CHAPTER

SEVENTEEN

NUMERICAL INTEGRATION

This chapter describes routines for performing numerical integration (quadrature) of a function in one dimension. There are routines for adaptive and non-adaptive integration of general functions, with specialised routines for specific cases. These include integration over infinite and semi-infinite ranges, singular integrals, including logarithmic singularities, computation of Cauchy principal values and oscillatory integrals. The library reimplements the algorithms used in QUADPACK, a numerical integration package written by Piessens, de Doncker-Kapenga, Ueberhuber and Kahaner. Fortran code for QUADPACK is available on Netlib. Also included are non-adaptive, fixed-order Gauss-Legendre integration routines with high precision coefficients, as well as fixed-order quadrature rules for a variety of weighting functions from IQPACK.

The functions described in this chapter are declared in the header file `gsl_integration.h`.

17.1 Introduction

Each algorithm computes an approximation to a definite integral of the form,

$$I = \int_a^b f(x)w(x)dx$$

where $w(x)$ is a weight function (for general integrands $w(x) = 1$). The user provides absolute and relative error bounds $(epsabs, epsrel)$ which specify the following accuracy requirement,

$$|RESULT - I| \leq \max(epsabs, epsrel|I|)$$

where $RESULT$ is the numerical approximation obtained by the algorithm. The algorithms attempt to estimate the absolute error $ABSERR = |RESULT - I|$ in such a way that the following inequality holds,

$$|RESULT - I| \leq ABSERR \leq \max(epsabs, epsrel|I|)$$

In short, the routines return the first approximation which has an absolute error smaller than $epsabs$ or a relative error smaller than $epsrel$.

Note that this is an *either-or* constraint, not simultaneous. To compute to a specified absolute error, set $epsrel$ to zero. To compute to a specified relative error, set $epsabs$ to zero. The routines will fail to converge if the error bounds are too stringent, but always return the best approximation obtained up to that stage.

The algorithms in QUADPACK use a naming convention based on the following letters:

```
Q - quadrature routine

N - non-adaptive integrator
A - adaptive integrator
```

(continues on next page)

(continued from previous page)

```
G - general integrand (user-defined)
W - weight function with integrand

S - singularities can be more readily integrated
P - points of special difficulty can be supplied
I - infinite range of integration
O - oscillatory weight function, cos or sin
F - Fourier integral
C - Cauchy principal value
```

The algorithms are built on pairs of quadrature rules, a higher order rule and a lower order rule. The higher order rule is used to compute the best approximation to an integral over a small range. The difference between the results of the higher order rule and the lower order rule gives an estimate of the error in the approximation.

17.1.1 Integrands without weight functions

The algorithms for general functions (without a weight function) are based on Gauss-Kronrod rules.

A Gauss-Kronrod rule begins with a classical Gaussian quadrature rule of order m. This is extended with additional points between each of the abscissae to give a higher order Kronrod rule of order $2m+1$. The Kronrod rule is efficient because it reuses existing function evaluations from the Gaussian rule.

The higher order Kronrod rule is used as the best approximation to the integral, and the difference between the two rules is used as an estimate of the error in the approximation.

17.1.2 Integrands with weight functions

For integrands with weight functions the algorithms use Clenshaw-Curtis quadrature rules.

A Clenshaw-Curtis rule begins with an n-th order Chebyshev polynomial approximation to the integrand. This polynomial can be integrated exactly to give an approximation to the integral of the original function. The Chebyshev expansion can be extended to higher orders to improve the approximation and provide an estimate of the error.

17.1.3 Integrands with singular weight functions

The presence of singularities (or other behavior) in the integrand can cause slow convergence in the Chebyshev approximation. The modified Clenshaw-Curtis rules used in QUADPACK separate out several common weight functions which cause slow convergence.

These weight functions are integrated analytically against the Chebyshev polynomials to precompute *modified Chebyshev moments*. Combining the moments with the Chebyshev approximation to the function gives the desired integral. The use of analytic integration for the singular part of the function allows exact cancellations and substantially improves the overall convergence behavior of the integration.

17.2 QNG non-adaptive Gauss-Kronrod integration

The QNG algorithm is a non-adaptive procedure which uses fixed Gauss-Kronrod-Patterson abscissae to sample the integrand at a maximum of 87 points. It is provided for fast integration of smooth functions.

int **gsl_integration_qng** (const *gsl_function* * *f*, double *a*, double *b*, double *epsabs*, double *epsrel*, double * *result*, double * *abserr*, size_t * *neval*)

This function applies the Gauss-Kronrod 10-point, 21-point, 43-point and 87-point integration rules in succession until an estimate of the integral of f over (a, b) is achieved within the desired absolute and relative error limits, `epsabs` and `epsrel`. The function returns the final approximation, `result`, an estimate of the absolute error, `abserr` and the number of function evaluations used, `neval`. The Gauss-Kronrod rules are designed in such a way that each rule uses all the results of its predecessors, in order to minimize the total number of function evaluations.

