathbb{C}$.

3. Extension to Two-dimensional Functions. Since GSL provides integration of one-dimensional functions only, we have to find a way to apply these procedures to two-dimensional functions, calculating the double integral. Here we use the idea expressed by Fubini's theorem that allows us to represent a double integral as an iterated integral (see [4, 919]).

Theorem 1. *Let $f(x, y)$ is a continuous function on the rectangular region R : $a \leq x \leq b, c \leq y \leq d$. Then the following equality holds:*

$$\iint_R f(x,y)\,d(x,y) = \int_a^b \int_c^d f(x,y)\,dy\,dx. \tag{2}$$

To implement the iterated integral, first we have to present the function $f(x,y)$ in the form that can be accepted by the GSL routines. The simple function that we will use for the purpose of the example this time is:

$$f(x,y) = x^k y, \tag{3}$$

where k is a given constant.

Listing 4. Two-dimensional function (3) that has to be integrated

```
double func(double x, void* params)
{
    double y = ((double*)params)[0];
    double k = ((double*)params)[1];

    return pow(x, k) * y;
}
```

Note how the pointer `params` is used to store both the second variable y and the parameter k. The signature of the function has the same shape, as the signature of the one-dimensional function in Listing 1.

The function that represents the outer integral is a function of y, and for each particular value of y the integration on x is performed:

$$\int \left(\int f(x,y)\,dx \right) dy. \tag{4}$$

The implementation of (4) again is a function with a signature in the format, that can be passed to the GSL integration routines:

Listing 5. Outer integrand has the predefined shape of the function signature

```
double outIntegr(double y, void* params)
{
    $\ldots$
    return result;
}
```

To perform the integration of $f(x, y)$ with fixed y, and x as an integration variable, y has to be set as a parameter of the inner integrand. Then the GSL function structure is built, and the integrator is invoked, just like in the case of Listing 3.

```
((double*) params)[0] = y;

// memory for the sub-interval ranges,
// results and error estimates
gsl_integration_workspace* wrkspc =
    gsl_integration_workspace_alloc(WRKSPC_SIZE);

// variables to pass to integration function
double result = 0;              // estimated result
double error = 0;               // estimated error

// function to integrate
gsl_function intgr_func;        // GSL function
intgr_func.function = &func;    // address of the function
intgr_func.params = params;     // parameters

// integration algorithm: inner integrand
gsl_integration_qags($\ldots$);
```

As final step in the function `outIntegr` (Listing 5), we must free the allocated memory for the workspace, and return the result:

```
gsl_integration_workspace_free(wrkspc);
return result;
```

Now the outer integrand function must be integrated by itself in a separate function, lets say, `integrate()`. This time there is no restriction to the signature of the function, because this function itself is not going to be an input of an integration routine.

First the parameter k must be set as a second element of the parameter array:

```
const int PRM_NUMB = 2;
double params[PRM_NUMB];
params[1] = k;
```

Again, the memory for the workspace must be allocated:

```
gsl_integration_workspace* wrkspc =
    gsl_integration_workspace_alloc(WRKSPC_SIZE);
```

Build the GSL function structure:

```
gsl_function intgr_func;              // GSL function
intgr_func.function = &outIntegr;     // address of the function
intgr_func.params = params;           // parameters
```

And finally, call the integration routine, free workspace memory, and return the result of the iterated integral:

```
gsl_integration_qags($\ldots$);
gsl_integration_workspace_free(wrkspc);
return result;
```

The call of the function `gsl_integration_qags()` is similar, as in the case of Listing 3.

The value of integral calculated in this basic example is 0.0116279, and as a simple verification shows, this is the expected valued for the example input function over the interval $[0, 1]$.

It is important to note that the functions that can be passed as an input of the GSL integration routines must be global functions, or if they are class member functions, they must be static. Unfortunately, this restriction makes almost pointless the effort to pass a member function to the integrator. Also, the memory that is used for the integration workspace is allocated once for the integration of the outer integrand, and then it is allocated and deleted each time the inner integrand is passed to the integrator. If many integral values must be calculated, this process may impede significantly the execution of the program.

4. Extension to Two-dimensional Complex Functions. The final stage is to extend the idea from the previous section for the specific case of complex-valued functions of real-valued arguments:

$$g(x, y), \text{ where } x, y \in \mathbb{R}, \text{ and } g \in \mathbb{C}. \tag{5}$$

First of all, we will go around the problem of workspace allocation and de-allocation, by setting the pointer to the integration workspace as a global variable. Also, we will keep the address of the complex-valued function that has

to be integrated in a global pointer, that will allow us to keep the signatures of the functions in the format that is required by the GSL procedures (see the signature of the function in Listing 1).

Listing 6. Global variables, enclosed in a workspace, preserve the shape of the functions signatures and prevent multiple memory allocation

```
namespace integr
{
    complex<double> (* ptr_func_cmlx_glb)
        (double, double) = nullptr; // complex function
    double left_lmt_x_glb = 0.0;    // lower limit of x
    double rght_lmt_x_glb = 0.0;    // upper limit of x
    int wrksp_size_glb = 0;         // workspace size
    gsl_integration_workspace*      // integration workspace
        ptr_wrkspc_glb = nullptr;
}
```

We enclose the needed global variables in a name space that introduces a certain level of naming security. Using global pointers and variables, memory for the workspace can be allocated only once in the client program before the whole integration process, and then de-allocated only once when integration is completed. Thus, the number of memory management operations is considerably reduced.

