A User Manual for Cubpack++
Version 1.1

Ronald Cools, Dirk Laurie and Luc Pluym

Report TW255, March 1997
A User Manual for Cubpack++
Version 1.1

Ronald Cools, Dirk Laurie* and Luc Pluym

Report TW 255, March 1997

Department of Computer Science, K.U.Leuven

Abstract

Cubpack++ is a large C++ class library dealing with automatic integration of functions over two-dimensional regions. Here we will discuss its user interface, a small collection of classes that make up a kind of language for describing integration problems. This User Manual is one of two mutually complementary documents in which the package is described: the other is the paper

Algorithm 764: Cubpack++ – A C++ package for automatic two-dimensional cubature.

which contains the mathematical background and algorithmic details.

CR Subject Classification: G.1.4 [Numerical Analysis]: Quadrature and Numerical Differentiation – adaptive quadrature; multiple quadrature; G.4 [Mathematical Software]: efficiency; reliability and robustness

*Dept. of Mathematics, Potchefstroomse Universiteit vir Christelike Hoër Onderrig, P.O. Box 1174, Vanderbijlpark 1900, South Africa
RONALD COOLS
Dept. of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: ronald.cools@cs.kuleuven.ac.be

DIRK LAURIE
Dept. of Mathematics, Potchefstroomse Universiteit vir Christelike Hoër Onderwys
P.O. Box 1174, Vanderbijlpark 1900, South Africa
e-mail: wskdpl@pukrs12.puk.ac.za

LUC PLUYM
Dept. of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
(Current address: the Shaanxi Institute for Finance and Economics, Xi’an Cuihua Lu 105,
ZIP 710061, P.R.China.)
e-mail: lucp@public.xa.sn.cn

Copyright ©1994, 1996, 1997

Registered system holders may reproduce all or any part of this publication for internal purposes, provided that the source of the material is clearly acknowledged, and the title page and copyright notice are retained.

Note that the Belgian Software-law of June 20, 1994, which implements the corresponding European regulation of May 14, 1991, is applicable.
1 Introduction

Cubpack++ is a large C++ class library dealing with automatic integration of functions over two-dimensional regions. Here we will discuss its user interface, a small collection of classes that make up a kind of language for describing integration problems. This User Manual is one of two complementary documents in which the package is described: the other is the paper [2] which contains the mathematical background and algorithmic details. Full information also appears in the include files of the source code, which are elaborately commented.

There will probably never be a final version of this User Manual. It should provide answers to all Frequently Asked Questions, so if you encounter any problems when using our software, please let us know.

Our intention is not to present a final solution to the problem of automatic two-dimensional cubature, but to provide a framework on which others may build. We will welcome any attempt to extend the package by adding your own classes or functions.

You are assumed to be familiar with the basics of the language C++, but not to be an expert on it. In fact, most of the time you won't need any C++ feature more advanced than knowing how to print out your results.

Let's start with an example.

2 An example

Suppose we would like to integrate $x^2$ over the triangle with vertices (0,0), (1,1) and (2,0). Using Cubpack++ we proceed as follows (see file Examples/vb1.c):

```c
#include <cubpack.h>
#include <iostream.h>

real f(const Point& p)
{
    real x=p.X();
    return x*x;
}

main()
{
    Point p1(0,0), p2(1,1), p3(2,0);
    TRIANGLE T(p1,p2,p3);
    cout << "The integral is " << Integrate(f,T) << endl;
}
```

As you see, it is quite easy to describe regions and to integrate over them. We now give a more systematic discussion of Cubpack++'s class library.
3 Basic classes

The following simple classes are built into Cubpack++.

real All our floating point numbers are instances of this type. We do this instead of using a predefined floating point type like float or double in order to make it as easy as possible for you to change the precision: by default real is a typedef to a double, but you can specify single precision (i.e. float) simply by making sure that the precompiler variable FLOAT is defined during compilation. In the unusual case when your compiler supports another floating point format, you may as a last resort edit the file real.h.

Point Two-dimensional points are thought of as entities in Cubpack++ except when it is essential to extract the coordinates, which you do as in the example: the coordinates of the point \( P \) are \( P.X() \) and \( P.Y() \). Addition and similar operations on Points are all defined.

Function You will normally wish to integrate a real functions of a Point. To save us all the trouble of writing \( \text{real } (*)(\text{const Point&}) \) every time we need to refer to such a function, the class Function has been defined. A function \( \text{real } f(\text{const Point& p}) \) as used in the example is a valid argument to any routine that requires a Function.

