CHAPTER

SEVEN

SPECIAL FUNCTIONS

This chapter describes the GSL special function library. The library includes routines for calculating the values of Airy functions, Bessel functions, Clausen functions, Coulomb wave functions, Coupling coefficients, the Dawson function, Debye functions, Dilogarithms, Elliptic integrals, Jacobi elliptic functions, Error functions, Exponential integrals, Fermi-Dirac functions, Gamma functions, Gegenbauer functions, Hermite polynomials and functions, Hypergeometric functions, Laguerre functions, Legendre functions and Spherical Harmonics, the Psi (Digamma) Function, Synchrotron functions, Transport functions, Trigonometric functions and Zeta functions. Each routine also computes an estimate of the numerical error in the calculated value of the function.

The functions in this chapter are declared in individual header files, such as `gsl_sf_airy.h`, `gsl_sf_bessel.h`, etc. The complete set of header files can be included using the file `gsl_sf.h`.

7.1 Usage

The special functions are available in two calling conventions, a *natural form* which returns the numerical value of the function and an *error-handling form* which returns an error code. The two types of function provide alternative ways of accessing the same underlying code.

The *natural form* returns only the value of the function and can be used directly in mathematical expressions. For example, the following function call will compute the value of the Bessel function $J_0(x)$:

```
double y = gsl_sf_bessel_J0 (x);
```

There is no way to access an error code or to estimate the error using this method. To allow access to this information the alternative error-handling form stores the value and error in a modifiable argument:

```
gsl_sf_result result;
int status = gsl_sf_bessel_J0_e (x, &result);
```

The error-handling functions have the suffix `_e`. The returned status value indicates error conditions such as overflow, underflow or loss of precision. If there are no errors the error-handling functions return `GSL_SUCCESS`.

7.2 The gsl_sf_result struct

The error handling form of the special functions always calculate an error estimate along with the value of the result. Therefore, structures are provided for amalgamating a value and error estimate. These structures are declared in the header file `gsl_sf_result.h`.

The following struct contains value and error fields.

gsl_sf_result

```
typedef struct
{
  double val;
  double err;
} gsl_sf_result;
```

The field `val` contains the value and the field `err` contains an estimate of the absolute error in the value.

In some cases, an overflow or underflow can be detected and handled by a function. In this case, it may be possible to return a scaling exponent as well as an error/value pair in order to save the result from exceeding the dynamic range of the built-in types. The following struct contains value and error fields as well as an exponent field such that the actual result is obtained as `result * 10^(e10)`.

gsl_sf_result_e10

```
typedef struct
{
  double val;
  double err;
  int    e10;
} gsl_sf_result_e10;
```

7.3 Modes

The goal of the library is to achieve double precision accuracy wherever possible. However the cost of evaluating some special functions to double precision can be significant, particularly where very high order terms are required. In these cases a `mode` argument, of type *gsl_mode_t* allows the accuracy of the function to be reduced in order to improve performance. The following precision levels are available for the mode argument,

gsl_mode_t

GSL_PREC_DOUBLE
Double-precision, a relative accuracy of approximately $2 * 10^{-16}$.

GSL_PREC_SINGLE
Single-precision, a relative accuracy of approximately 10^{-7}.

GSL_PREC_APPROX
Approximate values, a relative accuracy of approximately $5 * 10^{-4}$.

The approximate mode provides the fastest evaluation at the lowest accuracy.

7.4 Airy Functions and Derivatives

The Airy functions $Ai(x)$ and $Bi(x)$ are defined by the integral representations,

$$Ai(x) = \frac{1}{\pi} \int_0^\infty \cos(t^3/3 + xt)\, dt$$
$$Bi(x) = \frac{1}{\pi} \int_0^\infty (e^{-t^3/3+xt} + \sin(t^3/3 + xt))\, dt$$

For further information see Abramowitz & Stegun, Section 10.4. The Airy functions are defined in the header file `gsl_sf_airy.h`.

7.4.1 Airy Functions

double **gsl_sf_airy_Ai** (double *x*, *gsl_mode_t mode*)
int **gsl_sf_airy_Ai_e** (double *x*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute the Airy function $Ai(x)$ with an accuracy specified by `mode`.

double **gsl_sf_airy_Bi** (double *x*, *gsl_mode_t mode*)
int **gsl_sf_airy_Bi_e** (double *x*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute the Airy function $Bi(x)$ with an accuracy specified by `mode`.

double **gsl_sf_airy_Ai_scaled** (double *x*, *gsl_mode_t mode*)
int **gsl_sf_airy_Ai_scaled_e** (double *x*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute a scaled version of the Airy function $S_A(x) Ai(x)$. For $x > 0$ the scaling factor $S_A(x)$ is $\exp(+(2/3)x^{3/2})$, and is 1 for $x < 0$.

double **gsl_sf_airy_Bi_scaled** (double *x*, *gsl_mode_t mode*)
int **gsl_sf_airy_Bi_scaled_e** (double *x*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute a scaled version of the Airy function $S_B(x) Bi(x)$. For $x > 0$ the scaling factor $S_B(x)$ is $exp(-(2/3)x^{3/2})$, and is 1 for $x < 0$.

7.4.2 Derivatives of Airy Functions

double **gsl_sf_airy_Ai_deriv** (double *x*, *gsl_mode_t mode*)
int **gsl_sf_airy_Ai_deriv_e** (double *x*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute the Airy function derivative $Ai'(x)$ with an accuracy specified by `mode`.

double **gsl_sf_airy_Bi_deriv** (double *x*, *gsl_mode_t mode*)
int **gsl_sf_airy_Bi_deriv_e** (double *x*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute the Airy function derivative $Bi'(x)$ with an accuracy specified by `mode`.

double **gsl_sf_airy_Ai_deriv_scaled** (double *x*, *gsl_mode_t mode*)
int **gsl_sf_airy_Ai_deriv_scaled_e** (double *x*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute the scaled Airy function derivative $S_A(x) Ai'(x)$. For $x > 0$ the scaling factor $S_A(x)$ is $\exp(+(2/3)x^{3/2})$, and is 1 for $x < 0$.

double **gsl_sf_airy_Bi_deriv_scaled** (double *x*, *gsl_mode_t mode*)
int **gsl_sf_airy_Bi_deriv_scaled_e** (double *x*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute the scaled Airy function derivative $S_B(x) Bi'(x)$. For $x > 0$ the scaling factor $S_B(x)$ is $exp(-(2/3)x^{3/2})$, and is 1 for $x < 0$.

7.4.3 Zeros of Airy Functions

double **gsl_sf_airy_zero_Ai** (unsigned int *s*)
int **gsl_sf_airy_zero_Ai_e** (unsigned int *s*, *gsl_sf_result * result*)
These routines compute the location of the `s`-th zero of the Airy function $Ai(x)$.

double **gsl_sf_airy_zero_Bi** (unsigned int *s*)
int **gsl_sf_airy_zero_Bi_e** (unsigned int *s*, *gsl_sf_result * result*)
These routines compute the location of the `s`-th zero of the Airy function $Bi(x)$.

7.4.4 Zeros of Derivatives of Airy Functions

double **gsl_sf_airy_zero_Ai_deriv** (unsigned int *s*)
int **gsl_sf_airy_zero_Ai_deriv_e** (unsigned int *s*, *gsl_sf_result* * *result*)
These routines compute the location of the `s`-th zero of the Airy function derivative $Ai'(x)$.

double **gsl_sf_airy_zero_Bi_deriv** (unsigned int *s*)
int **gsl_sf_airy_zero_Bi_deriv_e** (unsigned int *s*, *gsl_sf_result* * *result*)
These routines compute the location of the `s`-th zero of the Airy function derivative $Bi'(x)$.

7.5 Bessel Functions

The routines described in this section compute the Cylindrical Bessel functions $J_n(x)$, $Y_n(x)$, Modified cylindrical Bessel functions $I_n(x)$, $K_n(x)$, Spherical Bessel functions $j_l(x)$, $y_l(x)$, and Modified Spherical Bessel functions $i_l(x)$, $k_l(x)$. For more information see Abramowitz & Stegun, Chapters 9 and 10. The Bessel functions are defined in the header file `gsl_sf_bessel.h`.

7.5.1 Regular Cylindrical Bessel Functions

double **gsl_sf_bessel_J0** (double *x*)
int **gsl_sf_bessel_J0_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the regular cylindrical Bessel function of zeroth order, $J_0(x)$.

double **gsl_sf_bessel_J1** (double *x*)
int **gsl_sf_bessel_J1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the regular cylindrical Bessel function of first order, $J_1(x)$.

double **gsl_sf_bessel_Jn** (int *n*, double *x*)
int **gsl_sf_bessel_Jn_e** (int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the regular cylindrical Bessel function of order `n`, $J_n(x)$.

int **gsl_sf_bessel_Jn_array** (int *nmin*, int *nmax*, double *x*, double *result_array[]*)
This routine computes the values of the regular cylindrical Bessel functions $J_n(x)$ for n from `nmin` to `nmax` inclusive, storing the results in the array `result_array`. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

7.5.2 Irregular Cylindrical Bessel Functions

double **gsl_sf_bessel_Y0** (double *x*)
int **gsl_sf_bessel_Y0_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular cylindrical Bessel function of zeroth order, $Y_0(x)$, for $x > 0$.

double **gsl_sf_bessel_Y1** (double *x*)
int **gsl_sf_bessel_Y1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular cylindrical Bessel function of first order, $Y_1(x)$, for $x > 0$.

double **gsl_sf_bessel_Yn** (int *n*, double *x*)
int **gsl_sf_bessel_Yn_e** (int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular cylindrical Bessel function of order `n`, $Y_n(x)$, for $x > 0$.

int **gsl_sf_bessel_Yn_array** (int *nmin*, int *nmax*, double *x*, double *result_array[]*)
This routine computes the values of the irregular cylindrical Bessel functions $Y_n(x)$ for n from `nmin` to `nmax` inclusive, storing the results in the array `result_array`. The domain of the function is $x > 0$. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

7.5.3 Regular Modified Cylindrical Bessel Functions

double **gsl_sf_bessel_I0** (double *x*)
int **gsl_sf_bessel_I0_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the regular modified cylindrical Bessel function of zeroth order, $I_0(x)$.

double **gsl_sf_bessel_I1** (double *x*)
int **gsl_sf_bessel_I1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the regular modified cylindrical Bessel function of first order, $I_1(x)$.

double **gsl_sf_bessel_In** (int *n*, double *x*)
int **gsl_sf_bessel_In_e** (int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the regular modified cylindrical Bessel function of order `n`, $I_n(x)$.

int **gsl_sf_bessel_In_array** (int *nmin*, int *nmax*, double *x*, double *result_array[]*)
This routine computes the values of the regular modified cylindrical Bessel functions $I_n(x)$ for n from `nmin` to `nmax` inclusive, storing the results in the array `result_array`. The start of the range `nmin` must be positive or zero. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

double **gsl_sf_bessel_I0_scaled** (double *x*)
int **gsl_sf_bessel_I0_scaled_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled regular modified cylindrical Bessel function of zeroth order $\exp(-|x|)I_0(x)$.

double **gsl_sf_bessel_I1_scaled** (double *x*)
int **gsl_sf_bessel_I1_scaled_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled regular modified cylindrical Bessel function of first order $\exp(-|x|)I_1(x)$.

double **gsl_sf_bessel_In_scaled** (int *n*, double *x*)
int **gsl_sf_bessel_In_scaled_e** (int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled regular modified cylindrical Bessel function of order `n`, $\exp(-|x|)I_n(x)$

int **gsl_sf_bessel_In_scaled_array** (int *nmin*, int *nmax*, double *x*, double *result_array[]*)
This routine computes the values of the scaled regular cylindrical Bessel functions $\exp(-|x|)I_n(x)$ for n from `nmin` to `nmax` inclusive, storing the results in the array `result_array`. The start of the range `nmin` must be positive or zero. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