17.3 QAG adaptive integration

The QAG algorithm is a simple adaptive integration procedure. The integration region is divided into subintervals, and on each iteration the subinterval with the largest estimated error is bisected. This reduces the overall error rapidly, as the subintervals become concentrated around local difficulties in the integrand. These subintervals are managed by the following struct,

gsl_integration_workspace

This workspace handles the memory for the subinterval ranges, results and error estimates.

gsl_integration_workspace * **gsl_integration_workspace_alloc** (size_t *n*)

This function allocates a workspace sufficient to hold `n` double precision intervals, their integration results and error estimates. One workspace may be used multiple times as all necessary reinitialization is performed automatically by the integration routines.

void **gsl_integration_workspace_free** (*gsl_integration_workspace* * *w*)

This function frees the memory associated with the workspace `w`.

int **gsl_integration_qag** (const *gsl_function* * *f*, double *a*, double *b*, double *epsabs*, double *epsrel*, size_t *limit*, int *key*, *gsl_integration_workspace* * *workspace*, double * *result*, double * *abserr*)

This function applies an integration rule adaptively until an estimate of the integral of f over (a, b) is achieved within the desired absolute and relative error limits, `epsabs` and `epsrel`. The function returns the final approximation, `result`, and an estimate of the absolute error, `abserr`. The integration rule is determined by the value of `key`, which should be chosen from the following symbolic names,

- **GSL_INTEG_GAUSS15** (key = *1*)
- **GSL_INTEG_GAUSS21** (key = *2*)
- **GSL_INTEG_GAUSS31** (key = *3*)
- **GSL_INTEG_GAUSS41** (key = *4*)
- **GSL_INTEG_GAUSS51** (key = *5*)
- **GSL_INTEG_GAUSS61** (key = *6*)

corresponding to the 15, 21, 31, 41, 51 and 61 point Gauss-Kronrod rules. The higher-order rules give better accuracy for smooth functions, while lower-order rules save time when the function contains local difficulties, such as discontinuities.

On each iteration the adaptive integration strategy bisects the interval with the largest error estimate. The subintervals and their results are stored in the memory provided by `workspace`. The maximum number of subintervals is given by `limit`, which may not exceed the allocated size of the workspace.

17.4 QAGS adaptive integration with singularities

The presence of an integrable singularity in the integration region causes an adaptive routine to concentrate new subintervals around the singularity. As the subintervals decrease in size the successive approximations to the integral converge in a limiting fashion. This approach to the limit can be accelerated using an extrapolation procedure. The QAGS algorithm combines adaptive bisection with the Wynn epsilon-algorithm to speed up the integration of many types of integrable singularities.

int **gsl_integration_qags** (const *gsl_function* * *f*, double *a*, double *b*, double *epsabs*, double *epsrel*, size_t *limit*, *gsl_integration_workspace* * *workspace*, double * *result*, double * *abserr*)

This function applies the Gauss-Kronrod 21-point integration rule adaptively until an estimate of the integral of f over (a, b) is achieved within the desired absolute and relative error limits, `epsabs` and `epsrel`. The results are extrapolated using the epsilon-algorithm, which accelerates the convergence of the integral in the presence of discontinuities and integrable singularities. The function returns the final approximation from the extrapolation, `result`, and an estimate of the absolute error, `abserr`. The subintervals and their results are stored in the memory provided by `workspace`. The maximum number of subintervals is given by `limit`, which may not exceed the allocated size of the workspace.

17.5 QAGP adaptive integration with known singular points

int **gsl_integration_qagp** (const *gsl_function* * *f*, double * *pts*, size_t *npts*, double *epsabs*, double *epsrel*, size_t *limit*, *gsl_integration_workspace* * *workspace*, double * *result*, double * *abserr*)

This function applies the adaptive integration algorithm QAGS taking account of the user-supplied locations of singular points. The array `pts` of length `npts` should contain the endpoints of the integration ranges defined by the integration region and locations of the singularities. For example, to integrate over the region (a, b) with break-points at x_1, x_2, x_3 (where $a < x_1 < x_2 < x_3 < b$) the following `pts` array should be used:

```
pts[0] = a
pts[1] = x_1
pts[2] = x_2
pts[3] = x_3
pts[4] = b
```

with `npts` = 5.

If you know the locations of the singular points in the integration region then this routine will be faster than *gsl_integration_qags()*.