Boolean During many years, C++ users had to define their own Boolean type, and there was some controversy as to the best way of defining them. We have opted for the simplest solution: our Boolean is an enum \{False, True\}. Recently the ANSI/ISO C++ standards committee added a new feature to the language: a built-in boolean type 'bool', with constants 'true' and 'false' (this will break code that uses these keywords as variable names). This new feature was implemented in version 2.6.0 of gcc/g++. However, because it will take some time before all available compilers implement this new feature, we continue to use our own class.
4 Describing simple regions

A simple region is roughly defined as a region that can be specified using a small number of parameters, and more precisely as a region that belongs to the list below. Like Points, simple regions can be created using constructors. We have tried as far as possible to use Points as the arguments to the constructors; in some cases involving regions bounded by circular arcs alternatives in terms of radii and angles have also been supplied. A region may have more than one constructor: C++ can recognise by the types of the actual arguments which one you mean.

In the constructor headers below, A, B, C, D, Center, BoundaryPoint and P are Points. When a constructor requires more than one Point, the Points should be distinct, except in the special cases specified below. Radius, InnerRadius, OuterRadius, ScaleX, ScaleY, Alpha and Beta are reals. Height is a Function, and Rho is a standard C function that takes one real argument and returns a real result.

Here is a list of the constructors for simple regions in Cubpack++.

\begin{verbatim}
TRIANGLE( A, B, C);
PARALLELOGRAM( A, B, C);
RECTANGLE( A, B, C);
CIRCLE( Center, BoundaryPoint);
CIRCLE( Center, Radius);
OUT_CIRCLE( Center, BoundaryPoint);
OUT_CIRCLE( Center, Radius);
PLANE;
PLANE( Center);
PLANE( Center, ScaleX, ScaleY);
GENERALIZED_RECTANGLE( Height, A, B);
POLAR_RECTANGLE ( Center, InnerRadius, OuterRadius, Alpha, Beta );
POLAR_RECTANGLE( A, B, D);
GENERALIZED_SECTOR( Rho, Alpha, Beta, Center);
PARABOLIC_SEGMENT( A, B, P);
INFINITE_STRIP( A, B);
SEMI_INFINITE_STRIP( A, B);
PLANE_SECTOR( Center, InnerRadius, Alpha, Beta);
PLANE_SECTOR( A, B, D);
\end{verbatim}

Here are more detailed descriptions of the simple regions. They are shown in Figures 1 and 2. In Figure 1 the region is the internal of the solid line. In Figure 2 the region is that part of the plane marked with X and bounded by the solid lines where lines with an arrow have to be extended to infinity.

TRIANGLE A, B and C are the vertices.

PARALLELOGRAM A, B and C are vertices such that BC is a diagonal of the parallelogram.

RECTANGLE is a special case: if a region is specified as a RECTANGLE, the software will check that angle BAC is a right angle.

CIRCLE The Center is specified, and either a BoundaryPoint or the Radius. The integration region is the enclosed disk.
<table>
<thead>
<tr>
<th>Triangle</th>
<th>Parallelogram</th>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Triangle" /></td>
<td><img src="image" alt="Parallelogram" /></td>
<td><img src="image" alt="Rectangle" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circle</th>
<th>Circle</th>
<th>Generalized rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Circle" /></td>
<td><img src="image" alt="Circle" /></td>
<td><img src="image" alt="Generalized rectangle" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Polar rectangle</th>
<th>Parabolic segment</th>
<th>Generalized sector</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Polar rectangle" /></td>
<td><img src="image" alt="Parabolic segment" /></td>
<td><img src="image" alt="Generalized sector" /></td>
</tr>
</tbody>
</table>

Figure 1: Finite regions available in Cubpack++. 
A User Manual for Cubpack++

Figure 2: Infinite regions available in Cubpack++.

OUT_CIRCLE The same as CIRCLE, except that the integration region is the complement of the disk.

PLANE The integration region is the whole two-dimensional plane. Strictly speaking, no parameters are necessary, but the integration routines use formulas with approximately half their points inside an ellipse centred at Center and having axes ScaleX and ScaleY. If you know where in the plane the action takes place, you can save some function evaluations by supplying these. Otherwise the value given for the ellipse defaults to the unit circle.

At this point, some users might find it useful to know more details. The cubature formula that is initially applied for this region can be transformed by an affine transformation to give an approximation to

\[ \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{(a(x-c_x))^2 + (b(y-c_y))^2} f(x, y) \, dx \, dy \]

while the cubature formula maintains its degree. We use this with \( a = \text{ScaleX}, b = \text{ScaleY} \) and \( c_x = \text{Center.X}, c_y = \text{Center.Y} \). The fact that Center, ScaleX and ScaleY are related to an ellipse with approximately the same number of points inside as outside, is used to determine a subdivision of the plane. For details of this algorithm, see Section 5.4 of the paper [2].