7.5.4 Irregular Modified Cylindrical Bessel Functions

double **gsl_sf_bessel_K0** (double *x*)
int **gsl_sf_bessel_K0_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular modified cylindrical Bessel function of zeroth order, $K_0(x)$, for $x > 0$.

double **gsl_sf_bessel_K1** (double *x*)
int **gsl_sf_bessel_K1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular modified cylindrical Bessel function of first order, $K_1(x)$, for $x > 0$.

double **gsl_sf_bessel_Kn** (int *n*, double *x*)
int **gsl_sf_bessel_Kn_e** (int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular modified cylindrical Bessel function of order `n`, $K_n(x)$, for $x > 0$.

int **gsl_sf_bessel_Kn_array** (int *nmin*, int *nmax*, double *x*, double *result_array[]*)
This routine computes the values of the irregular modified cylindrical Bessel functions $K_n(x)$ for n from `nmin` to `nmax` inclusive, storing the results in the array `result_array`. The start of the range `nmin` must be positive or zero. The domain of the function is $x > 0$. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

double **gsl_sf_bessel_K0_scaled** (double *x*)
int **gsl_sf_bessel_K0_scaled_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled irregular modified cylindrical Bessel function of zeroth order $\exp(x)K_0(x)$ for $x > 0$.

double **gsl_sf_bessel_K1_scaled** (double *x*)
int **gsl_sf_bessel_K1_scaled_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled irregular modified cylindrical Bessel function of first order $\exp(x)K_1(x)$ for $x > 0$.

double **gsl_sf_bessel_Kn_scaled** (int *n*, double *x*)
int **gsl_sf_bessel_Kn_scaled_e** (int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled irregular modified cylindrical Bessel function of order `n`, $\exp(x)K_n(x)$, for $x > 0$.

int **gsl_sf_bessel_Kn_scaled_array** (int *nmin*, int *nmax*, double *x*, double *result_array[]*)
This routine computes the values of the scaled irregular cylindrical Bessel functions $\exp(x)K_n(x)$ for n from `nmin` to `nmax` inclusive, storing the results in the array `result_array`. The start of the range `nmin` must be positive or zero. The domain of the function is $x > 0$. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

7.5.5 Regular Spherical Bessel Functions

double **gsl_sf_bessel_j0** (double *x*)
int **gsl_sf_bessel_j0_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the regular spherical Bessel function of zeroth order, $j_0(x) = \sin(x)/x$.

double **gsl_sf_bessel_j1** (double *x*)
int **gsl_sf_bessel_j1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the regular spherical Bessel function of first order, $j_1(x) = (\sin(x)/x - \cos(x))/x$.

double **gsl_sf_bessel_j2** (double *x*)
int **gsl_sf_bessel_j2_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the regular spherical Bessel function of second order, $j_2(x) = ((3/x^2 - 1)\sin(x) - 3\cos(x)/x)/x$.

double **gsl_sf_bessel_jl** (int *l*, double *x*)
int **gsl_sf_bessel_jl_e** (int *l*, double *x*, *gsl_sf_result* * *result*)
These routines compute the regular spherical Bessel function of order `l`, $j_l(x)$, for $l \geq 0$ and $x \geq 0$.

int **gsl_sf_bessel_jl_array** (int *lmax*, double *x*, double *result_array[]*)
This routine computes the values of the regular spherical Bessel functions $j_l(x)$ for l from 0 to `lmax` inclusive for $lmax \geq 0$ and $x \geq 0$, storing the results in the array `result_array`. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

int **gsl_sf_bessel_jl_steed_array** (int *lmax*, double *x*, double * *result_array*)
This routine uses Steed's method to compute the values of the regular spherical Bessel functions $j_l(x)$ for l from 0 to `lmax` inclusive for $lmax \geq 0$ and $x \geq 0$, storing the results in the array `result_array`. The Steed/Barnett algorithm is described in Comp. Phys. Comm. 21, 297 (1981). Steed's method is more stable than the recurrence used in the other functions but is also slower.

7.5.6 Irregular Spherical Bessel Functions

double **gsl_sf_bessel_y0** (double *x*)
int **gsl_sf_bessel_y0_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular spherical Bessel function of zeroth order, $y_0(x) = -\cos(x)/x$.

double **gsl_sf_bessel_y1** (double *x*)
int **gsl_sf_bessel_y1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular spherical Bessel function of first order, $y_1(x) = -(\cos(x)/x + \sin(x))/x$.

double **gsl_sf_bessel_y2** (double *x*)
int **gsl_sf_bessel_y2_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular spherical Bessel function of second order, $y_2(x) = (-3/x^3 + 1/x)\cos(x) - (3/x^2)\sin(x)$.

double **gsl_sf_bessel_yl** (int *l*, double *x*)
int **gsl_sf_bessel_yl_e** (int *l*, double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular spherical Bessel function of order `l`, $y_l(x)$, for $l \geq 0$.

int **gsl_sf_bessel_yl_array** (int *lmax*, double *x*, double *result_array[]*)
This routine computes the values of the irregular spherical Bessel functions $y_l(x)$ for l from 0 to `lmax` inclusive for $lmax \geq 0$, storing the results in the array `result_array`. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

7.5.7 Regular Modified Spherical Bessel Functions

The regular modified spherical Bessel functions $i_l(x)$ are related to the modified Bessel functions of fractional order, $i_l(x) = \sqrt{\pi/(2x)} I_{l+1/2}(x)$

double **gsl_sf_bessel_i0_scaled** (double *x*)
int **gsl_sf_bessel_i0_scaled_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled regular modified spherical Bessel function of zeroth order, $\exp(-|x|) i_0(x)$.

double **gsl_sf_bessel_i1_scaled** (double *x*)
int **gsl_sf_bessel_i1_scaled_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled regular modified spherical Bessel function of first order, $\exp(-|x|) i_1(x)$.

double **gsl_sf_bessel_i2_scaled** (double *x*)
int **gsl_sf_bessel_i2_scaled_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled regular modified spherical Bessel function of second order, $\exp(-|x|) i_2(x)$

double **gsl_sf_bessel_il_scaled** (int *l*, double *x*)
int **gsl_sf_bessel_il_scaled_e** (int *l*, double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled regular modified spherical Bessel function of order `l`, $\exp(-|x|) i_l(x)$

int **gsl_sf_bessel_il_scaled_array** (int *lmax*, double *x*, double *result_array[]*)
This routine computes the values of the scaled regular modified spherical Bessel functions $\exp(-|x|) i_l(x)$ for l from 0 to `lmax` inclusive for $lmax \geq 0$, storing the results in the array `result_array`. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

7.5.8 Irregular Modified Spherical Bessel Functions

The irregular modified spherical Bessel functions $k_l(x)$ are related to the irregular modified Bessel functions of fractional order, $k_l(x) = \sqrt{\pi/(2x)} K_{l+1/2}(x)$.

double **gsl_sf_bessel_k0_scaled** (double *x*)
int **gsl_sf_bessel_k0_scaled_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled irregular modified spherical Bessel function of zeroth order, $\exp(x) k_0(x)$, for $x > 0$.

double **gsl_sf_bessel_k1_scaled** (double *x*)

int **gsl_sf_bessel_k1_scaled_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled irregular modified spherical Bessel function of first order, $\exp(x)k_1(x)$, for $x > 0$.

double **gsl_sf_bessel_k2_scaled** (double *x*)
int **gsl_sf_bessel_k2_scaled_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled irregular modified spherical Bessel function of second order, $\exp(x)k_2(x)$, for $x > 0$.

double **gsl_sf_bessel_kl_scaled** (int *l*, double *x*)
int **gsl_sf_bessel_kl_scaled_e** (int *l*, double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled irregular modified spherical Bessel function of order `l`, $\exp(x)k_l(x)$, for $x > 0$.

int **gsl_sf_bessel_kl_scaled_array** (int *lmax*, double *x*, double *result_array[]*)
This routine computes the values of the scaled irregular modified spherical Bessel functions $\exp(x)k_l(x)$ for l from 0 to `lmax` inclusive for $lmax \geq 0$ and $x > 0$, storing the results in the array `result_array`. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

7.5.9 Regular Bessel Function—Fractional Order

double **gsl_sf_bessel_Jnu** (double *nu*, double *x*)
int **gsl_sf_bessel_Jnu_e** (double *nu*, double *x*, *gsl_sf_result* * *result*)
These routines compute the regular cylindrical Bessel function of fractional order ν, $J_\nu(x)$.

int **gsl_sf_bessel_sequence_Jnu_e** (double *nu*, *gsl_mode_t* *mode*, size_t *size*, double *v[]*)
This function computes the regular cylindrical Bessel function of fractional order ν, $J_\nu(x)$, evaluated at a series of x values. The array `v` of length `size` contains the x values. They are assumed to be strictly ordered and positive. The array is over-written with the values of $J_\nu(x_i)$.

7.5.10 Irregular Bessel Functions—Fractional Order

double **gsl_sf_bessel_Ynu** (double *nu*, double *x*)
int **gsl_sf_bessel_Ynu_e** (double *nu*, double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular cylindrical Bessel function of fractional order ν, $Y_\nu(x)$.

7.5.11 Regular Modified Bessel Functions—Fractional Order

double **gsl_sf_bessel_Inu** (double *nu*, double *x*)
int **gsl_sf_bessel_Inu_e** (double *nu*, double *x*, *gsl_sf_result* * *result*)
These routines compute the regular modified Bessel function of fractional order ν, $I_\nu(x)$ for $x > 0$, $\nu > 0$.

double **gsl_sf_bessel_Inu_scaled** (double *nu*, double *x*)
int **gsl_sf_bessel_Inu_scaled_e** (double *nu*, double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled regular modified Bessel function of fractional order ν, $\exp(-|x|)I_\nu(x)$ for $x > 0$, $\nu > 0$.

7.5.12 Irregular Modified Bessel Functions—Fractional Order

double **gsl_sf_bessel_Knu** (double *nu*, double *x*)
int **gsl_sf_bessel_Knu_e** (double *nu*, double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular modified Bessel function of fractional order ν, $K_\nu(x)$ for $x > 0$, $\nu > 0$.

double **gsl_sf_bessel_lnKnu** (double *nu*, double *x*)
int **gsl_sf_bessel_lnKnu_e** (double *nu*, double *x*, *gsl_sf_result* * *result*)
These routines compute the logarithm of the irregular modified Bessel function of fractional order ν, $\ln(K_\nu(x))$ for $x > 0$, $\nu > 0$.

double **gsl_sf_bessel_Knu_scaled** (double *nu*, double *x*)
int **gsl_sf_bessel_Knu_scaled_e** (double *nu*, double *x*, *gsl_sf_result* * *result*)
These routines compute the scaled irregular modified Bessel function of fractional order ν, $\exp(+|x|)K_\nu(x)$ for $x > 0$, $\nu > 0$.

7.5.13 Zeros of Regular Bessel Functions

double **gsl_sf_bessel_zero_J0** (unsigned int *s*)
int **gsl_sf_bessel_zero_J0_e** (unsigned int *s*, *gsl_sf_result* * *result*)
These routines compute the location of the `s`-th positive zero of the Bessel function $J_0(x)$.

double **gsl_sf_bessel_zero_J1** (unsigned int *s*)
int **gsl_sf_bessel_zero_J1_e** (unsigned int *s*, *gsl_sf_result* * *result*)
These routines compute the location of the `s`-th positive zero of the Bessel function $J_1(x)$.

double **gsl_sf_bessel_zero_Jnu** (double *nu*, unsigned int *s*)
int **gsl_sf_bessel_zero_Jnu_e** (double *nu*, unsigned int *s*, *gsl_sf_result* * *result*)
These routines compute the location of the `s`-th positive zero of the Bessel function $J_\nu(x)$. The current implementation does not support negative values of `nu`.