17.6 QAGI adaptive integration on infinite intervals

int **gsl_integration_qagi** (*gsl_function* * *f*, double *epsabs*, double *epsrel*, size_t *limit*, *gsl_integration_workspace* * *workspace*, double * *result*, double * *abserr*)

This function computes the integral of the function `f` over the infinite interval $(-\infty, +\infty)$. The integral is mapped onto the semi-open interval $(0, 1]$ using the transformation $x = (1 - t)/t$,

$$\int_{-\infty}^{+\infty} dx f(x) = \int_0^1 dt (f((1-t)/t) + f(-(1-t)/t))/t^2.$$

It is then integrated using the QAGS algorithm. The normal 21-point Gauss-Kronrod rule of QAGS is replaced by a 15-point rule, because the transformation can generate an integrable singularity at the origin. In this case a lower-order rule is more efficient.

int **gsl_integration_qagiu** (*gsl_function* * *f*, double *a*, double *epsabs*, double *epsrel*, size_t *limit*, *gsl_integration_workspace* * *workspace*, double * *result*, double * *abserr*)

This function computes the integral of the function `f` over the semi-infinite interval $(a, +\infty)$. The integral is mapped onto the semi-open interval $(0, 1]$ using the transformation $x = a + (1 - t)/t$,

$$\int_a^{+\infty} dx f(x) = \int_0^1 dt f(a + (1 - t)/t)/t^2$$

and then integrated using the QAGS algorithm.

int **gsl_integration_qagil** (*gsl_function* * *f*, double *b*, double *epsabs*, double *epsrel*, size_t *limit*, *gsl_integration_workspace* * *workspace*, double * *result*, double * *abserr*)

This function computes the integral of the function `f` over the semi-infinite interval $(-\infty, b)$. The integral is mapped onto the semi-open interval $(0, 1]$ using the transformation $x = b - (1 - t)/t$,

$$\int_{-\infty}^b dx f(x) = \int_0^1 dt f(b - (1 - t)/t)/t^2$$

and then integrated using the QAGS algorithm.

17.7 QAWC adaptive integration for Cauchy principal values

int **gsl_integration_qawc** (*gsl_function* * *f*, double *a*, double *b*, double *c*, double *epsabs*, double *epsrel*, size_t *limit*, *gsl_integration_workspace* * *workspace*, double * *result*, double * *abserr*)

This function computes the Cauchy principal value of the integral of f over (a, b), with a singularity at `c`,

$$I = \int_a^b dx \frac{f(x)}{x - c} = \lim_{\epsilon \to 0} \left\{ \int_a^{c-\epsilon} dx \frac{f(x)}{x - c} + \int_{c+\epsilon}^b dx \frac{f(x)}{x - c} \right\}$$

The adaptive bisection algorithm of QAG is used, with modifications to ensure that subdivisions do not occur at the singular point $x = c$. When a subinterval contains the point $x = c$ or is close to it then a special 25-point modified Clenshaw-Curtis rule is used to control the singularity. Further away from the singularity the algorithm uses an ordinary 15-point Gauss-Kronrod integration rule.

17.8 QAWS adaptive integration for singular functions

The QAWS algorithm is designed for integrands with algebraic-logarithmic singularities at the end-points of an integration region. In order to work efficiently the algorithm requires a precomputed table of Chebyshev moments.

gsl_integration_qaws_table

This structure contains precomputed quantities for the QAWS algorithm.

gsl_integration_qaws_table * **gsl_integration_qaws_table_alloc** (double *alpha*, double *beta*, int *mu*, int *nu*)

This function allocates space for a `gsl_integration_qaws_table` struct describing a singular weight function $w(x)$ with the parameters $(\alpha, \beta, \mu, \nu)$,

$$w(x) = (x - a)^\alpha (b - x)^\beta \log^\mu (x - a) \log^\nu (b - x)$$

where $\alpha > -1$, $\beta > -1$, and $\mu = 0, 1$, $\nu = 0, 1$. The weight function can take four different forms depending on the values of μ and ν,

Weight function $w(x)$	(μ, ν)
$(x-a)^\alpha (b-x)^\beta$	$(0,0)$
$(x-a)^\alpha (b-x)^\beta \log(x-a)$	$(1,0)$
$(x-a)^\alpha (b-x)^\beta \log(b-x)$	$(0,1)$
$(x-a)^\alpha (b-x)^\beta \log(x-a) \log(b-x)$	$(1,1)$

The singular points (a, b) do not have to be specified until the integral is computed, where they are the endpoints of the integration range.