GENERALIZED_RECTANGLE This region has three straight sides that meet at right angles in the points A and B, and one curved side. The perpendicular distance between a point P on the directed line segment AB and the point on the curve orthogonally to the left of P is \( \text{Height}(P) \). It is necessary that \( \text{Height}(P) \geq 0 \) for all P between A and B.
POLAR_RECTANGLE This region is bounded by the rays $\theta = \alpha$ and $\theta = \beta$ (where $\beta > \alpha$) and the arcs $r = \text{InnerRadius}$ and $r = \text{OuterRadius}$ (where $\text{OuterRadius} > \text{InnerRadius}$) in polar coordinates with origin Center. An annulus ($\beta = \alpha + 2\pi$) is a special case.

Alternatively, you may specify the region in terms of Points: A and B are the points on the ray $\theta = \alpha$ at InnerRadius and OuterRadius, respectively, and D is the point on the ray $\theta = \beta$ at OuterRadius. You need not specify Center: the other points determine it fully, except when $AB \perp BD$. In that case, the region should be specified as a RECTANGLE and not as a limiting case of a POLAR_RECTANGLE. You cannot specify an annulus in terms of Points. Special cases are a sector ($\text{Center} = A$) and a cut circle ($\text{Center} = A, D = B$). From a topological point of view, points on either side of the line $AB$ in a cut circle are not thought of as close to each other. In §10.3 we shall later give an example that illustrates the difference between a circle and a cut circle.

GENERALIZED_SECTOR This region has two straight sides and one curved side. It is described in polar coordinates measured from Center. The straight sides run along rays at angles $\text{Alpha}$ and $\text{Beta}$ respectively, and the curved side is defined by the function $\text{Rho}$ as follows: the distance from Center to the curve along a ray at an angle $\theta$ between $\text{Alpha}$ and $\text{Beta}$ is given by $\text{Rho}(\theta)$.

PARABOLIC_SEGMENT This region is bounded by a straight line $AB$ and a parabolic arc. The arc is defined by an external point $P$ such that $AP$ and $BP$ are tangents.

INFINITE_STRIP The line segment $AB$ is a diameter of the strip.

SEMI-INFINITE_STRIP The strip lies orthogonally to the left of the directed line segment $AB$.

PLANE_SECTOR This region is bounded by the rays $\theta = \alpha$ and $\theta = \beta$ (where $\beta > \alpha$) and the arc $r = \text{InnerRadius}$ in polar coordinates with origin Center. Alternatively, you may specify the region in terms of Points: A is the point on the ray at $\theta = \alpha$ at $r = \text{InnerRadius}$, and B and D are points on an arc $r = \text{OuterRadius}$ (with $\text{OuterRadius} > \text{InnerRadius}$) at $\theta = \alpha$ and $\theta = \beta$ respectively. You need not specify Center: the other points determine it fully, except when $AB \perp BD$. In that case, the region should be specified as a SEMI-INFINITE_STRIP and not as a limiting case of a PLANE_SECTOR. Special cases include the half-plane ($D = 2A - B$) and cut plane ($\text{Center} = A, D = B$).

Small example programs that illustrate the use of these simple regions are provided in the directory Examples. We refer to the file Examples CONTENTS for its contents.

As you see all names of simple regions contain upper case letters only, the components of compound words being separated by underscores. All classes representing simple regions are derived from a class COMPOUNDED_REGION, which also comprises the transformed and subdivided regions that arise during the computation.
5 The integrators

Cubpack++ uses only one global adaptive integration algorithm, but to meet differing user requirements we have supplied four versions of the central routine Integrate. These will be discussed in this and the following section.

5.1 No-fuss integration

When you just want to know the integral of a Function f over a COMPOUND_REGION CR, you can use the function Integrate(f, CR) as in the example in Section 2. The value returned will then be an approximation of the integral.

This simple version of the integrator uses default values for the accuracy to which the integral should be evaluated and the amount of work permitted. If you want explicit control over the integration process, you can supply these values: the full header with its default arguments is:

```c
real Integrate(
    Function f,
    COMPOUND_REGION& CR,
    real AbsoluteErrorRequest = 0.0,
    real RelativeErrorRequest = DEFAULT_REL_ERR_REQ ,
    unsigned long MaxEval = 100000);
```

The constant DEFAULT_REL_ERR_REQ is defined in the file real.h. As usual with C++ you may omit any number of trailing arguments for which defaults are given.