7.6 Clausen Functions

The Clausen function is defined by the following integral,

$$Cl_2(x) = -\int_0^x dt \log\left(2\sin\left(t/2\right)\right)$$

It is related to the *dilogarithm* by $Cl_2(\theta) = \Im Li_2(\exp(i\theta))$. The Clausen functions are declared in the header file `gsl_sf_clausen.h`.

double **gsl_sf_clausen** (double *x*)
int **gsl_sf_clausen_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the Clausen integral $Cl_2(x)$.

7.7 Coulomb Functions

The prototypes of the Coulomb functions are declared in the header file `gsl_sf_coulomb.h`. Both bound state and scattering solutions are available.

7.7.1 Normalized Hydrogenic Bound States

double **gsl_sf_hydrogenicR_1** (double *Z*, double *r*)
int **gsl_sf_hydrogenicR_1_e** (double *Z*, double *r*, *gsl_sf_result* * *result*)
These routines compute the lowest-order normalized hydrogenic bound state radial wavefunction $R_1 := 2Z\sqrt{Z}\exp(-Zr)$.

double **gsl_sf_hydrogenicR** (int *n*, int *l*, double *Z*, double *r*)

int **gsl_sf_hydrogenicR_e** (int *n*, int *l*, double *Z*, double *r*, *gsl_sf_result* * *result*)

These routines compute the n-th normalized hydrogenic bound state radial wavefunction,

$$R_n := \frac{2Z^{3/2}}{n^2}\left(\frac{2Zr}{n}\right)^l \sqrt{\frac{(n-l-1)!}{(n+l)!}}\exp(-Zr/n)L^{2l+1}_{n-l-1}(2Zr/n).$$

where $L^a_b(x)$ is the *generalized Laguerre polynomial*. The normalization is chosen such that the wavefunction ψ is given by $\psi(n,l,r) = R_n Y_{lm}$.

7.7.2 Coulomb Wave Functions

The Coulomb wave functions $F_L(\eta,x)$, $G_L(\eta,x)$ are described in Abramowitz & Stegun, Chapter 14. Because there can be a large dynamic range of values for these functions, overflows are handled gracefully. If an overflow occurs, GSL_EOVRFLW is signalled and exponent(s) are returned through the modifiable parameters exp_F, exp_G. The full solution can be reconstructed from the following relations,

$$F_L(\eta,x) = fc[k_L] * \exp(exp_F)$$
$$G_L(\eta,x) = gc[k_L] * \exp(exp_G)$$

$$F'_L(\eta,x) = fcp[k_L] * \exp(exp_F)$$
$$G'_L(\eta,x) = gcp[k_L] * \exp(exp_G)$$

int **gsl_sf_coulomb_wave_FG_e** (double *eta*, double *x*, double *L_F*, int *k*, *gsl_sf_result* * *F*, *gsl_sf_result* * *Fp*, *gsl_sf_result* * *G*, *gsl_sf_result* * *Gp*, double * *exp_F*, double * *exp_G*)

This function computes the Coulomb wave functions $F_L(\eta,x)$, $G_{L-k}(\eta,x)$ and their derivatives $F'_L(\eta,x)$, $G'_{L-k}(\eta,x)$ with respect to x. The parameters are restricted to $L, L-k > -1/2$, $x > 0$ and integer k. Note that L itself is not restricted to being an integer. The results are stored in the parameters F, G for the function values and Fp, Gp for the derivative values. If an overflow occurs, GSL_EOVRFLW is returned and scaling exponents are stored in the modifiable parameters exp_F, exp_G.

int **gsl_sf_coulomb_wave_F_array** (double *L_min*, int *kmax*, double *eta*, double *x*, double *fc_array[]*, double * *F_exponent*)

This function computes the Coulomb wave function $F_L(\eta,x)$ for $L = Lmin \ldots Lmin + kmax$, storing the results in fc_array. In the case of overflow the exponent is stored in F_exponent.

int **gsl_sf_coulomb_wave_FG_array** (double *L_min*, int *kmax*, double *eta*, double *x*, double *fc_array[]*, double *gc_array[]*, double * *F_exponent*, double * *G_exponent*)

This function computes the functions $F_L(\eta,x)$, $G_L(\eta,x)$ for $L = Lmin \ldots Lmin + kmax$ storing the results in fc_array and gc_array. In the case of overflow the exponents are stored in F_exponent and G_exponent.

int **gsl_sf_coulomb_wave_FGp_array** (double *L_min*, int *kmax*, double *eta*, double *x*, double *fc_array[]*, double *fcp_array[]*, double *gc_array[]*, double *gcp_array[]*, double * *F_exponent*, double * *G_exponent*)

This function computes the functions $F_L(\eta,x)$, $G_L(\eta,x)$ and their derivatives $F'_L(\eta,x)$, $G'_L(\eta,x)$ for $L = Lmin \ldots Lmin + kmax$ storing the results in fc_array, gc_array, fcp_array and gcp_array. In the case of overflow the exponents are stored in F_exponent and G_exponent.

int **gsl_sf_coulomb_wave_sphF_array** (double *L_min*, int *kmax*, double *eta*, double *x*, double *fc_array[]*, double *F_exponent[]*)

This function computes the Coulomb wave function divided by the argument $F_L(\eta,x)/x$ for $L = Lmin \ldots Lmin + kmax$, storing the results in fc_array. In the case of overflow the exponent is stored in F_exponent. This function reduces to spherical Bessel functions in the limit $\eta \to 0$.

7.7.3 Coulomb Wave Function Normalization Constant

The Coulomb wave function normalization constant is defined in Abramowitz 14.1.7.

int **gsl_sf_coulomb_CL_e** (double *L*, double *eta*, *gsl_sf_result* * *result*)
This function computes the Coulomb wave function normalization constant $C_L(\eta)$ for $L > -1$.

int **gsl_sf_coulomb_CL_array** (double *Lmin*, int *kmax*, double *eta*, double *cl[]*)
This function computes the Coulomb wave function normalization constant $C_L(\eta)$ for $L = Lmin \ldots Lmin + kmax$, $Lmin > -1$.

7.8 Coupling Coefficients

The Wigner 3-j, 6-j and 9-j symbols give the coupling coefficients for combined angular momentum vectors. Since the arguments of the standard coupling coefficient functions are integer or half-integer, the arguments of the following functions are, by convention, integers equal to twice the actual spin value. For information on the 3-j coefficients see Abramowitz & Stegun, Section 27.9. The functions described in this section are declared in the header file `gsl_sf_coupling.h`.

7.8.1 3-j Symbols

double **gsl_sf_coupling_3j** (int *two_ja*, int *two_jb*, int *two_jc*, int *two_ma*, int *two_mb*, int *two_mc*)
int **gsl_sf_coupling_3j_e** (int *two_ja*, int *two_jb*, int *two_jc*, int *two_ma*, int *two_mb*, int *two_mc*, *gsl_sf_result* * *result*)
These routines compute the Wigner 3-j coefficient,

$$\begin{pmatrix} ja & jb & jc \\ ma & mb & mc \end{pmatrix}$$

where the arguments are given in half-integer units, ja = `two_ja`/2, ma = `two_ma`/2, etc.

7.8.2 6-j Symbols

double **gsl_sf_coupling_6j** (int *two_ja*, int *two_jb*, int *two_jc*, int *two_jd*, int *two_je*, int *two_jf*)
int **gsl_sf_coupling_6j_e** (int *two_ja*, int *two_jb*, int *two_jc*, int *two_jd*, int *two_je*, int *two_jf*, *gsl_sf_result* * *result*)
These routines compute the Wigner 6-j coefficient,

$$\begin{Bmatrix} ja & jb & jc \\ jd & je & jf \end{Bmatrix}$$

where the arguments are given in half-integer units, ja = `two_ja`/2, ma = `two_ma`/2, etc.

7.8.3 9-j Symbols

double **gsl_sf_coupling_9j** (int *two_ja*, int *two_jb*, int *two_jc*, int *two_jd*, int *two_je*, int *two_jf*, int *two_jg*, int *two_jh*, int *two_ji*)
int **gsl_sf_coupling_9j_e** (int *two_ja*, int *two_jb*, int *two_jc*, int *two_jd*, int *two_je*, int *two_jf*, int *two_jg*, int *two_jh*, int *two_ji*, *gsl_sf_result* * *result*)
These routines compute the Wigner 9-j coefficient,

$$\begin{Bmatrix} ja & jb & jc \\ jd & je & jf \\ jg & jh & ji \end{Bmatrix}$$

where the arguments are given in half-integer units, ja = `two_ja`/2, ma = `two_ma`/2, etc.

7.9 Dawson Function

The Dawson integral is defined by

$$\exp(-x^2) \int_0^x dt \exp(t^2)$$

A table of Dawson's integral can be found in Abramowitz & Stegun, Table 7.5. The Dawson functions are declared in the header file `gsl_sf_dawson.h`.

double **gsl_sf_dawson** (double *x*)
int **gsl_sf_dawson_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the value of Dawson's integral for `x`.

7.10 Debye Functions

The Debye functions $D_n(x)$ are defined by the following integral,

$$D_n(x) = \frac{n}{x^n} \int_0^x dt \frac{t^n}{e^t - 1}$$

For further information see Abramowitz & Stegun, Section 27.1. The Debye functions are declared in the header file `gsl_sf_debye.h`.

double **gsl_sf_debye_1** (double *x*)
int **gsl_sf_debye_1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the first-order Debye function $D_1(x)$.

double **gsl_sf_debye_2** (double *x*)
int **gsl_sf_debye_2_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the second-order Debye function $D_2(x)$.

double **gsl_sf_debye_3** (double *x*)
int **gsl_sf_debye_3_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the third-order Debye function $D_3(x)$.

double **gsl_sf_debye_4** (double *x*)
int **gsl_sf_debye_4_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the fourth-order Debye function $D_4(x)$.

double **gsl_sf_debye_5** (double *x*)
int **gsl_sf_debye_5_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the fifth-order Debye function $D_5(x)$.

double **gsl_sf_debye_6** (double *x*)
int **gsl_sf_debye_6_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the sixth-order Debye function $D_6(x)$.

7.11 Dilogarithm

The dilogarithm is defined as

$$Li_2(z) = -\int_0^z ds \frac{\log(1-s)}{s}$$

The functions described in this section are declared in the header file `gsl_sf_dilog.h`.

7.11.1 Real Argument

double **gsl_sf_dilog** (double *x*)
int **gsl_sf_dilog_e** (double *x*, *gsl_sf_result* * *result*)

These routines compute the dilogarithm for a real argument. In Lewin's notation this is $Li_2(x)$, the real part of the dilogarithm of a real x. It is defined by the integral representation

$$Li_2(x) = -\Re \int_0^x ds \log(1-s)/s$$

Note that $\Im(Li_2(x)) = 0$ for $x \leq 1$, and $-\pi \log(x)$ for $x > 1$.

Note that Abramowitz & Stegun refer to the Spence integral $S(x) = Li_2(1-x)$ as the dilogarithm rather than $Li_2(x)$.

7.11.2 Complex Argument

int **gsl_sf_complex_dilog_e** (double *r*, double *theta*, *gsl_sf_result* * *result_re*, *gsl_sf_result* * *result_im*)

This function computes the full complex-valued dilogarithm for the complex argument $z = r \exp(i\theta)$. The real and imaginary parts of the result are returned in `result_re`, `result_im`.

7.12 Elementary Operations

The following functions allow for the propagation of errors when combining quantities by multiplication. The functions are declared in the header file `gsl_sf_elementary.h`.

double **gsl_sf_multiply** (double *x*, double *y*)
int **gsl_sf_multiply_e** (double *x*, double *y*, *gsl_sf_result* * *result*)

This function multiplies `x` and `y` storing the product and its associated error in `result`.

int **gsl_sf_multiply_err_e** (double *x*, double *dx*, double *y*, double *dy*, *gsl_sf_result* * *result*)

This function multiplies `x` and `y` with associated absolute errors `dx` and `dy`. The product $xy \pm xy\sqrt{(dx/x)^2 + (dy/y)^2}$ is stored in `result`.