The function returns a pointer to the newly allocated table `gsl_integration_qaws_table` if no errors were detected, and 0 in the case of error.

int **gsl_integration_qaws_table_set** (*gsl_integration_qaws_table* * *t*, double *alpha*, double *beta*, int *mu*, int *nu*)

This function modifies the parameters $(\alpha, \beta, \mu, \nu)$ of an existing `gsl_integration_qaws_table` struct `t`.

void **gsl_integration_qaws_table_free** (*gsl_integration_qaws_table* * *t*)

This function frees all the memory associated with the `gsl_integration_qaws_table` struct `t`.

int **gsl_integration_qaws** (*gsl_function* * *f*, const double *a*, const double *b*, *gsl_integration_qaws_table* * *t*, const double *epsabs*, const double *epsrel*, const size_t *limit*, *gsl_integration_workspace* * *workspace*, double * *result*, double * *abserr*)

This function computes the integral of the function $f(x)$ over the interval (a, b) with the singular weight function $(x-a)^\alpha (b-x)^\beta \log^\mu(x-a) \log^\nu(b-x)$. The parameters of the weight function $(\alpha, \beta, \mu, \nu)$ are taken from the table `t`. The integral is,

$$I = \int_a^b dx f(x) (x-a)^\alpha (b-x)^\beta \log^\mu(x-a) \log^\nu(b-x).$$

The adaptive bisection algorithm of QAG is used. When a subinterval contains one of the endpoints then a special 25-point modified Clenshaw-Curtis rule is used to control the singularities. For subintervals which do not include the endpoints an ordinary 15-point Gauss-Kronrod integration rule is used.

17.9 QAWO adaptive integration for oscillatory functions

The QAWO algorithm is designed for integrands with an oscillatory factor, $\sin(\omega x)$ or $\cos(\omega x)$. In order to work efficiently the algorithm requires a table of Chebyshev moments which must be pre-computed with calls to the functions below.

gsl_integration_qawo_table * **gsl_integration_qawo_table_alloc** (double *omega*, double *L*, enum gsl_integration_qawo_enum *sine*, size_t *n*)

This function allocates space for a `gsl_integration_qawo_table` struct and its associated workspace describing a sine or cosine weight function $w(x)$ with the parameters (ω, L),

$$w(x) = \left\{ \begin{array}{c} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\}$$

The parameter `L` must be the length of the interval over which the function will be integrated $L = b - a$. The choice of sine or cosine is made with the parameter `sine` which should be chosen from one of the two following symbolic values:

GSL_INTEG_COSINE

GSL_INTEG_SINE

The gsl_integration_qawo_table is a table of the trigonometric coefficients required in the integration process. The parameter n determines the number of levels of coefficients that are computed. Each level corresponds to one bisection of the interval L, so that n levels are sufficient for subintervals down to the length $L/2^n$. The integration routine *gsl_integration_qawo()* returns the error GSL_ETABLE if the number of levels is insufficient for the requested accuracy.

int **gsl_integration_qawo_table_set** (gsl_integration_qawo_table * *t*, double *omega*, double *L*, enum gsl_integration_qawo_enum *sine*)
This function changes the parameters omega, L and sine of the existing workspace t.

int **gsl_integration_qawo_table_set_length** (gsl_integration_qawo_table * *t*, double *L*)
This function allows the length parameter L of the workspace t to be changed.

void **gsl_integration_qawo_table_free** (gsl_integration_qawo_table * *t*)
This function frees all the memory associated with the workspace t.

int **gsl_integration_qawo** (*gsl_function* * *f*, const double *a*, const double *epsabs*, const double *epsrel*, const size_t *limit*, *gsl_integration_workspace* * *workspace*, gsl_integration_qawo_table * *wf*, double * *result*, double * *abserr*)
This function uses an adaptive algorithm to compute the integral of f over (a, b) with the weight function $\sin(\omega x)$ or $\cos(\omega x)$ defined by the table wf,

$$I = \int_a^b dx f(x) \left\{ \begin{array}{c} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\}$$

The results are extrapolated using the epsilon-algorithm to accelerate the convergence of the integral. The function returns the final approximation from the extrapolation, result, and an estimate of the absolute error, abserr. The subintervals and their results are stored in the memory provided by workspace. The maximum number of subintervals is given by limit, which may not exceed the allocated size of the workspace.

Those subintervals with "large" widths d where $d\omega > 4$ are computed using a 25-point Clenshaw-Curtis integration rule, which handles the oscillatory behavior. Subintervals with a "small" widths where $d\omega < 4$ are computed using a 15-point Gauss-Kronrod integration.