The integrator tries to compute an approximation to the integral of f on CR which satisfies

\[ |\text{AbsoluteError}| \leq \text{AbsoluteErrorRequest} \]

or

\[ |\text{AbsoluteError}| \leq |\text{Integral}| \times \text{RelativeErrorRequest}, \]

whichever it reaches first. The result is returned via Integral and AbsoluteError. However it will at most do MaxEval function evaluations.

5.2 Advanced integration

If the automatic integrator cannot satisfy one of the error requests, the flag Success will be set to False; otherwise it will be set to True. The flag Success cannot be seen when you use the simple integrator whose only output is a function value. If you wish to see it, or to store the integral and error estimates in variables of your own choice, the header of Integrate is:

```c
void Integrate(
    Function f,
    COMPOUND_REGION& CR,
    real& Integral,
    real& AbsoluteError,
    Boolean& Success,
    real AbsoluteErrorRequest,
    real RelativeErrorRequest,
    unsigned long MaxEval);
```
Note that this version requires you to specify all the arguments: no default values are given. The three reference parameters are used to return computed quantities.

Two further versions of \texttt{Integrate}, intended for more complicated situations, are described in the next section.

For all four versions of \texttt{Integrate}, the integral and error estimate are stored with the region \texttt{CR} itself, and can be retrieved at any stage after integration by the real functions \texttt{CRIntegral()} and \texttt{CRAbsoluteError()} respectively.

6 Handling collections of regions

Sometimes you want to integrate over a collection of regions, rather than over one of the simple regions mentioned above. Suppose for instance that you would like to integrate over the house-shaped region shown in Figure 3. In \texttt{Cubpack++} you would use a generalization of a \texttt{COMPOUND_REGION}, called a \texttt{REGION_COLLECTION}. It goes like this:

\begin{verbatim}
REGION_COLLECTION House;
TRIANGLE Roof( Point(0,1), Point(2,1), Point(1,2));
RECTANGLE Walls(Point(0,0), Point(0,1), Point(2,0));
House = Roof + Walls;
\end{verbatim}

\begin{verbatim}
cout << "integral over the house-shaped region " << Integrate(f,House);
\end{verbatim}

You can add as many \texttt{COMPOUND_REGION}s as you like, giving a \texttt{REGION_COLLECTION} (which is itself a \texttt{COMPOUND_REGION}). The addition can either use the + operator as above, or the += operator as in C.

You needn't use \texttt{REGION_COLLECTION} explicitly, you can also do it like this:

\begin{verbatim}
... cout << Integrate(f,Roof+Walls);
\end{verbatim}

Figure 3: A house

After calling \texttt{Integrate} you can access the integral and error on each part of a collection, i.e. you know \texttt{Roof.InTEGRAL()} as well as \texttt{Walls.InTEGRAL()} et cetera. This is made possible by the implementation of the addition operators for \texttt{COMPOUND_REGION}s. Rather than adding a copy of the original \texttt{COMPOUND_REGION} to the \texttt{REGION_COLLECTION} a reference to its contents is stored.

You are allowed to integrate different functions over different parts of a \texttt{REGION_COLLECTION}, which may for instance be useful when your integrand is discontinuous. In that case, you use a version of \texttt{Integrate} that omits the first argument (i.e. the integrand), and instead associate an integrand with each region, like this:
Roof.LocalIntegrand(f1);
Walls.LocalIntegrand(f2);
cout << "the result is " << Integrate(Roof+Walls);

The full header for this integrator is:

```cpp
real Integrate(
    COMPOUND_REGION& CR,
    real AbsoluteErrorRequest = 0.0,
    real RelativeErrorRequest = DEFAULT_REL_ERR_REQ,
    unsigned long MaxEval = 100000);
```

A complete example is given in the file Examples/vb15.c.

If you forgot to specify the integrand using LocalIntegrand() for each region in your REGION_COLLECTION before calling Integrate(), Cubpack++ will generate an error message.

Similarly, there exists a version of the more elaborate Integrate() without the Function argument. The full header is:

```cpp
void Integrate(
    COMPOUND_REGION& CR,
    real& Integral,
    real& AbsoluteError,
    Boolean& Success,
    real AbsoluteErrorRequest,
    real RelativeErrorRequest,
    unsigned long MaxEval);
```

Although the four versions of Integrate have the same name, C++ deduces which one is required by its actual arguments.
7 A possible pitfall

One feature of the automatic integrator requires further mention. If you e.g. create a triangular region by saying \texttt{TRIANGLE T1(A,B,C)}, then the package will construct an object \texttt{T1} that contains a data-store to be used to store all regions created when one integrates over the triangle. Apart from this data-store, the object \texttt{T1} also contains a number of attributes (the total approximate integral and error estimate, requested absolute and relative tolerances, and a count of the number of functions evaluations) all with respect to the triangle mentioned. After integration, the original region is no longer available, but it will have been replaced by a \texttt{COMPOUND_REGION} consisting of all the transformations and subdivisions that the automatic integrator made.