7.13 Elliptic Integrals

The functions described in this section are declared in the header file `gsl_sf_ellint.h`. Further information about the elliptic integrals can be found in Abramowitz & Stegun, Chapter 17.

7.13.1 Definition of Legendre Forms

The Legendre forms of elliptic integrals $F(\phi, k)$, $E(\phi, k)$ and $\Pi(\phi, k, n)$ are defined by,

$$F(\phi,k) = \int_0^\phi dt \frac{1}{\sqrt{(1-k^2\sin^2(t))}}$$
$$E(\phi,k) = \int_0^\phi dt \sqrt{(1-k^2\sin^2(t))}$$
$$\Pi(\phi,k,n) = \int_0^\phi dt \frac{1}{(1+n\sin^2(t))\sqrt{1-k^2\sin^2(t)}}$$

The complete Legendre forms are denoted by $K(k) = F(\pi/2, k)$ and $E(k) = E(\pi/2, k)$.

The notation used here is based on Carlson, "Numerische Mathematik" 33 (1979) 1 and differs slightly from that used by Abramowitz & Stegun, where the functions are given in terms of the parameter $m = k^2$ and n is replaced by $-n$.

7.13.2 Definition of Carlson Forms

The Carlson symmetric forms of elliptical integrals $RC(x, y)$, $RD(x, y, z)$, $RF(x, y, z)$ and $RJ(x, y, z, p)$ are defined by,

$$RC(x,y) = 1/2 \int_0^\infty dt (t+x)^{-1/2}(t+y)^{-1}$$
$$RD(x,y,z) = 3/2 \int_0^\infty dt (t+x)^{-1/2}(t+y)^{-1/2}(t+z)^{-3/2}$$
$$RF(x,y,z) = 1/2 \int_0^\infty dt (t+x)^{-1/2}(t+y)^{-1/2}(t+z)^{-1/2}$$
$$RJ(x,y,z,p) = 3/2 \int_0^\infty dt (t+x)^{-1/2}(t+y)^{-1/2}(t+z)^{-1/2}(t+p)^{-1}$$

7.13.3 Legendre Form of Complete Elliptic Integrals

double **gsl_sf_ellint_Kcomp** (double *k*, *gsl_mode_t mode*)
int **gsl_sf_ellint_Kcomp_e** (double *k*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute the complete elliptic integral $K(k)$ to the accuracy specified by the mode variable `mode`. Note that Abramowitz & Stegun define this function in terms of the parameter $m = k^2$.

double **gsl_sf_ellint_Ecomp** (double *k*, *gsl_mode_t mode*)
int **gsl_sf_ellint_Ecomp_e** (double *k*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute the complete elliptic integral $E(k)$ to the accuracy specified by the mode variable `mode`. Note that Abramowitz & Stegun define this function in terms of the parameter $m = k^2$.

double **gsl_sf_ellint_Pcomp** (double *k*, double *n*, *gsl_mode_t mode*)
int **gsl_sf_ellint_Pcomp_e** (double *k*, double *n*, *gsl_mode_t mode*, *gsl_sf_result * result*)
These routines compute the complete elliptic integral $\Pi(k, n)$ to the accuracy specified by the mode variable `mode`. Note that Abramowitz & Stegun define this function in terms of the parameters $m = k^2$ and $\sin^2(\alpha) = k^2$, with the change of sign $n \to -n$.

7.13.4 Legendre Form of Incomplete Elliptic Integrals

double **gsl_sf_ellint_F** (double *phi*, double *k*, *gsl_mode_t mode*)

int **gsl_sf_ellint_F_e** (double *phi*, double *k*, *gsl_mode_t mode*, *gsl_sf_result* * *result*)
These routines compute the incomplete elliptic integral $F(\phi, k)$ to the accuracy specified by the mode variable mode. Note that Abramowitz & Stegun define this function in terms of the parameter $m = k^2$.

double **gsl_sf_ellint_E** (double *phi*, double *k*, *gsl_mode_t mode*)
int **gsl_sf_ellint_E_e** (double *phi*, double *k*, *gsl_mode_t mode*, *gsl_sf_result* * *result*)
These routines compute the incomplete elliptic integral $E(\phi, k)$ to the accuracy specified by the mode variable mode. Note that Abramowitz & Stegun define this function in terms of the parameter $m = k^2$.

double **gsl_sf_ellint_P** (double *phi*, double *k*, double *n*, *gsl_mode_t mode*)
int **gsl_sf_ellint_P_e** (double *phi*, double *k*, double *n*, *gsl_mode_t mode*, *gsl_sf_result* * *result*)
These routines compute the incomplete elliptic integral $\Pi(\phi, k, n)$ to the accuracy specified by the mode variable mode. Note that Abramowitz & Stegun define this function in terms of the parameters $m = k^2$ and $\sin^2(\alpha) = k^2$, with the change of sign $n \to -n$.

double **gsl_sf_ellint_D** (double *phi*, double *k*, *gsl_mode_t mode*)
int **gsl_sf_ellint_D_e** (double *phi*, double *k*, *gsl_mode_t mode*, *gsl_sf_result* * *result*)
These functions compute the incomplete elliptic integral $D(\phi, k)$ which is defined through the Carlson form $RD(x, y, z)$ by the following relation,

$$D(\phi, k) = \frac{1}{3}(\sin\phi)^3 RD(1 - \sin^2(\phi), 1 - k^2 \sin^2(\phi), 1)$$

7.13.5 Carlson Forms

double **gsl_sf_ellint_RC** (double *x*, double *y*, *gsl_mode_t mode*)
int **gsl_sf_ellint_RC_e** (double *x*, double *y*, *gsl_mode_t mode*, *gsl_sf_result* * *result*)
These routines compute the incomplete elliptic integral $RC(x, y)$ to the accuracy specified by the mode variable mode.

double **gsl_sf_ellint_RD** (double *x*, double *y*, double *z*, *gsl_mode_t mode*)
int **gsl_sf_ellint_RD_e** (double *x*, double *y*, double *z*, *gsl_mode_t mode*, *gsl_sf_result* * *result*)
These routines compute the incomplete elliptic integral $RD(x, y, z)$ to the accuracy specified by the mode variable mode.

double **gsl_sf_ellint_RF** (double *x*, double *y*, double *z*, *gsl_mode_t mode*)
int **gsl_sf_ellint_RF_e** (double *x*, double *y*, double *z*, *gsl_mode_t mode*, *gsl_sf_result* * *result*)
These routines compute the incomplete elliptic integral $RF(x, y, z)$ to the accuracy specified by the mode variable mode.

double **gsl_sf_ellint_RJ** (double *x*, double *y*, double *z*, double *p*, *gsl_mode_t mode*)
int **gsl_sf_ellint_RJ_e** (double *x*, double *y*, double *z*, double *p*, *gsl_mode_t mode*, *gsl_sf_result* * *result*)
These routines compute the incomplete elliptic integral $RJ(x, y, z, p)$ to the accuracy specified by the mode variable mode.

7.14 Elliptic Functions (Jacobi)

The Jacobian Elliptic functions are defined in Abramowitz & Stegun, Chapter 16. The functions are declared in the header file `gsl_sf_elljac.h`.

int **gsl_sf_elljac_e** (double *u*, double *m*, double * *sn*, double * *cn*, double * *dn*)
This function computes the Jacobian elliptic functions $sn(u|m)$, $cn(u|m)$, $dn(u|m)$ by descending Landen transformations.

7.15 Error Functions

The error function is described in Abramowitz & Stegun, Chapter 7. The functions in this section are declared in the header file `gsl_sf_erf.h`.

7.15.1 Error Function

double **gsl_sf_erf** (double *x*)
int **gsl_sf_erf_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the error function $\mathrm{erf}(x)$, where $\mathrm{erf}(x) = (2/\sqrt{\pi}) \int_0^x dt \exp(-t^2)$.

7.15.2 Complementary Error Function

double **gsl_sf_erfc** (double *x*)
int **gsl_sf_erfc_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the complementary error function $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) = (2/\sqrt{\pi}) \int_x^\infty \exp(-t^2)$

7.15.3 Log Complementary Error Function

double **gsl_sf_log_erfc** (double *x*)
int **gsl_sf_log_erfc_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the logarithm of the complementary error function $\log(\mathrm{erfc}(x))$.

7.15.4 Probability functions

The probability functions for the Normal or Gaussian distribution are described in Abramowitz & Stegun, Section 26.2.

double **gsl_sf_erf_Z** (double *x*)
int **gsl_sf_erf_Z_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the Gaussian probability density function $Z(x) = (1/\sqrt{2\pi}) \exp(-x^2/2)$

double **gsl_sf_erf_Q** (double *x*)
int **gsl_sf_erf_Q_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the upper tail of the Gaussian probability function $Q(x) = (1/\sqrt{2\pi}) \int_x^\infty dt \exp(-t^2/2)$

The *hazard function* for the normal distribution, also known as the inverse Mills' ratio, is defined as,

$$h(x) = \frac{Z(x)}{Q(x)} = \sqrt{\frac{2}{\pi}} \frac{\exp(-x^2/2)}{\mathrm{erfc}(x/\sqrt{2})}$$

It decreases rapidly as x approaches $-\infty$ and asymptotes to $h(x) \sim x$ as x approaches $+\infty$.

double **gsl_sf_hazard** (double *x*)
int **gsl_sf_hazard_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the hazard function for the normal distribution.

7.16 Exponential Functions

The functions described in this section are declared in the header file `gsl_sf_exp.h`.

7.16.1 Exponential Function

double **gsl_sf_exp** (double *x*)
int **gsl_sf_exp_e** (double *x*, *gsl_sf_result* * *result*)
These routines provide an exponential function $\exp(x)$ using GSL semantics and error checking.

int **gsl_sf_exp_e10_e** (double *x*, *gsl_sf_result_e10* * *result*)
This function computes the exponential $\exp(x)$ using the *gsl_sf_result_e10* type to return a result with extended range. This function may be useful if the value of $\exp(x)$ would overflow the numeric range of `double`.

double **gsl_sf_exp_mult** (double *x*, double *y*)
int **gsl_sf_exp_mult_e** (double *x*, double *y*, *gsl_sf_result* * *result*)
These routines exponentiate `x` and multiply by the factor `y` to return the product $y \exp(x)$.

int **gsl_sf_exp_mult_e10_e** (const double *x*, const double *y*, *gsl_sf_result_e10* * *result*)
This function computes the product $y \exp(x)$ using the *gsl_sf_result_e10* type to return a result with extended numeric range.