17.10 QAWF adaptive integration for Fourier integrals

int **gsl_integration_qawf** (*gsl_function* * *f*, const double *a*, const double *epsabs*, const size_t *limit*, *gsl_integration_workspace* * *workspace*, *gsl_integration_workspace* * *cycle_workspace*, gsl_integration_qawo_table * *wf*, double * *result*, double * *abserr*)
This function attempts to compute a Fourier integral of the function f over the semi-infinite interval $[a, +\infty)$

$$I = \int_a^{+\infty} dx f(x) \left\{ \begin{array}{c} \sin(\omega x) \\ \cos(\omega x) \end{array} \right\}$$

The parameter ω and choice of sin or cos is taken from the table wf (the length L can take any value, since it is overridden by this function to a value appropriate for the Fourier integration). The integral is computed using the QAWO algorithm over each of the subintervals,

$$\begin{aligned} C_1 &= [a, a + c] \\ C_2 &= [a + c, a + 2c] \\ \ldots &= \ldots \\ C_k &= [a + (k - 1)c, a + kc] \end{aligned}$$

where $c = (2floor(|\omega|)+1)\pi/|\omega|$. The width c is chosen to cover an odd number of periods so that the contributions from the intervals alternate in sign and are monotonically decreasing when f is positive and monotonically decreasing. The sum of this sequence of contributions is accelerated using the epsilon-algorithm.

This function works to an overall absolute tolerance of `abserr`. The following strategy is used: on each interval C_k the algorithm tries to achieve the tolerance

$$TOL_k = u_k abserr$$

where $u_k = (1-p)p^{k-1}$ and $p = 9/10$. The sum of the geometric series of contributions from each interval gives an overall tolerance of `abserr`.

If the integration of a subinterval leads to difficulties then the accuracy requirement for subsequent intervals is relaxed,

$$TOL_k = u_k \max\left(abserr, \max_{i<k}\left(E_i\right)\right)$$

where E_k is the estimated error on the interval C_k.

The subintervals and their results are stored in the memory provided by `workspace`. The maximum number of subintervals is given by `limit`, which may not exceed the allocated size of the workspace. The integration over each subinterval uses the memory provided by `cycle_workspace` as workspace for the QAWO algorithm.

17.11 CQUAD doubly-adaptive integration

CQUAD is a new doubly-adaptive general-purpose quadrature routine which can handle most types of singularities, non-numerical function values such as `Inf` or `NaN`, as well as some divergent integrals. It generally requires more function evaluations than the integration routines in QUADPACK, yet fails less often for difficult integrands.

The underlying algorithm uses a doubly-adaptive scheme in which Clenshaw-Curtis quadrature rules of increasing degree are used to compute the integral in each interval. The L_2-norm of the difference between the underlying interpolatory polynomials of two successive rules is used as an error estimate. The interval is subdivided if the difference between two successive rules is too large or a rule of maximum degree has been reached.

gsl_integration_cquad_workspace * **gsl_integration_cquad_workspace_alloc** (size_t *n*)
This function allocates a workspace sufficient to hold the data for `n` intervals. The number `n` is not the maximum number of intervals that will be evaluated. If the workspace is full, intervals with smaller error estimates will be discarded. A minimum of 3 intervals is required and for most functions, a workspace of size 100 is sufficient.

void **gsl_integration_cquad_workspace_free** (gsl_integration_cquad_workspace * *w*)
This function frees the memory associated with the workspace `w`.

int **gsl_integration_cquad** (const *gsl_function* * *f*, double *a*, double *b*, double *epsabs*, double *epsrel*, gsl_integration_cquad_workspace * *workspace*, double * *result*, double * *abserr*, size_t * *nevals*)
This function computes the integral of f over (a, b) within the desired absolute and relative error limits, `epsabs` and `epsrel` using the CQUAD algorithm. The function returns the final approximation, `result`, an estimate of the absolute error, `abserr`, and the number of function evaluations required, `nevals`.

The CQUAD algorithm divides the integration region into subintervals, and in each iteration, the subinterval with the largest estimated error is processed. The algorithm uses Clenshaw-Curtis quadrature rules of degree 4, 8, 16 and 32 over 5, 9, 17 and 33 nodes respectively. Each interval is initialized with the lowest-degree rule. When an interval is processed, the next-higher degree rule is evaluated and an error estimate is computed based on the L_2-norm of the difference between the underlying interpolating polynomials of both rules. If the highest-degree rule has already been used, or the interpolatory polynomials differ significantly, the interval is bisected.

The subintervals and their results are stored in the memory provided by `workspace`. If the error estimate or the number of function evaluations is not needed, the pointers `abserr` and `nevals` can be set to `NULL`.