The advantage of this feature is that you can now continue integrating (i.e. without wasting previous computations) if you require higher precision: for example, if the behaviour of your integrand is unfamiliar to you, you will be able to assess the reliability of the computed integrals by printing them out for requested tolerances of $10^{-2}, 10^{-3}, \ldots$ until you are satisfied with the result. Details on how this can be done are given in \S8.

We are aware that some users might find this a bit difficult to live with in the beginning. They should remember that \texttt{Cubpack++} is a package to compute integrals and not a package to manipulate geometric objects. They should keep in mind that a \texttt{TRIANGLE} is much more than just the geometric object. Compared to existing Fortran 77 codes, an object \texttt{TRIANGLE} replaces at least 6 parameters in the calling sequence, including a real and an integer workarray Fortran 77 users have to provide.

As a consequence of this side-effect a sequence of code like

\begin{verbatim}
TRIANGLE T1(A,B,C);
Integrate (f,T1);
Integrate (g,T1);
\end{verbatim}

will produce an error (Attempt to modify integrand during integration).

If you need a copy of the original region after integration (for example, when you wish to integrate another function over the same region), you must construct a separate instance. It is not sufficient to say

\begin{verbatim}
TRIANGLE T1(A,B,C), T2=T1;
\end{verbatim}

because class \texttt{COMPOUND_REGION} uses reference semantics everywhere. \texttt{T2} will not be another triangle with the same specifications as \texttt{T1} but another reference to the same triangle. After calling \texttt{Integrate(cf,T1)} you will find that \texttt{T1.Integral()} and \texttt{T2.Integral()} return the same value. If either of the two objects is destructed or passes out of scope, you can still reach its contents via the other one.

You must therefore say

\begin{verbatim}
TRIANGLE T1(A,B,C), T2(A,B,C);
\end{verbatim}

to force the creation of two independent copies.

A complete example is given in the file \texttt{Examples/vb9.c}.
8 Tightening the tolerance

It can happen that you want to compute the integral of a certain function over a certain region several times. E.g. with increasing requested accuracies or with increasing allowed number of function evaluations. Cubpack++ offers several ways to achieve this.

1. Repeated calls of

\[ \text{Integrate(Function } f, \text{ COMPOUND\_REGION} & \text{ CR, ...);} \]

are allowed if \( f \) and \( \text{CR} \) are not changed by \( you \) between successive calls. (Remember that \( \text{CR} \) is changed by \text{Integrate}!)

2. Repeated calls of

\[ \text{Integrate(COMPOUND\_REGION} & \text{ CR, ...);} \]

are allowed if before the first call (and only then) a

\[ \text{CR.LocalIntegrand}(f); \]

is executed and if \( \text{CR} \) is not changed by \( you \).

3. After a call of

\[ \text{Integrate(Function } f, \text{ COMPOUND\_REGION} & \text{ CR, ...);} \]

repeated calls of

\[ \text{Integrate(COMPOUND\_REGION} & \text{ CR, ...);} \]

are allowed if \( \text{CR} \) is not changed by \( you \).

Although we cannot stop you, you should never change the integrand \( f \) between successive calls by changing global variables that are used by \( f \). Remember that after one call of \text{Integrate} the original region is replaced by a \text{COMPOUND\_REGION} consisting of all the transformations and subdivisions that the automatic integrator made. Cubpack++ detects if you call \text{Integrate} with such a \text{COMPOUND\_REGION} with a different function than you used the first time but it cannot detect situations where you modify the function between calls. Complete examples are given in the files Examples/vb14.c and Examples/vb17.c.
9 Measuring the performance

If you want to know how much work Cubpack++ has done to evaluate your integral, you should define a variable (say Count) of the class EvaluationCounter. This variable can be thought of as a stop-watch: after you have said Count.Start(), a counter is incremented whenever a function evaluation has been performed. To read the counter, you call Count.Read(), and to stop it, you say Count.Stop(). You can reset it to a value n by Count.Reset(n) or to zero by Count.Reset(). Remember to start the counter again after resetting it! See the example in §10.1.

On some systems it is also possible to measure CPU or actual elapsed time. We have provided a class Chrono, but the functions in that class normally do nothing. You should replace them by your own system-dependent functions. As an example, we did this for UNIX systems that have a getrusage() system call or a times() system call. On systems with getrusage(), the relevant part of the code is activated when the symbol GETRUSAGE is defined to the C preprocessor (usually by compiling with the option -DGETRUSAGE). On systems with times(), the relevant part of the code is activated when the symbol TIMES is defined to the C preprocessor. In both cases, the elapsed time is measured in milliseconds. For details, read the file chrono.h and the example in the file Examples/vb4.c.