7.16.2 Relative Exponential Functions

double **gsl_sf_expm1** (double *x*)
int **gsl_sf_expm1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the quantity $\exp(x) - 1$ using an algorithm that is accurate for small x.

double **gsl_sf_exprel** (double *x*)
int **gsl_sf_exprel_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the quantity $(\exp(x) - 1)/x$ using an algorithm that is accurate for small `x`. For small `x` the algorithm is based on the expansion $(\exp(x) - 1)/x = 1 + x/2 + x^2/(2*3) + x^3/(2*3*4) + \ldots$.

double **gsl_sf_exprel_2** (double *x*)
int **gsl_sf_exprel_2_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the quantity $2(\exp(x) - 1 - x)/x^2$ using an algorithm that is accurate for small `x`. For small `x` the algorithm is based on the expansion $2(\exp(x)-1-x)/x^2 = 1+x/3+x^2/(3*4)+x^3/(3*4*5)+\ldots$.

double **gsl_sf_exprel_n** (int *n*, double *x*)
int **gsl_sf_exprel_n_e** (int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the N-relative exponential, which is the `n`-th generalization of the functions *gsl_sf_exprel()* and *gsl_sf_exprel_2()*. The N-relative exponential is given by,

$$\begin{aligned}\mathrm{exprel}_N(x) &= N!/x^N \left(\exp(x) - \sum_{k=0}^{N-1} x^k/k! \right) \\ &= 1 + x/(N+1) + x^2/((N+1)(N+2)) + \ldots \\ &= {}_1F_1(1, 1+N, x)\end{aligned}$$

7.16.3 Exponentiation With Error Estimate

int **gsl_sf_exp_err_e** (double *x*, double *dx*, *gsl_sf_result* * *result*)
This function exponentiates `x` with an associated absolute error `dx`.

int **gsl_sf_exp_err_e10_e** (double *x*, double *dx*, *gsl_sf_result_e10* * *result*)
This function exponentiates a quantity `x` with an associated absolute error `dx` using the *gsl_sf_result_e10* type to return a result with extended range.

int **gsl_sf_exp_mult_err_e** (double *x*, double *dx*, double *y*, double *dy*, *gsl_sf_result* * *result*)
This routine computes the product $y \exp(x)$ for the quantities `x`, `y` with associated absolute errors `dx`, `dy`.

int **gsl_sf_exp_mult_err_e10_e** (double *x*, double *dx*, double *y*, double *dy*, *gsl_sf_result_e10* * *result*)
This routine computes the product $y \exp(x)$ for the quantities `x`, `y` with associated absolute errors `dx`, `dy` using the *gsl_sf_result_e10* type to return a result with extended range.

7.17 Exponential Integrals

Information on the exponential integrals can be found in Abramowitz & Stegun, Chapter 5. These functions are declared in the header file `gsl_sf_expint.h`.

7.17.1 Exponential Integral

double **gsl_sf_expint_E1** (double *x*)
int **gsl_sf_expint_E1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the exponential integral $E_1(x)$,

$$E_1(x) := \Re \int_1^\infty dt \exp(-xt)/t.$$

double **gsl_sf_expint_E2** (double *x*)
int **gsl_sf_expint_E2_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the second-order exponential integral $E_2(x)$,

$$E_2(x) := \Re \int_1^\infty dt \exp(-xt)/t^2$$

double **gsl_sf_expint_En** (int *n*, double *x*)
int **gsl_sf_expint_En_e** (int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the exponential integral $E_n(x)$ of order `n`,

$$E_n(x) := \Re \int_1^\infty dt \exp(-xt)/t^n.$$

7.17.2 Ei(x)

double **gsl_sf_expint_Ei** (double *x*)
int **gsl_sf_expint_Ei_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the exponential integral $Ei(x)$,

$$\mathrm{Ei}(x) = -PV\left(\int_{-x}^\infty dt \exp(-t)/t\right)$$

where PV denotes the principal value of the integral.

7.17.3 Hyperbolic Integrals

double **gsl_sf_Shi** (double *x*)

int **gsl_sf_Shi_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the integral

$$\mathrm{Shi}(x) = \int_0^x dt \sinh(t)/t$$

double **gsl_sf_Chi** (double *x*)
int **gsl_sf_Chi_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the integral

$$\mathrm{Chi}(x) := \Re\left[\gamma_E + \log(x) + \int_0^x dt(\cosh(t) - 1)/t\right]$$

where γ_E is the Euler constant (available as the macro `M_EULER`).

7.17.4 Ei_3(x)

double **gsl_sf_expint_3** (double *x*)
int **gsl_sf_expint_3_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the third-order exponential integral

$$\mathrm{Ei}_3(x) = \int_0^x dt \exp(-t^3)$$

for $x \geq 0$.

7.17.5 Trigonometric Integrals

double **gsl_sf_Si** (const double *x*)
int **gsl_sf_Si_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the Sine integral

$$\mathrm{Si}(x) = \int_0^x dt \sin(t)/t$$

double **gsl_sf_Ci** (const double *x*)
int **gsl_sf_Ci_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the Cosine integral

$$\mathrm{Ci}(x) = -\int_x^\infty dt \cos(t)/t$$

for $x > 0$

7.17.6 Arctangent Integral

double **gsl_sf_atanint** (double *x*)
int **gsl_sf_atanint_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the Arctangent integral, which is defined as

$$\mathrm{AtanInt}(x) = \int_0^x dt \arctan(t)/t$$

7.18 Fermi-Dirac Function

The functions described in this section are declared in the header file `gsl_sf_fermi_dirac.h`.

7.18.1 Complete Fermi-Dirac Integrals

The complete Fermi-Dirac integral $F_j(x)$ is given by,

$$F_j(x) := \frac{1}{\Gamma(j+1)} \int_0^\infty dt \frac{t^j}{(\exp(t-x)+1)}$$

Note that the Fermi-Dirac integral is sometimes defined without the normalisation factor in other texts.

double **gsl_sf_fermi_dirac_m1** (double *x*)
int **gsl_sf_fermi_dirac_m1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the complete Fermi-Dirac integral with an index of -1. This integral is given by $F_{-1}(x) = e^x/(1+e^x)$.

double **gsl_sf_fermi_dirac_0** (double *x*)
int **gsl_sf_fermi_dirac_0_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the complete Fermi-Dirac integral with an index of 0. This integral is given by $F_0(x) = \ln(1+e^x)$.

double **gsl_sf_fermi_dirac_1** (double *x*)
int **gsl_sf_fermi_dirac_1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the complete Fermi-Dirac integral with an index of 1, $F_1(x) = \int_0^\infty dt (t/(\exp(t-x)+1))$.

double **gsl_sf_fermi_dirac_2** (double *x*)
int **gsl_sf_fermi_dirac_2_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the complete Fermi-Dirac integral with an index of 2, $F_2(x) = (1/2) \int_0^\infty dt (t^2/(\exp(t-x)+1))$.

double **gsl_sf_fermi_dirac_int** (int *j*, double *x*)
int **gsl_sf_fermi_dirac_int_e** (int *j*, double *x*, *gsl_sf_result* * *result*)
These routines compute the complete Fermi-Dirac integral with an integer index of j, $F_j(x) = (1/\Gamma(j+1)) \int_0^\infty dt (t^j/(\exp(t-x)+1))$.

double **gsl_sf_fermi_dirac_mhalf** (double *x*)
int **gsl_sf_fermi_dirac_mhalf_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the complete Fermi-Dirac integral $F_{-1/2}(x)$.

double **gsl_sf_fermi_dirac_half** (double *x*)
int **gsl_sf_fermi_dirac_half_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the complete Fermi-Dirac integral $F_{1/2}(x)$.

double **gsl_sf_fermi_dirac_3half** (double *x*)
int **gsl_sf_fermi_dirac_3half_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the complete Fermi-Dirac integral $F_{3/2}(x)$.

7.18.2 Incomplete Fermi-Dirac Integrals

The incomplete Fermi-Dirac integral $F_j(x,b)$ is given by,

$$F_j(x,b) := \frac{1}{\Gamma(j+1)} \int_b^\infty dt \frac{t^j}{(\exp(t-x)+1)}$$

double **gsl_sf_fermi_dirac_inc_0** (double *x*, double *b*)
int **gsl_sf_fermi_dirac_inc_0_e** (double *x*, double *b*, *gsl_sf_result* * *result*)
These routines compute the incomplete Fermi-Dirac integral with an index of zero, $F_0(x,b) = \ln(1 + e^{b-x}) - (b - x)$

7.19 Gamma and Beta Functions

The following routines compute the gamma and beta functions in their full and incomplete forms, as well as various kinds of factorials. The functions described in this section are declared in the header file `gsl_sf_gamma.h`.

7.19.1 Gamma Functions

The Gamma function is defined by the following integral,

$$\Gamma(x) = \int_0^\infty dt t^{x-1} \exp(-t)$$

It is related to the factorial function by $\Gamma(n) = (n-1)!$ for positive integer n. Further information on the Gamma function can be found in Abramowitz & Stegun, Chapter 6.

double **gsl_sf_gamma** (double *x*)
int **gsl_sf_gamma_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the Gamma function $\Gamma(x)$, subject to x not being a negative integer or zero. The function is computed using the real Lanczos method. The maximum value of x such that $\Gamma(x)$ is not considered an overflow is given by the macro `GSL_SF_GAMMA_XMAX` and is 171.0.

double **gsl_sf_lngamma** (double *x*)
int **gsl_sf_lngamma_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the logarithm of the Gamma function, $\log(\Gamma(x))$, subject to x not being a negative integer or zero. For $x < 0$ the real part of $\log(\Gamma(x))$ is returned, which is equivalent to $\log(|\Gamma(x)|)$. The function is computed using the real Lanczos method.

int **gsl_sf_lngamma_sgn_e** (double *x*, *gsl_sf_result* * *result_lg*, double * *sgn*)
This routine computes the sign of the gamma function and the logarithm of its magnitude, subject to x not being a negative integer or zero. The function is computed using the real Lanczos method. The value of the gamma function and its error can be reconstructed using the relation $\Gamma(x) = sgn * \exp(result\_lg)$, taking into account the two components of `result_lg`.

double **gsl_sf_gammastar** (double *x*)
int **gsl_sf_gammastar_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the regulated Gamma Function $\Gamma^*(x)$ for $x > 0$. The regulated gamma function is given by,

$$\begin{aligned}\Gamma^*(x) &= \Gamma(x)/(\sqrt{2\pi}x^{(x-1/2)}\exp(-x)) \\ &= \left(1 + \frac{1}{12x} + ...\right) \quad \text{for } x \to \infty\end{aligned}$$

and is a useful suggestion of Temme.

double **gsl_sf_gammainv** (double *x*)
int **gsl_sf_gammainv_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the reciprocal of the gamma function, $1/\Gamma(x)$ using the real Lanczos method.

int **gsl_sf_lngamma_complex_e** (double *zr*, double *zi*, *gsl_sf_result* * *lnr*, *gsl_sf_result* * *arg*)
This routine computes $\log(\Gamma(z))$ for complex $z = z_r + iz_i$ and z not a negative integer or zero, using the

complex Lanczos method. The returned parameters are $lnr = \log|\Gamma(z)|$ and $arg = \arg(\Gamma(z))$ in $(-\pi, \pi]$. Note that the phase part (`arg`) is not well-determined when $|z|$ is very large, due to inevitable roundoff in restricting to $(-\pi, \pi]$. This will result in a `GSL_ELOSS` error when it occurs. The absolute value part (`lnr`), however, never suffers from loss of precision.