17.12 Romberg integration

The Romberg integration method estimates the definite integral

$$I = \int_a^b f(x)dx$$

by applying Richardson extrapolation on the trapezoidal rule, using equally spaced points with spacing

$$h_k = (b-a)2^{-k}$$

for $k = 1, \ldots, n$. For each k, Richardson extrapolation is used $k-1$ times on previous approximations to improve the order of accuracy as much as possible. Romberg integration typically works well (and converges quickly) for smooth integrands with no singularities in the interval or at the end points.

gsl_integration_romberg_workspace * **gsl_integration_romberg_alloc** (const size_t *n*)
This function allocates a workspace for Romberg integration, specifying a maximum of n iterations, or divisions of the interval. Since the number of divisions is $2^n + 1$, n can be kept relatively small (i.e. 10 or 20). It is capped at a maximum value of 30 to prevent overflow. The size of the workspace is $O(2n)$.

void **gsl_integration_romberg_free** (gsl_integration_romberg_workspace * *w*)
This function frees the memory associated with the workspace `w`.

int **gsl_integration_romberg** (const *gsl_function* * *f*, const double *a*, const double *b*, const double *epsabs*, const double *epsrel*, double * *result*, size_t * *neval*, gsl_integration_romberg_workspace * *w*)
This function integrates $f(x)$, specified by `f`, from `a` to `b`, storing the answer in `result`. At each step in the iteration, convergence is tested by checking:

$$|I_k - I_{k-1}| \leq \max(epsabs, epsrel \times |I_k|)$$

where I_k is the current approximation and I_{k-1} is the approximation of the previous iteration. If the method does not converge within the previously specified n iterations, the function stores the best current estimate in `result` and returns `GSL_EMAXITER`. If the method converges, the function returns `GSL_SUCCESS`. The total number of function evaluations is returned in `neval`.

17.13 Gauss-Legendre integration

The fixed-order Gauss-Legendre integration routines are provided for fast integration of smooth functions with known polynomial order. The n-point Gauss-Legendre rule is exact for polynomials of order $2n-1$ or less. For example, these rules are useful when integrating basis functions to form mass matrices for the Galerkin method. Unlike other numerical integration routines within the library, these routines do not accept absolute or relative error bounds.

gsl_integration_glfixed_table * **gsl_integration_glfixed_table_alloc** (size_t *n*)
This function determines the Gauss-Legendre abscissae and weights necessary for an n-point fixed order integration scheme. If possible, high precision precomputed coefficients are used. If precomputed weights are not available, lower precision coefficients are computed on the fly.

double **gsl_integration_glfixed** (const *gsl_function* * *f*, double *a*, double *b*, const gsl_integration_glfixed_table * *t*)
This function applies the Gauss-Legendre integration rule contained in table `t` and returns the result.

int **gsl_integration_glfixed_point** (double *a*, double *b*, size_t *i*, double * *xi*, double * *wi*, const gsl_integration_glfixed_table * *t*)
For `i` in $[0, \ldots, n-1]$, this function obtains the `i`-th Gauss-Legendre point `xi` and weight `wi` on the interval [`a`, `b`]. The points and weights are ordered by increasing point value. A function f may be integrated on [`a`, `b`] by summing $wi * f(xi)$ over `i`.

void **gsl_integration_glfixed_table_free**(gsl_integration_glfixed_table * *t*)
This function frees the memory associated with the table `t`.

17.14 Fixed point quadratures

The routines in this section approximate an integral by the sum

$$\int_a^b w(x)f(x)dx = \sum_{i=1}^{n} w_i f(x_i)$$

where $f(x)$ is the function to be integrated and $w(x)$ is a weighting function. The n weights w_i and nodes x_i are carefully chosen so that the result is exact when $f(x)$ is a polynomial of degree $2n-1$ or less. Once the user chooses the order n and weighting function $w(x)$, the weights w_i and nodes x_i can be precomputed and used to efficiently evaluate integrals for any number of functions $f(x)$.

This method works best when $f(x)$ is well approximated by a polynomial on the interval (a, b), and so is not suitable for functions with singularities. Since the user specifies ahead of time how many quadrature nodes will be used, these routines do not accept absolute or relative error bounds. The table below lists the weighting functions currently supported.

Name	Interval	Weighting function $w(x)$	Constraints
Legendre	(a,b)	1	$b > a$
Chebyshev Type 1	(a,b)	$1/\sqrt{(b-x)(x-a)}$	$b > a$
Gegenbauer	(a,b)	$((b-x)(x-a))^\alpha$	$\alpha > -1, b > a$
Jacobi	(a,b)	$(b-x)^\alpha(x-a)^\beta$	$\alpha, \beta > -1, b > a$
Laguerre	(a,∞)	$(x-a)^\alpha \exp(-b(x-a))$	$\alpha > -1, b > 0$
Hermite	$(-\infty,\infty)$	$\lvert x-a\rvert^\alpha \exp(-b(x-a)^2)$	$\alpha > -1, b > 0$
Exponential	(a,b)	$\lvert x-(a+b)/2\rvert^\alpha$	$\alpha > -1, b > a$
Rational	(a,∞)	$(x-a)^\alpha(x+b)^\beta$	$\alpha > -1, \alpha+\beta+2n < 0, a+b > 0$
Chebyshev Type 2	(a,b)	$\sqrt{(b-x)(x-a)}$	$b > a$