10 Examples

The Cubpack++ package has been run successfully on several C++ compilers (see also §11 and Table 1). The test results below were obtained by Gnu C++ 2.7.2 with libg++ 2.7.2. You can download the software by anonymous ftp from ftp.cs.kuleuven.ac.be where it is located in the directory pub/NumAnal-ApplMath/Cubpack or using a World Wide Web Browser from HREF="http://www.cs.kuleuven.ac.be/~ronald/".

Example main programs are in the directory Examples; see the file Example/CONTENTS for its contents. In the following sections we look at some of them in detail.

10.1 A plane sector

The function \( f(x,y) = \exp(-((x^2 + y^2)/2)) \) is to be integrated over a wedge-shaped region starting at the origin and spanning the polar range \( 0 \leq \theta \leq \arctan 2 \):

\[
\int_0^\infty \int_0^{2x} e^{-((x^2+y^2)/2)} \, dz \, dy.
\]

This is Example 33 from [3].

In this example, we use the advanced integrator in a loop with decreasing required error. We force the use of a relative error criterion by specifying the AbsoluteErrorRequest as zero. The exact C++ code is (see file Examples/vb3.c):
// Example 33 from ditamo
// Exact value = arctan(2)
// Note that the exact value mentioned in the paper describing ditamo
// is wrong.

#include <cubpack++.h>
#include <iostream.h>
#include <iomanip.h>

real f(const Point& p)
{
    real x=p.X(), y=p.Y();
    return exp(0.5*(-x*x -y*y));
}

int main()
{
    Point origin(0,0);
    real innerradius=0, alfa=0, beta=atan(2.0);
    PLANE_SECTOR wedge(origin,innerradius,alfa,beta);
    EvaluationCounter count;

    cout.setf(ios::scientific,ios::floatfield);
    cout<<"req. rel. error est integral est error abs error evaluations"<<endl;
    count.Start();
    for (real req_err=0.05; req_err>1e-12; req_err/=10)
    {
        cout << setprecision(1) << " " << req_err << " "
             << setprecision(10) << Integrate(f,wedge,0,req_err) << " ";
        cout << setprecision(1) << wedge.AbsoluteError() << " "
             << fabs(wedge.Integral() - atan(2.0)) << " "
             << count.Read() << endl;
    }
    count.Stop();

    return 0;
}
Note that two statements are used to send output to cout. This is done to make sure that the integral has been computed by the time we print it out, because the sequence in which the individual terms are evaluated in an expression containing several \(<\) operators, has been left undefined in C++.

The above program produces the following output (g++, Ultrix).

<table>
<thead>
<tr>
<th>req. rel. error</th>
<th>est integral</th>
<th>est error</th>
<th>abs error</th>
<th>evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;5.0e-02</td>
<td>1.1096504402e+00</td>
<td>3.4e-02</td>
<td>2.5e-03</td>
<td>292</td>
</tr>
<tr>
<td>&lt;5.0e-03</td>
<td>1.1071935478e+00</td>
<td>4.1e-03</td>
<td>4.5e-05</td>
<td>1181</td>
</tr>
<tr>
<td>&lt;5.0e-04</td>
<td>1.1071470438e+00</td>
<td>5.1e-04</td>
<td>1.7e-06</td>
<td>3105</td>
</tr>
<tr>
<td>&lt;5.0e-05</td>
<td>1.1071488562e+00</td>
<td>5.3e-05</td>
<td>1.4e-07</td>
<td>5657</td>
</tr>
<tr>
<td>&lt;5.0e-06</td>
<td>1.1071486939e+00</td>
<td>5.3e-06</td>
<td>2.4e-08</td>
<td>11239</td>
</tr>
<tr>
<td>&lt;5.0e-07</td>
<td>1.1071487183e+00</td>
<td>5.3e-07</td>
<td>5.5e-10</td>
<td>16377</td>
</tr>
<tr>
<td>&lt;5.0e-08</td>
<td>1.1071487178e+00</td>
<td>5.5e-08</td>
<td>1.0e-12</td>
<td>25553</td>
</tr>
<tr>
<td>&lt;5.0e-09</td>
<td>1.1071487178e+00</td>
<td>5.5e-09</td>
<td>1.1e-11</td>
<td>38577</td>
</tr>
<tr>
<td>&lt;5.0e-10</td>
<td>1.1071487178e+00</td>
<td>5.4e-10</td>
<td>5.2e-13</td>
<td>53081</td>
</tr>
<tr>
<td>&lt;5.0e-11</td>
<td>1.1071487178e+00</td>
<td>5.5e-11</td>
<td>2.6e-13</td>
<td>73949</td>
</tr>
<tr>
<td>&lt;5.0e-12</td>
<td>1.1071487178e+00</td>
<td>5.5e-12</td>
<td>1.6e-15</td>
<td>117794</td>
</tr>
</tbody>
</table>