7.19.2 Factorials

Although factorials can be computed from the Gamma function, using the relation $n! = \Gamma(n+1)$ for non-negative integer n, it is usually more efficient to call the functions in this section, particularly for small values of n, whose factorial values are maintained in hardcoded tables.

double **gsl_sf_fact** (unsigned int *n*)
int **gsl_sf_fact_e** (unsigned int *n*, *gsl_sf_result* * *result*)
These routines compute the factorial $n!$. The factorial is related to the Gamma function by $n! = \Gamma(n+1)$. The maximum value of n such that $n!$ is not considered an overflow is given by the macro `GSL_SF_FACT_NMAX` and is 170.

double **gsl_sf_doublefact** (unsigned int *n*)
int **gsl_sf_doublefact_e** (unsigned int *n*, *gsl_sf_result* * *result*)
These routines compute the double factorial $n!! = n(n-2)(n-4)\ldots$. The maximum value of n such that $n!!$ is not considered an overflow is given by the macro `GSL_SF_DOUBLEFACT_NMAX` and is 297.

double **gsl_sf_lnfact** (unsigned int *n*)
int **gsl_sf_lnfact_e** (unsigned int *n*, *gsl_sf_result* * *result*)
These routines compute the logarithm of the factorial of `n`, $\log(n!)$. The algorithm is faster than computing $\ln(\Gamma(n+1))$ via *gsl_sf_lngamma()* for $n < 170$, but defers for larger `n`.

double **gsl_sf_lndoublefact** (unsigned int *n*)
int **gsl_sf_lndoublefact_e** (unsigned int *n*, *gsl_sf_result* * *result*)
These routines compute the logarithm of the double factorial of `n`, $\log(n!!)$.

double **gsl_sf_choose** (unsigned int *n*, unsigned int *m*)
int **gsl_sf_choose_e** (unsigned int *n*, unsigned int *m*, *gsl_sf_result* * *result*)
These routines compute the combinatorial factor `n choose m` $= n!/(m!(n-m)!)$

double **gsl_sf_lnchoose** (unsigned int *n*, unsigned int *m*)
int **gsl_sf_lnchoose_e** (unsigned int *n*, unsigned int *m*, *gsl_sf_result* * *result*)
These routines compute the logarithm of `n choose m`. This is equivalent to the sum $\log(n!) - \log(m!) - \log((n-m)!)$.

double **gsl_sf_taylorcoeff** (int *n*, double *x*)
int **gsl_sf_taylorcoeff_e** (int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the Taylor coefficient $x^n/n!$ for $x \geq 0$, $n \geq 0$

7.19.3 Pochhammer Symbol

double **gsl_sf_poch** (double *a*, double *x*)
int **gsl_sf_poch_e** (double *a*, double *x*, *gsl_sf_result* * *result*)
These routines compute the Pochhammer symbol $(a)_x = \Gamma(a+x)/\Gamma(a)$. The Pochhammer symbol is also known as the Apell symbol and sometimes written as (a, x). When a and $a+x$ are negative integers or zero, the limiting value of the ratio is returned.

double **gsl_sf_lnpoch** (double *a*, double *x*)
int **gsl_sf_lnpoch_e** (double *a*, double *x*, *gsl_sf_result* * *result*)
These routines compute the logarithm of the Pochhammer symbol, $\log((a)_x) = \log(\Gamma(a+x)/\Gamma(a))$.

int **gsl_sf_lnpoch_sgn_e** (double *a*, double *x*, *gsl_sf_result* * *result*, double * *sgn*)
These routines compute the sign of the Pochhammer symbol and the logarithm of its magnitude. The computed parameters are $result = \log(|(a)_x|)$ with a corresponding error term, and $sgn = \mathrm{sgn}((a)_x)$ where $(a)_x = \Gamma(a+x)/\Gamma(a)$.

double **gsl_sf_pochrel** (double *a*, double *x*)
int **gsl_sf_pochrel_e** (double *a*, double *x*, *gsl_sf_result* * *result*)
These routines compute the relative Pochhammer symbol $((a)_x - 1)/x$ where $(a)_x = \Gamma(a+x)/\Gamma(a)$.

7.19.4 Incomplete Gamma Functions

double **gsl_sf_gamma_inc** (double *a*, double *x*)
int **gsl_sf_gamma_inc_e** (double *a*, double *x*, *gsl_sf_result* * *result*)
These functions compute the unnormalized incomplete Gamma Function $\Gamma(a,x) = \int_x^\infty dt t^{(a-1)} \exp(-t)$ for a real and $x \geq 0$.

double **gsl_sf_gamma_inc_Q** (double *a*, double *x*)
int **gsl_sf_gamma_inc_Q_e** (double *a*, double *x*, *gsl_sf_result* * *result*)
These routines compute the normalized incomplete Gamma Function $Q(a,x) = 1/\Gamma(a) \int_x^\infty dt t^{(a-1)} \exp(-t)$ for $a > 0$, $x \geq 0$.

double **gsl_sf_gamma_inc_P** (double *a*, double *x*)
int **gsl_sf_gamma_inc_P_e** (double *a*, double *x*, *gsl_sf_result* * *result*)
These routines compute the complementary normalized incomplete Gamma Function $P(a,x) = 1 - Q(a,x) = 1/\Gamma(a) \int_0^x dt t^{(a-1)} \exp(-t)$ for $a > 0$, $x \geq 0$.

Note that Abramowitz & Stegun call $P(a,x)$ the incomplete gamma function (section 6.5).

7.19.5 Beta Functions

double **gsl_sf_beta** (double *a*, double *b*)
int **gsl_sf_beta_e** (double *a*, double *b*, *gsl_sf_result* * *result*)
These routines compute the Beta Function, $B(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$ subject to a and b not being negative integers.

double **gsl_sf_lnbeta** (double *a*, double *b*)
int **gsl_sf_lnbeta_e** (double *a*, double *b*, *gsl_sf_result* * *result*)
These routines compute the logarithm of the Beta Function, $\log(B(a,b))$ subject to a and b not being negative integers.

7.19.6 Incomplete Beta Function

double **gsl_sf_beta_inc** (double *a*, double *b*, double *x*)
int **gsl_sf_beta_inc_e** (double *a*, double *b*, double *x*, *gsl_sf_result* * *result*)
These routines compute the normalized incomplete Beta function $I_x(a,b) = B_x(a,b)/B(a,b)$ where

$$B_x(a,b) = \int_0^x t^{a-1}(1-t)^{b-1} dt$$

for $0 \leq x \leq 1$. For $a > 0$, $b > 0$ the value is computed using a continued fraction expansion. For all other values it is computed using the relation

$$I_x(a,b,x) = (1/a)x^a {}_2F_1(a, 1-b, a+1, x)/B(a,b)$$

7.20 Gegenbauer Functions

The Gegenbauer polynomials are defined in Abramowitz & Stegun, Chapter 22, where they are known as Ultraspherical polynomials. The functions described in this section are declared in the header file `gsl_sf_gegenbauer.h`.

double **gsl_sf_gegenpoly_1** (double *lambda*, double *x*)
double **gsl_sf_gegenpoly_2** (double *lambda*, double *x*)
double **gsl_sf_gegenpoly_3** (double *lambda*, double *x*)
int **gsl_sf_gegenpoly_1_e** (double *lambda*, double *x*, *gsl_sf_result* * *result*)
int **gsl_sf_gegenpoly_2_e** (double *lambda*, double *x*, *gsl_sf_result* * *result*)
int **gsl_sf_gegenpoly_3_e** (double *lambda*, double *x*, *gsl_sf_result* * *result*)

These functions evaluate the Gegenbauer polynomials $C_n^{(\lambda)}(x)$ using explicit representations for $n = 1, 2, 3$.

double **gsl_sf_gegenpoly_n** (int *n*, double *lambda*, double *x*)
int **gsl_sf_gegenpoly_n_e** (int *n*, double *lambda*, double *x*, *gsl_sf_result* * *result*)

These functions evaluate the Gegenbauer polynomial $C_n^{(\lambda)}(x)$ for a specific value of `n`, `lambda`, `x` subject to $\lambda > -1/2$, $n \geq 0$.

int **gsl_sf_gegenpoly_array** (int *nmax*, double *lambda*, double *x*, double *result_array[]*)

This function computes an array of Gegenbauer polynomials $C_n^{(\lambda)}(x)$ for $n = 0, 1, 2, \ldots, nmax$, subject to $\lambda > -1/2$, $nmax \geq 0$.

7.21 Hermite Polynomials and Functions

Hermite polynomials and functions are discussed in Abramowitz & Stegun, Chapter 22 and Szego, Gabor (1939, 1958, 1967), Orthogonal Polynomials, American Mathematical Society. The Hermite polynomials and functions are defined in the header file `gsl_sf_hermite.h`.

7.21.1 Hermite Polynomials

The Hermite polynomials exist in two variants: the physicist version $H_n(x)$ and the probabilist version $He_n(x)$. They are defined by the derivatives

$$H_n(x) = (-1)^n e^{x^2} \left(\frac{d}{dx}\right)^n e^{-x^2}$$
$$He_n(x) = (-1)^n e^{x^2/2} \left(\frac{d}{dx}\right)^n e^{-x^2/2}$$

They are connected via

$$H_n(x) = 2^{n/2} He_n\left(\sqrt{2}x\right)$$
$$He_n(x) = 2^{-n/2} H_n\left(\frac{x}{\sqrt{2}}\right)$$

and satisfy the ordinary differential equations

$$H_n''(x) - 2xH_n'(x) + 2nH_n(x) = 0$$
$$He_n''(x) - xHe_n'(x) + nHe_n(x) = 0$$

double **gsl_sf_hermite** (const int *n*, const double *x*)

int **gsl_sf_hermite_e** (const int *n*, const double *x*, *gsl_sf_result* * *result*)
These routines evaluate the physicist Hermite polynomial $H_n(x)$ of order n at position x. If an overflow is detected, GSL_EOVRFLW is returned without calling the error handler.

int **gsl_sf_hermite_array** (const int *nmax*, const double *x*, double * *result_array*)
This routine evaluates all physicist Hermite polynomials H_n up to order nmax at position x. The results are stored in result_array.

double **gsl_sf_hermite_series** (const int *n*, const double *x*, const double * *a*)
int **gsl_sf_hermite_series_e** (const int *n*, const double *x*, const double * *a*, *gsl_sf_result* * *result*)
These routines evaluate the series $\sum_{j=0}^{n} a_j H_j(x)$ with H_j being the j-th physicist Hermite polynomial using the Clenshaw algorithm.

double **gsl_sf_hermite_prob** (const int *n*, const double *x*)
int **gsl_sf_hermite_prob_e** (const int *n*, const double *x*, *gsl_sf_result* * *result*)
These routines evaluate the probabilist Hermite polynomial $He_n(x)$ of order n at position x. If an overflow is detected, GSL_EOVRFLW is returned without calling the error handler.

int **gsl_sf_hermite_prob_array** (const int *nmax*, const double *x*, double * *result_array*)
This routine evaluates all probabilist Hermite polynomials $He_n(x)$ up to order nmax at position x. The results are stored in result_array.

double **gsl_sf_hermite_prob_series** (const int *n*, const double *x*, const double * *a*)
int **gsl_sf_hermite_prob_series_e** (const int *n*, const double *x*, const double * *a*, *gsl_sf_result* * *result*)
These routines evaluate the series $\sum_{j=0}^{n} a_j He_j(x)$ with He_j being the j-th probabilist Hermite polynomial using the Clenshaw algorithm.

7.21.2 Derivatives of Hermite Polynomials

double **gsl_sf_hermite_deriv** (const int *m*, const int *n*, const double *x*)
int **gsl_sf_hermite_deriv_e** (const int *m*, const int *n*, const double *x*, *gsl_sf_result* * *result*)
These routines evaluate the m-th derivative of the physicist Hermite polynomial $H_n(x)$ of order n at position x.

int **gsl_sf_hermite_array_deriv** (const int *m*, const int *nmax*, const double *x*, double * *result_array*)
This routine evaluates the m-th derivative of all physicist Hermite polynomials $H_n(x)$ from orders $0, \ldots,$ nmax at position x. The result $d^m/dx^m H_n(x)$ is stored in result_array[n]. The output result_array must have length at least nmax + 1.

int **gsl_sf_hermite_deriv_array** (const int *mmax*, const int *n*, const double *x*, double * *result_array*)
This routine evaluates all derivative orders from $0, \ldots,$ mmax of the physicist Hermite polynomial of order n, H_n, at position x. The result $d^m/dx^m H_n(x)$ is stored in result_array[m]. The output result_array must have length at least mmax + 1.

double **gsl_sf_hermite_prob_deriv** (const int *m*, const int *n*, const double *x*)
int **gsl_sf_hermite_prob_deriv_e** (const int *m*, const int *n*, const double *x*, *gsl_sf_result* * *result*)
These routines evaluate the m-th derivative of the probabilist Hermite polynomial $He_n(x)$ of order n at position x.

int **gsl_sf_hermite_prob_array_deriv** (const int *m*, const int *nmax*, const double *x*, double * *result_array*)
This routine evaluates the m-th derivative of all probabilist Hermite polynomials $He_n(x)$ from orders $0, \ldots,$ nmax at position x. The result $d^m/dx^m He_n(x)$ is stored in result_array[n]. The output result_array must have length at least nmax + 1.

int **gsl_sf_hermite_prob_deriv_array** (const int *mmax*, const int *n*, const double *x*, double * *result_array*)
This routine evaluates all derivative orders from $0, \ldots,$ mmax of the probabilist Hermite polynomial of or-

der n, He_n, at position x. The result $d^m/dx^m He_n(x)$ is stored in result_array[m]. The output result_array must have length at least mmax + 1.