The fixed point quadrature routines use the following workspace to store the nodes and weights, as well as additional variables for intermediate calculations:

gsl_integration_fixed_workspace
This workspace is used for fixed point quadrature rules and looks like this:

```
typedef struct
{
  size_t n;         /* number of nodes/weights */
  double *weights;  /* quadrature weights */
  double *x;        /* quadrature nodes */
  double *diag;     /* diagonal of Jacobi matrix */
  double *subdiag;  /* subdiagonal of Jacobi matrix */
  const gsl_integration_fixed_type * type;
} gsl_integration_fixed_workspace;
```

gsl_integration_fixed_workspace * **gsl_integration_fixed_alloc**(const *gsl_integration_fixed_type* * *T*, const size_t *n*, const double *a*, const double *b*, const double *alpha*, const double *beta*)
This function allocates a workspace for computing integrals with interpolating quadratures using `n` quadrature nodes. The parameters `a`, `b`, *alpha*, and `beta` specify the integration interval and/or weighting function for

the various quadrature types. See the *table* above for constraints on these parameters. The size of the workspace is $O(4n)$.

gsl_integration_fixed_type
The type of quadrature used is specified by `T` which can be set to the following choices:

gsl_integration_fixed_legendre
This specifies Legendre quadrature integration. The parameters `alpha` and `beta` are ignored for this type.

gsl_integration_fixed_chebyshev
This specifies Chebyshev type 1 quadrature integration. The parameters `alpha` and `beta` are ignored for this type.

gsl_integration_fixed_gegenbauer
This specifies Gegenbauer quadrature integration. The parameter `beta` is ignored for this type.

gsl_integration_fixed_jacobi
This specifies Jacobi quadrature integration.

gsl_integration_fixed_laguerre
This specifies Laguerre quadrature integration. The parameter `beta` is ignored for this type.

gsl_integration_fixed_hermite
This specifies Hermite quadrature integration. The parameter `beta` is ignored for this type.

gsl_integration_fixed_exponential
This specifies exponential quadrature integration. The parameter `beta` is ignored for this type.

gsl_integration_fixed_rational
This specifies rational quadrature integration.

gsl_integration_fixed_chebyshev2
This specifies Chebyshev type 2 quadrature integration. The parameters `alpha` and `beta` are ignored for this type.

void **gsl_integration_fixed_free** (*gsl_integration_fixed_workspace* * *w*)
This function frees the memory assocated with the workspace `w`

size_t **gsl_integration_fixed_n** (const *gsl_integration_fixed_workspace* * *w*)
This function returns the number of quadrature nodes and weights.

double * **gsl_integration_fixed_nodes** (const *gsl_integration_fixed_workspace* * *w*)
This function returns a pointer to an array of size `n` containing the quadrature nodes x_i.

double * **gsl_integration_fixed_weights** (const *gsl_integration_fixed_workspace* * *w*)
This function returns a pointer to an array of size `n` containing the quadrature weights w_i.

int **gsl_integration_fixed** (const *gsl_function* * *func*, double * *result*, const *gsl_integration_fixed_workspace* * *w*)
This function integrates the function $f(x)$ provided in `func` using previously computed fixed quadrature rules. The integral is approximated as

$$\sum_{i=1}^{n} w_i f(x_i)$$

where w_i are the quadrature weights and x_i are the quadrature nodes computed previously by *gsl_integration_fixed_alloc()*. The sum is stored in `result` on output.

17.15 Error codes

In addition to the standard error codes for invalid arguments the functions can return the following values,

GSL_EMAXITER	the maximum number of subdivisions was exceeded.
GSL_EROUND	cannot reach tolerance because of roundoff error, or roundoff error was detected in the extrapolation table.
GSL_ESING	a non-integrable singularity or other bad integrand behavior was found in the integration interval.
GSL_EDIVERGE	the integral is divergent, or too slowly convergent to be integrated numerically.
GSL_EDOM	error in the values of the input arguments

17.16 Examples

17.16.1 Adaptive integration example

The integrator QAGS will handle a large class of definite integrals. For example, consider the following integral, which has an algebraic-logarithmic singularity at the origin,

$$\int_0^1 x^{-1/2} \log(x) dx = -4$$

The program below computes this integral to a relative accuracy bound of 1e-7.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f (double x, void * params) {
  double alpha = *(double *) params;
  double f = log(alpha*x) / sqrt(x);
  return f;
}

int
main (void)
{
  gsl_integration_workspace * w
    = gsl_integration_workspace_alloc (1000);

  double result, error;
  double expected = -4.0;
  double alpha = 1.0;

  gsl_function F;
  F.function = &f;
  F.params = &alpha;

  gsl_integration_qags (&F, 0, 1, 0, 1e-7, 1000,
                        w, &result, &error);

  printf ("result          = % .18f\n", result);
  printf ("exact result    = % .18f\n", expected);
```

(continues on next page)

(continued from previous page)

```
  printf ("estimated error = % .18f\n", error);
  printf ("actual error    = % .18f\n", result - expected);
  printf ("intervals       = %zu\n", w->size);

  gsl_integration_workspace_free (w);

  return 0;
}
```

The results below show that the desired accuracy is achieved after 8 subdivisions.