10.2 When geometry should not rule

A referee asked us to integrate the following "reasonable integrand" over the House-shaped region shown in Figure 3:

\[
f(x,y) = \left[ \sqrt{(x-\pi^{0.5})^2 + (y-1)^2 + 0.00001} + 3(y-\pi^{0.35})^4 + (x-1)^2 + 0.00002 \right] \\
\times \sin[(x-y-\pi^{0.5} + \pi^{0.35})/((x+y-2)^2 + 0.01)]
\]

If one substitutes \(y = 1\), one obtains

\[
f(x,1) = \left[ \sqrt{(x-\pi^{0.5})^2 + 0.00001} + 3(1-\pi^{0.35})^4 + (x-1)^2 + 0.00002 \right] \\
\times \sin[(x-1-\pi^{0.5} + \pi^{0.35})/((x-1)^2 + 0.01)]
\]

and one immediately sees that this function has fast oscillations in the neighbourhood of \(x = 1\). A picture also reveals a discontinuity in the first derivative in the neighbourhood of \(x = 1.8\). It is obvious that oscillations occur in the neighbourhood of the line \(x + y - 2 = 0\) because there the denominator of the argument of the \(\sin\) becomes small.

A straightforward application of Cubpack++ is given in the following program:
```cpp
#include <cubpack.h>
#include <iostream.h>

#define sqr(x) ((x)*(x))

static const real sqrt_PI = sqrt(M_PI),
p35_PI = pow(M_PI, 0.35e0);

real f(const Point& p) { real x=p.X(), y=p.Y();
    return ( sqrt( sqr(x-sqrt_PI) + sqr(y-1) + 0.00001) +
            pow(pow(y-p35_PI, 4) + sqr(x-1) + 0.00002, 1.0/3.0) ) * sin( (x-y-sqrt_PI+p35_PI)/(sqrt(x+y-2) + 0.01) );
}

int main () {
    EvaluationCounter count;
    //----------------------------
    TRIANGLE Roof( Point(0,1), Point(2,1), Point(1,2));
    RECTANGLE Walls( Point(0,0), Point(0,1), Point(2,0));
    REGION_COLLECTION House = Walls + Roof;

    count.Start();
    cout <<"The integral is " << Integrate(f,House,0,0.5e-1,1000000);
    cout <<" with absolute error " << House.AbsoluteError() << endl;
    count.Stop();
    cout << count.Read() << " function evaluations were used." << endl;
    //----------------------------
    return 0;
    } //----------------------------
```

(This example is in file Examples/vb13.c.) A plot of the points where the function was evaluated is shown in Figure 4. The actual output of the above program is:

The integral is -0.40791 with absolute error 0.0203924
266548 function evaluations were used.

The geometry was here used to specify the region of integration. A logical choice but everyone knows that this is rarely the best one can do in numerical integration. A well known rule-of-thumb is

- **make sure the difficulty in the integrand is located in a vertex or on the edge of the region.**

We can give two additional rules:
- If the difficulty in the integrand is along a line that cannot be made an edge, make sure the line is parallel to an edge of the region that contains it.

- Cover as much as possible by rectangles (or parallelograms).

The above example violates the second rule and thus one pays. With little extra effort, a significant improvement is obtained. Replace the core of the above routine by

```c
//------------------
TRIANGLE T1(Point(0,0), Point(1,0), Point(0,1)),
    T2(Point(1,0), Point(2,0), Point(1.5,0.5)),
    T3(Point(2,0), Point(2,1), Point(1.5,0.5));
RECTANGLE R1(Point(1,0), Point(0,1), Point(2,1));
REGION_COLLECTION House = T1 + T2 + T3 + R1;
//------------------
```

and the following output is obtained:

The integral is -0.407393 with absolute error 0.0203507
105968 function evaluations were used.

The contributions of each of the user-defined subregions is given in the following table.