7.21.3 Hermite Functions

The Hermite functions are defined by

$$\psi_n(x) = \left(2^n n! \sqrt{\pi}\right)^{-1/2} e^{-x^2/2} H_n(x)$$

and satisfy the Schrödinger equation for a quantum mechanical harmonic oscillator

$$\psi_n''(x) + (2n + 1 - x^2)\psi_n(x) = 0$$

They are orthonormal,

$$\int_{-\infty}^{\infty} \psi_m(x)\psi_n(x)dx = \delta_{mn}$$

and form an orthonormal basis of $L^2(\mathbb{R})$. The Hermite functions are also eigenfunctions of the continuous Fourier transform. GSL offers two methods for evaluating the Hermite functions. The first uses the standard three-term recurrence relation which has $O(n)$ complexity and is the most accurate. The second uses a Cauchy integral approach due to Bunck (2009) which has $O(\sqrt{n})$ complexity which represents a significant speed improvement for large n, although it is slightly less accurate.

double **gsl_sf_hermite_func** (const int *n*, const double *x*)
int **gsl_sf_hermite_func_e** (const int *n*, const double *x*, *gsl_sf_result* * *result*)
These routines evaluate the Hermite function $\psi_n(x)$ of order n at position x using a three term recurrence relation. The algorithm complexity is $O(n)$.

double **gsl_sf_hermite_func_fast** (const int *n*, const double *x*)
int **gsl_sf_hermite_func_fast_e** (const int *n*, const double *x*, *gsl_sf_result* * *result*)
These routines evaluate the Hermite function $\psi_n(x)$ of order n at position x using a the Cauchy integral algorithm due to Bunck, 2009. The algorithm complexity is $O(\sqrt{n})$.

int **gsl_sf_hermite_func_array** (const int *nmax*, const double *x*, double * *result_array*)
This routine evaluates all Hermite functions $\psi_n(x)$ for orders $n = 0, \ldots,$ nmax at position x, using the recurrence relation algorithm. The results are stored in result_array which has length at least nmax + 1.

double **gsl_sf_hermite_func_series** (const int *n*, const double *x*, const double * *a*)
int **gsl_sf_hermite_func_series_e** (const int *n*, const double *x*, const double * *a*, *gsl_sf_result* * *result*)
These routines evaluate the series $\sum_{j=0}^{n} a_j \psi_j(x)$ with ψ_j being the j-th Hermite function using the Clenshaw algorithm.

7.21.4 Derivatives of Hermite Functions

double **gsl_sf_hermite_func_der** (const int *m*, const int *n*, const double *x*)
int **gsl_sf_hermite_func_der_e** (const int *m*, const int *n*, const double *x*, *gsl_sf_result* * *result*)
These routines evaluate the m-th derivative of the Hermite function $\psi_n(x)$ of order n at position x.

7.21.5 Zeros of Hermite Polynomials and Hermite Functions

These routines calculate the s-th zero of the Hermite polynomial/function of order n. Since the zeros are symmetrical around zero, only positive zeros are calculated, ordered from smallest to largest, starting from index 1. Only for odd polynomial orders a zeroth zero exists, its value always being zero.

double **gsl_sf_hermite_zero** (const int *n*, const int *s*)
int **gsl_sf_hermite_zero_e** (const int *n*, const int *s*, *gsl_sf_result* * *result*)
These routines evaluate the s-th zero of the physicist Hermite polynomial $H_n(x)$ of order n.

double **gsl_sf_hermite_prob_zero** (const int *n*, const int *s*)
int **gsl_sf_hermite_prob_zero_e** (const int *n*, const int *s*, *gsl_sf_result* * *result*)
These routines evaluate the s-th zero of the probabilist Hermite polynomial $He_n(x)$ of order n.

double **gsl_sf_hermite_func_zero** (const int *n*, const int *s*)
int **gsl_sf_hermite_func_zero_e** (const int *n*, const int *s*, *gsl_sf_result* * *result*)
These routines evaluate the s-th zero of the Hermite function $\psi_n(x)$ of order n.

7.22 Hypergeometric Functions

Hypergeometric functions are described in Abramowitz & Stegun, Chapters 13 and 15. These functions are declared in the header file gsl_sf_hyperg.h.

double **gsl_sf_hyperg_0F1** (double *c*, double *x*)
int **gsl_sf_hyperg_0F1_e** (double *c*, double *x*, *gsl_sf_result* * *result*)
These routines compute the hypergeometric function

$$ {}_0F_1(c,x) $$

double **gsl_sf_hyperg_1F1_int** (int *m*, int *n*, double *x*)
int **gsl_sf_hyperg_1F1_int_e** (int *m*, int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the confluent hypergeometric function

$$ {}_1F_1(m,n,x) = M(m,n,x) $$

for integer parameters m, n.

double **gsl_sf_hyperg_1F1** (double *a*, double *b*, double *x*)
int **gsl_sf_hyperg_1F1_e** (double *a*, double *b*, double *x*, *gsl_sf_result* * *result*)
These routines compute the confluent hypergeometric function

$$ {}_1F_1(a,b,x) = M(a,b,x) $$

for general parameters a, b.

double **gsl_sf_hyperg_U_int** (int *m*, int *n*, double *x*)
int **gsl_sf_hyperg_U_int_e** (int *m*, int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the confluent hypergeometric function $U(m,n,x)$ for integer parameters m, n.

int **gsl_sf_hyperg_U_int_e10_e** (int *m*, int *n*, double *x*, *gsl_sf_result_e10* * *result*)
This routine computes the confluent hypergeometric function $U(m,n,x)$ for integer parameters m, n using the *gsl_sf_result_e10* type to return a result with extended range.

double **gsl_sf_hyperg_U** (double *a*, double *b*, double *x*)
int **gsl_sf_hyperg_U_e** (double *a*, double *b*, double *x*, *gsl_sf_result* * *result*)
These routines compute the confluent hypergeometric function $U(a,b,x)$.

int **gsl_sf_hyperg_U_e10_e** (double *a*, double *b*, double *x*, *gsl_sf_result_e10* * *result*)
This routine computes the confluent hypergeometric function $U(a,b,x)$ using the *gsl_sf_result_e10* type to return a result with extended range.

double **gsl_sf_hyperg_2F1** (double *a*, double *b*, double *c*, double *x*)

int **gsl_sf_hyperg_2F1_e** (double *a*, double *b*, double *c*, double *x*, *gsl_sf_result* * *result*)
These routines compute the Gauss hypergeometric function

$$ {}_2F_1(a, b, c, x) = F(a, b, c, x) $$

for $|x| < 1$. If the arguments (a, b, c, x) are too close to a singularity then the function can return the error code GSL_EMAXITER when the series approximation converges too slowly. This occurs in the region of $x = 1$, $c - a - b = m$ for integer m.

double **gsl_sf_hyperg_2F1_conj** (double *aR*, double *aI*, double *c*, double *x*)
int **gsl_sf_hyperg_2F1_conj_e** (double *aR*, double *aI*, double *c*, double *x*, *gsl_sf_result* * *result*)
These routines compute the Gauss hypergeometric function

$$ {}_2F_1(a_R + ia_I, aR - iaI, c, x) $$

with complex parameters for $|x| < 1$.

double **gsl_sf_hyperg_2F1_renorm** (double *a*, double *b*, double *c*, double *x*)
int **gsl_sf_hyperg_2F1_renorm_e** (double *a*, double *b*, double *c*, double *x*, *gsl_sf_result* * *result*)
These routines compute the renormalized Gauss hypergeometric function

$$ {}_2F_1(a, b, c, x)/\Gamma(c) $$

for $|x| < 1$.

double **gsl_sf_hyperg_2F1_conj_renorm** (double *aR*, double *aI*, double *c*, double *x*)
int **gsl_sf_hyperg_2F1_conj_renorm_e** (double *aR*, double *aI*, double *c*, double *x*, *gsl_sf_result* * *result*)
These routines compute the renormalized Gauss hypergeometric function

$$ {}_2F_1(a_R + ia_I, a_R - ia_I, c, x)/\Gamma(c) $$

for $|x| < 1$.

double **gsl_sf_hyperg_2F0** (double *a*, double *b*, double *x*)
int **gsl_sf_hyperg_2F0_e** (double *a*, double *b*, double *x*, *gsl_sf_result* * *result*)
These routines compute the hypergeometric function

$$ {}_2F_0(a, b, x) $$

The series representation is a divergent hypergeometric series. However, for $x < 0$ we have

$$ {}_2F_0(a, b, x) = (-1/x)^a U(a, 1 + a - b, -1/x) $$

7.23 Laguerre Functions

The generalized Laguerre polynomials, sometimes referred to as associated Laguerre polynomials, are defined in terms of confluent hypergeometric functions as

$$ L_n^a(x) = \frac{(a+1)_n}{n!} {}_1F_1(-n, a+1, x) $$

where $(a)_n$ is the *Pochhammer symbol* (rising factorial). They are related to the plain Laguerre polynomials $L_n(x)$ by $L_n^0(x) = L_n(x)$ and $L_n^k(x) = (-1)^k (d^k/dx^k) L_{(n+k)}(x)$ For more information see Abramowitz & Stegun, Chapter 22.

The functions described in this section are declared in the header file gsl_sf_laguerre.h.

double **gsl_sf_laguerre_1** (double *a*, double *x*)
double **gsl_sf_laguerre_2** (double *a*, double *x*)
double **gsl_sf_laguerre_3** (double *a*, double *x*)
int **gsl_sf_laguerre_1_e** (double *a*, double *x*, *gsl_sf_result* * *result*)
int **gsl_sf_laguerre_2_e** (double *a*, double *x*, *gsl_sf_result* * *result*)
int **gsl_sf_laguerre_3_e** (double *a*, double *x*, *gsl_sf_result* * *result*)
These routines evaluate the generalized Laguerre polynomials $L_1^a(x)$, $L_2^a(x)$, $L_3^a(x)$ using explicit representations.

double **gsl_sf_laguerre_n** (const int *n*, const double *a*, const double *x*)
int **gsl_sf_laguerre_n_e** (int *n*, double *a*, double *x*, *gsl_sf_result* * *result*)
These routines evaluate the generalized Laguerre polynomials $L_n^a(x)$ for $a > -1$, $n \geq 0$.

7.24 Lambert W Functions

Lambert's W functions, $W(x)$, are defined to be solutions of the equation $W(x)\exp(W(x)) = x$. This function has multiple branches for $x < 0$; however, it has only two real-valued branches. We define $W_0(x)$ to be the principal branch, where $W > -1$ for $x < 0$, and $W_{-1}(x)$ to be the other real branch, where $W < -1$ for $x < 0$. The Lambert functions are declared in the header file `gsl_sf_lambert.h`.

double **gsl_sf_lambert_W0** (double *x*)
int **gsl_sf_lambert_W0_e** (double *x*, *gsl_sf_result* * *result*)
These compute the principal branch of the Lambert W function, $W_0(x)$.

double **gsl_sf_lambert_Wm1** (double *x*)
int **gsl_sf_lambert_Wm1_e** (double *x*, *gsl_sf_result* * *result*)
These compute the secondary real-valued branch of the Lambert W function, $W_{-1}(x)$.