```
result          = -4.000000000000085265
exact result    = -4.000000000000000000
estimated error =  0.000000000000135447
actual error    = -0.000000000000085265
intervals       = 8
```

In fact, the extrapolation procedure used by `QAGS` produces an accuracy of almost twice as many digits. The error estimate returned by the extrapolation procedure is larger than the actual error, giving a margin of safety of one order of magnitude.

17.16.2 Fixed-point quadrature example

In this example, we use a fixed-point quadrature rule to integrate the integral

$$\int_{-\infty}^{\infty} e^{-x^2} \left(x^m + 1\right) dx = \begin{cases} \sqrt{\pi} + \Gamma\left(\frac{m+1}{2}\right), & m \text{ even} \\ \sqrt{\pi}, & m \text{ odd} \end{cases}$$

for integer m. Consulting our *table* of fixed point quadratures, we see that this integral can be evaluated with a Hermite quadrature rule, setting $\alpha = 0, a = 0, b = 1$. Since we are integrating a polynomial of degree m, we need to choose the number of nodes $n \geq (m + 1)/2$ to achieve the best results.

First we will try integrating for $m = 10, n = 5$, which does not satisfy our criteria above:

```
$ ./integration2 10 5
```

The output is,

```
m               = 10
intervals       = 5
result          =  47.468529694563351029
exact result    =  54.115231635459025483
actual error    = -6.646701940895674454
```

So, we find a large error. Now we try integrating for $m = 10, n = 6$ which does satisfy the criteria above:

```
$ ./integration2 10 6
```

The output is,

```
m               = 10
intervals       = 6
result          =  54.115231635459096537
exact result    =  54.115231635459025483
actual error    =  0.000000000000071054
```

The program is given below.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>
#include <gsl/gsl_sf_gamma.h>

double
f(double x, void * params)
{
  int m = *(int *) params;
  double f = gsl_pow_int(x, m) + 1.0;
  return f;
}

int
main (int argc, char *argv[])
{
  gsl_integration_fixed_workspace * w;
  const gsl_integration_fixed_type * T = gsl_integration_fixed_hermite;
  int m = 10;
  int n = 6;
  double expected, result;
  gsl_function F;

  if (argc > 1)
    m = atoi(argv[1]);

  if (argc > 2)
    n = atoi(argv[2]);

  w = gsl_integration_fixed_alloc(T, n, 0.0, 1.0, 0.0, 0.0);

  F.function = &f;
  F.params = &m;

  gsl_integration_fixed(&F, &result, w);

  if (m % 2 == 0)
    expected = M_SQRTPI + gsl_sf_gamma(0.5*(1.0 + m));
  else
    expected = M_SQRTPI;

  printf ("m               = %d\n", m);
  printf ("intervals       = %zu\n", gsl_integration_fixed_n(w));
  printf ("result          = % .18f\n", result);
  printf ("exact result    = % .18f\n", expected);
  printf ("actual error    = % .18f\n", result - expected);

  gsl_integration_fixed_free (w);

  return 0;
}
```

17.17 References and Further Reading

The following book is the definitive reference for QUADPACK, and was written by the original authors. It provides descriptions of the algorithms, program listings, test programs and examples. It also includes useful advice on numerical integration and many references to the numerical integration literature used in developing QUADPACK.

- R. Piessens, E. de Doncker-Kapenga, C.W. Ueberhuber, D.K. Kahaner. QUADPACK A subroutine package for automatic integration Springer Verlag, 1983.

The CQUAD integration algorithm is described in the following paper:

- P. Gonnet, "Increasing the Reliability of Adaptive Quadrature Using Explicit Interpolants", ACM Transactions on Mathematical Software, Volume 37 (2010), Issue 3, Article 26.

The fixed-point quadrature routines are based on IQPACK, described in the following papers:

- S. Elhay, J. Kautsky, Algorithm 655: IQPACK, FORTRAN Subroutines for the Weights of Interpolatory Quadrature, ACM Transactions on Mathematical Software, Volume 13, Number 4, December 1987, pages 399-415.
- J. Kautsky, S. Elhay, Calculation of the Weights of Interpolatory Quadratures, Numerische Mathematik, Volume 40, Number 3, October 1982, pages 407-422.