<table>
<thead>
<tr>
<th>Region</th>
<th>Integral</th>
<th>Absolute Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>-0.283031</td>
<td>0.00816722</td>
</tr>
<tr>
<td>T1</td>
<td>-0.221416</td>
<td>1.09496e-06</td>
</tr>
<tr>
<td>T2</td>
<td>0.068406</td>
<td>0.0055473</td>
</tr>
<tr>
<td>T3</td>
<td>0.0286485</td>
<td>0.00663513</td>
</tr>
</tbody>
</table>
Figure 5: House: integration-inspired subdivision with 1 rectangle

Figure 6: House: integration-inspired subdivision with 2 rectangles
A plot of the points where the function was evaluated is shown in Figure 5. Note that the area of the rectangles Walls and R1 are equal. If the triangles T2 and T3 are replaced by one rectangle and three triangles, the number of function evaluations is reduced by more than 10%. See Figure 6. For implementation details see the example in file Examples/vb16.c.)

10.3 When a circle should be a cut circle

We next give an example illustrating the distinction between a circle and a cut circle. This example also illustrates how Fortran can be called from C++ on some Unix systems.

The Hankel function of the first kind

\[ f(x, y) = |H_0^{(1)}(\omega z)| \]

where \( \omega \) is a number on the unit circle, and \( z = x + iy \), is to be integrated over the unit disk. The integral does of course not depend on the parameter \( \omega \), but the numerical behaviour of Cubpack++ is influenced by it.

This function (with \( \omega = 1 \)) is plotted on page 359 of [1]. It has a discontinuity along the ray \( \arg z = -\pi \). If we ask Integrate to integrate it by specifying the region as a CIRCLE, it achieves the requested tolerance \( 10^{-6} \) after 1441 function evaluations when \( \omega = 1 \) and after 6991 function evaluations when \( \omega = (4 + 3i)/5 \). If we specify the region as a cut circle, i.e. a POLAR_RECTANGLE with \( B = C = -\omega \), Cubpack++ uses 1887 function evaluations, independent of the choice of \( \omega \).

For this example, we computed the Hankel function with the aid of the Fortran routine ZBESH by D. E. Amos as supplied in the SLATEC library [4]. The C++ code used to call the Fortran function ZBESH is

```c
extern "C" {void zbesh_ (real&, real&, real&, int&, int&, int&,
       real[], real[], int&, int&);}
real AbsHankel ( const Point& z) {
    real x=z.X(), y=z.Y(), cr[10], ci[10], fmu=0;
    int kode=1, m=1, n=1, nz, ierr;
    zbesh_ (x,y,fmu,kode,m,n,cr,ci,nz,ierr);
    return sqrt(cr[0]*cr[0]+ci[0]*ci[0]);
}
```

The full C++ code for this example is in the file Examples/vb_hankel.c.

The difficulty when linking C++ and Fortran is that one must add a platform dependent archive. On a DECstation with Ultrix using f77 and the gnu C++ compiler, the option `-l Lots solves this problem. On a SUN Ultra with SunOS 5.5.1 (Solaris) using f90 and SUN's CC compiler, the option `-lf90 is required. Combining f90 and the gnu C++ compiler requires `-L/opt/SUNWsupro/SC4.0/lib` `-lf90.`
11 Installation guide

Together with the SPARCompiler C++4.0 comes a document “Migration Guide: C++ 3.0 to C++ 4.0 – Surviving with an Evolving Language”. The second part of the title is very illuminating. The C++ language is evolving and so are the compilers. Old bugs are very regularly replaced by new ones. At the moment the only way to find out whether a code is portable or not, is to try all compilers one can get. And so we did and we will continue to do this. This situation will remain as long as the ANSI-standard is not finalised and as long as the available compilers do not implement this standard.

For some compilers, our code is too complex. If we could work around this, we did. Sometimes, waiting for a new release was worthwhile. A list of compilers that compile the current version of Cubpack++ successfully is given in Table 1.

Table 1: Systems for which Cubpack++ has proven it works.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Version</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>g++/gcc</td>
<td>2.7.2</td>
<td>Ultrix 4.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SunOS 5.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux 1.2.4</td>
</tr>
<tr>
<td>g++/gcc</td>
<td>2.6.0</td>
<td>HP-UX</td>
</tr>
<tr>
<td>xlC</td>
<td></td>
<td>IBM AIX Version 3.2</td>
</tr>
<tr>
<td>cxx</td>
<td></td>
<td>DEC OSF/1 V1.3</td>
</tr>
<tr>
<td>CC</td>
<td>SPARCompiler C++ 4.0</td>
<td>Solaris 2.3 = SunOS 5.3</td>
</tr>
<tr>
<td>Turbo C++</td>
<td>3.00</td>
<td>MSDOS 5.0</td>
</tr>
</tbody>
</table>

The following files are system dependent:

chrono.c: For details we refer to §9.

real.h: For details we refer to §3.

templist.h: Used to define the variable TEMPLATEINCLUDE for those compilers that need inclusion of the template implementation.

The steps you have to do to get Cubpack++ at work for you, are described in the file INSTALL in the parent directory.
References