The integration of the complex-valued function is initiated in a function `intgrt2dc()` that has the following form:

Listing 7. Integrated a complex valued function of two real arguments

```
complex<double>                     // resulting complex number
    intgrt2dc(
        complex<double> (* ptr_func)
            (double, double),       // function to integrate
        double left_lmt_x,          // lower limit of x
        double rght_lmt_x,          // upper limit of x
        double left_lmt_y,          // lower limit of y
        double rght_lmt_y           // upper limit of y
        )
{
    // initialize global variables
    integr::ptr_func_cmlx_glb = ptr_func;
    integr::left_lmt_x_glb = left_lmt_x;
    integr::rght_lmt_x_glb = rght_lmt_x;
```

```
    complex<double> result;

    // integrate real part of the function
    result.real(itrIntgrndReal(left_lmt_y, rght_lmt_y));

    // integrate imaginary part of the function
    result.imag(itrIntgrndImag(left_lmt_y, rght_lmt_y));

    // set global variables to zero
    integr::ptr_func_cmlx_glb = nullptr;
    integr::left_lmt_x_glb = 0;
    integr::rght_lmt_x_glb = 0;
    integr::wrksp_size_glb = 0;

    return result;
}
```

The real and imaginary parts of the resulting complex number in Listing 7 are integrated separately, the complex number is constructed, and then returned as value of the function integral. The real and imaginary parts are integrated analogously, that is why we will describe the real part calculation.

The iterated integral for the real part of the function is calculated in the procedure `itrIntgrndReal()`, following the example from the previous section, in which the GSL integration routine is triggered:

```
double result = integrGSL(
    &outIntgrndReal,            // outer integrand
    &par,                       // store parameter
    left_lmt_y,                 // lower limit of y
    rght_lmt_y,                 // upper limit of y
    integr::ptr_wrkspc_glb,     // integration workspace
    integr::wrksp_size_glb,     // workspace size
    GSLPRC_ACC                  // accuracy bound
);
```

The outer integrand that is iterated over is a function `outIntgrndReal` that triggers the GSL integration routine by passing the address of a function `funcReal()`. The role of the latter one is to calculate the real part of the input complex-valued function using the global pointer `ptr_func_cmlx_glb`:

```
double y = *((double*)params);
return integr::ptr_func_cmlx_glb(x, y).real();
```

In this way the real part of the function $g(x, y)$ is integrated as an iterated integral, but also the workspace is going to be allocated only once for the whole set of integrals that have to be calculated by the client program. The integration of the imaginary part of the function is performed in the same way.

5. Conclusions. The presented extension of the GSL routines for the case of the complex-valued two-dimensional functions is part of our project that implements BIEM software system for for solving a 3D elastodynamics problem for wave propagation in a continuously inhomogeneous half-space. We have tested the approach with calculation of integrals of a fundamental solution of the method, where each separate integral is calculated for a 3×3 functor of complex-valued functions. The integrals that are calculated per experiment are at least 3528 in the minimal case, and the time consumed by the method is around 7 minutes on standard PC with Intel Core i3-3220 CPU 3.30GHz $\times$ 4, running on a Linux distribution.

We must note that because of the characteristics of the fundamental solution, in some cases it may be considered a "badly-behaved" integrand and it contains singularities that cannot be processed by the regular integration algorithms, even by the QAGS adaptive integration GSL implementation. In these cases we adopt a Monte Carlo method for integration. The reason why we do not process all integrands using the Monte Carlo method is that the execution time of the program rises dramatically, and also the traditional worse accuracy of those methods.

A future improvement of the presented approach is to ensure that the integration procedures are thread-safe, and to adopt parallel computing of the separate independent integrals.

REFERENCES

[1] Rice J. R. A Metalgorithm for Adaptive Quadrature. *Journal of the ACM*, **22** (1975), No 1, 61–82.

[2] Piessens R., E. de Doncker-Kapenga, C. W. Überhuber, D. K. Kahaner. Quadpack: A Subroutine Package for Automatic Integration. *Springer Series in Computational Mathematics*, **1** (1983).

[3] Zwillinger D. The Handbook of Integration. A K Peters/CRC Press, 1992.

[4] Thomas G. B., R. L. Finney. Calculus and Analytic Geometry. 8th ed., Reading (MA), Addison-Wesley, 1996.

[5] Frigo M., S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, **93** (2005), No 2, 216–231.

[6] Press W. H., S. A. Teukolsky, W. T. Vetterling, B. P. Flannery. Numerical Recipes: The Art of Scientific Computing. 3rd ed., Cambridge University Press, 2007.

[7] Gough B. GNU Scientific Library Reference Manual. 3rd ed., Network Theory Ltd., 2009.

[8] Burden R. L., J. D. Faires. Numerical Analysis. Brooks/Cole, Cengage Learning, 2011.

[9] Marinov M., Ts. Rangelov. Integro-differential Equations for Anti-plane Cracks in Inhomogeneous Piezoelectric Plane. *Comptes rendus de l'Academie bulgare des Sciences*, **64** (2011), No 12, 1669–1679.

[10] Quarteroni A., F. Saleri, P. Gervasio. Scientific Computing with MATLAB and Octave. *Texts in Computational Science and Engineering*, **2** (2014).

[11] Constanda C., D. Doty, W. Hamill. Boundary Integral Equation Methods and Numerical Solutions. *Developments in Mathematics*, **35** (2016).

Lasko M. Laskov
Informatics Department
New Bulgarian University
21, Montevideo Blvd
1618 Sofia, Bulgaria
e-mail: `llaskov@nbu.bg`

Received September 17, 2020
Final Accepted October 18, 2020