7.25 Legendre Functions and Spherical Harmonics

The Legendre Functions and Legendre Polynomials are described in Abramowitz & Stegun, Chapter 8. These functions are declared in the header file `gsl_sf_legendre.h`.

7.25.1 Legendre Polynomials

double **gsl_sf_legendre_P1** (double *x*)
double **gsl_sf_legendre_P2** (double *x*)
double **gsl_sf_legendre_P3** (double *x*)
int **gsl_sf_legendre_P1_e** (double *x*, *gsl_sf_result* * *result*)
int **gsl_sf_legendre_P2_e** (double *x*, *gsl_sf_result* * *result*)
int **gsl_sf_legendre_P3_e** (double *x*, *gsl_sf_result* * *result*)
These functions evaluate the Legendre polynomials $P_l(x)$ using explicit representations for $l = 1, 2, 3$.

double **gsl_sf_legendre_Pl** (int *l*, double *x*)
int **gsl_sf_legendre_Pl_e** (int *l*, double *x*, *gsl_sf_result* * *result*)
These functions evaluate the Legendre polynomial $P_l(x)$ for a specific value of `l`, `x` subject to $l \geq 0$ and $|x| \leq 1$.

int **gsl_sf_legendre_Pl_array** (int *lmax*, double *x*, double *result_array[]*)
int **gsl_sf_legendre_Pl_deriv_array** (int *lmax*, double *x*, double *result_array[]*, double *result_deriv_array[]*)
These functions compute arrays of Legendre polynomials $P_l(x)$ and derivatives $dP_l(x)/dx$ for $l = 0, \ldots, lmax$ and $|x| \leq 1$.

double **gsl_sf_legendre_Q0** (double *x*)
int **gsl_sf_legendre_Q0_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the Legendre function $Q_0(x)$ for $x > -1$ and $x \neq 1$.

double **gsl_sf_legendre_Q1** (double *x*)
int **gsl_sf_legendre_Q1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the Legendre function $Q_1(x)$ for $x > -1$ and $x \neq 1$.

double **gsl_sf_legendre_Ql** (int *l*, double *x*)
int **gsl_sf_legendre_Ql_e** (int *l*, double *x*, *gsl_sf_result* * *result*)
These routines compute the Legendre function $Q_l(x)$ for $x > -1$, $x \neq 1$ and $l \geq 0$.

7.25.2 Associated Legendre Polynomials and Spherical Harmonics

The following functions compute the associated Legendre polynomials $P_l^m(x)$ which are solutions of the differential equation

$$(1-x^2)\frac{d^2}{dx^2}P_l^m(x) - 2x\frac{d}{dx}P_l^m(x) + \left(l(l+1) - \frac{m^2}{1-x^2}\right)P_l^m(x) = 0$$

where the degree l and order m satisfy $0 \leq l$ and $0 \leq m \leq l$. The functions $P_l^m(x)$ grow combinatorially with l and can overflow for l larger than about 150. Alternatively, one may calculate normalized associated Legendre polynomials. There are a number of different normalization conventions, and these functions can be stably computed up to degree and order 2700. The following normalizations are provided:

- Schmidt semi-normalization

  Schmidt semi-normalized associated Legendre polynomials are often used in the magnetics community and are defined as

  $$S_l^0(x) = P_l^0(x)$$
  $$S_l^m(x) = (-1)^m\sqrt{2\frac{(l-m)!}{(l+m)!}}P_l^m(x), m > 0$$

  The factor of $(-1)^m$ is called the Condon-Shortley phase factor and can be excluded if desired by setting the parameter `csphase = 1` in the functions below.

- Spherical Harmonic Normalization

  The associated Legendre polynomials suitable for calculating spherical harmonics are defined as

  $$Y_l^m(x) = (-1)^m\sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}}P_l^m(x)$$

  where again the phase factor $(-1)^m$ can be included or excluded if desired.

- Full Normalization

  The fully normalized associated Legendre polynomials are defined as

  $$N_l^m(x) = (-1)^m\sqrt{(l+\frac{1}{2})\frac{(l-m)!}{(l+m)!}}P_l^m(x)$$

  and have the property

  $$\int_{-1}^{1} N_l^m(x)^2 dx = 1$$

The normalized associated Legendre routines below use a recurrence relation which is stable up to a degree and order of about 2700. Beyond this, the computed functions could suffer from underflow leading to incorrect results. Routines are provided to compute first and second derivatives $dP_l^m(x)/dx$ and $d^2P_l^m(x)/dx^2$ as well as their alternate versions $dP_l^m(\cos\theta)/d\theta$ and $d^2P_l^m(\cos\theta)/d\theta^2$. While there is a simple scaling relationship between the two forms, the derivatives involving θ are heavily used in spherical harmonic expansions and so these routines are also provided.

In the functions below, a parameter of type `gsl_sf_legendre_t` specifies the type of normalization to use. The possible values are

gsl_sf_legendre_t

Value	Description
GSL_SF_LEGENDRE_NONE	The unnormalized associated Legendre polynomials $P_l^m(x)$
GSL_SF_LEGENDRE_SCHMIDT	The Schmidt semi-normalized associated Legendre polynomials $S_l^m(x)$
GSL_SF_LEGENDRE_SPHARM	The spherical harmonic associated Legendre polynomials $Y_l^m(x)$
GSL_SF_LEGENDRE_FULL	The fully normalized associated Legendre polynomials $N_l^m(x)$

int **gsl_sf_legendre_array**(const *gsl_sf_legendre_t norm*, const size_t *lmax*, const double *x*, double *result_array[]*)

int **gsl_sf_legendre_array_e**(const *gsl_sf_legendre_t norm*, const size_t *lmax*, const double *x*, const double *csphase*, double *result_array[]*)

These functions calculate all normalized associated Legendre polynomials for $0 \le l \le lmax$ and $0 \le m \le l$ for $|x| \le 1$. The `norm` parameter specifies which normalization is used. The normalized $P_l^m(x)$ values are stored in `result_array`, whose minimum size can be obtained from calling `gsl_sf_legendre_array_n()`. The array index of $P_l^m(x)$ is obtained from calling `gsl_sf_legendre_array_index(l, m)`. To include or exclude the Condon-Shortley phase factor of $(-1)^m$, set the parameter `csphase` to either -1 or 1 respectively in the `_e` function. This factor is excluded by default.

int **gsl_sf_legendre_deriv_array**(const *gsl_sf_legendre_t norm*, const size_t *lmax*, const double *x*, double *result_array[]*, double *result_deriv_array[]*)

int **gsl_sf_legendre_deriv_array_e**(const *gsl_sf_legendre_t norm*, const size_t *lmax*, const double *x*, const double *csphase*, double *result_array[]*, double *result_deriv_array[]*)

These functions calculate all normalized associated Legendre functions and their first derivatives up to degree `lmax` for $|x| < 1$. The parameter `norm` specifies the normalization used. The normalized $P_l^m(x)$ values and their derivatives $dP_l^m(x)/dx$ are stored in `result_array` and `result_deriv_array` respectively. To include or exclude the Condon-Shortley phase factor of $(-1)^m$, set the parameter `csphase` to either -1 or 1 respectively in the `_e` function. This factor is excluded by default.

int **gsl_sf_legendre_deriv_alt_array**(const *gsl_sf_legendre_t norm*, const size_t *lmax*, const double *x*, double *result_array[]*, double *result_deriv_array[]*)

int **gsl_sf_legendre_deriv_alt_array_e**(const *gsl_sf_legendre_t norm*, const size_t *lmax*, const double *x*, const double *csphase*, double *result_array[]*, double *result_deriv_array[]*)

These functions calculate all normalized associated Legendre functions and their (alternate) first derivatives up to degree `lmax` for $|x| < 1$. The normalized $P_l^m(x)$ values and their derivatives $dP_l^m(\cos\theta)/d\theta$ are stored in `result_array` and `result_deriv_array` respectively. To include or exclude the Condon-Shortley phase factor of $(-1)^m$, set the parameter `csphase` to either -1 or 1 respectively in the `_e` function. This factor is excluded by default.

int **gsl_sf_legendre_deriv2_array**(const *gsl_sf_legendre_t norm*, const size_t *lmax*, const double *x*, double *result_array[]*, double *result_deriv_array[]*, double *result_deriv2_array[]*)

int **gsl_sf_legendre_deriv2_array_e** (const *gsl_sf_legendre_t norm*, const size_t *lmax*, const double *x*, const double *csphase*, double *result_array[]*, double *result_deriv_array[]*, double *result_deriv2_array[]*)

These functions calculate all normalized associated Legendre functions and their first and second derivatives up to degree `lmax` for $|x| < 1$. The parameter `norm` specifies the normalization used. The normalized $P_l^m(x)$, their first derivatives $dP_l^m(x)/dx$, and their second derivatives $d^2P_l^m(x)/dx^2$ are stored in `result_array`, `result_deriv_array`, and `result_deriv2_array` respectively. To include or exclude the Condon-Shortley phase factor of $(-1)^m$, set the parameter `csphase` to either -1 or 1 respectively in the `_e` function. This factor is excluded by default.

int **gsl_sf_legendre_deriv2_alt_array** (const *gsl_sf_legendre_t norm*, const size_t *lmax*, const double *x*, double *result_array[]*, double *result_deriv_array[]*, double *result_deriv2_array[]*)

int **gsl_sf_legendre_deriv2_alt_array_e** (const *gsl_sf_legendre_t norm*, const size_t *lmax*, const double *x*, const double *csphase*, double *result_array[]*, double *result_deriv_array[]*, double *result_deriv2_array[]*)

These functions calculate all normalized associated Legendre functions and their (alternate) first and second derivatives up to degree `lmax` for $|x| < 1$. The parameter `norm` specifies the normalization used. The normalized $P_l^m(x)$, their first derivatives $dP_l^m(\cos\theta)/d\theta$, and their second derivatives $d^2P_l^m(\cos\theta)/d\theta^2$ are stored in `result_array`, `result_deriv_array`, and `result_deriv2_array` respectively. To include or exclude the Condon-Shortley phase factor of $(-1)^m$, set the parameter `csphase` to either -1 or 1 respectively in the `_e` function. This factor is excluded by default.

size_t **gsl_sf_legendre_array_n** (const size_t *lmax*)

This function returns the minimum array size for maximum degree `lmax` needed for the array versions of the associated Legendre functions. Size is calculated as the total number of $P_l^m(x)$ functions, plus extra space for precomputing multiplicative factors used in the recurrence relations.

size_t **gsl_sf_legendre_array_index** (const size_t *l*, const size_t *m*)

This function returns the index into `result_array`, `result_deriv_array`, or `result_deriv2_array` corresponding to $P_l^m(x)$, $P_l^{'m}(x)$, or $P_l^{''m}(x)$. The index is given by $l(l+1)/2 + m$.

double **gsl_sf_legendre_Plm** (int *l*, int *m*, double *x*)

int **gsl_sf_legendre_Plm_e** (int *l*, int *m*, double *x*, *gsl_sf_result* * *result*)

These routines compute the associated Legendre polynomial $P_l^m(x)$ for $m \geq 0$, $l \geq m$, and $|x| \leq 1$.

double **gsl_sf_legendre_sphPlm** (int *l*, int *m*, double *x*)

int **gsl_sf_legendre_sphPlm_e** (int *l*, int *m*, double *x*, *gsl_sf_result* * *result*)

These routines compute the normalized associated Legendre polynomial $\sqrt{(2l+1)/(4\pi)}\sqrt{(l-m)!/(l+m)!}P_l^m(x)$ suitable for use in spherical harmonics. The parameters must satisfy $m \geq 0$, $l \geq m$, and $|x| \leq 1$. These routines avoid the overflows that occur for the standard normalization of $P_l^m(x)$.

int **gsl_sf_legendre_Plm_array** (int *lmax*, int *m*, double *x*, double *result_array[]*)

int **gsl_sf_legendre_Plm_deriv_array** (int *lmax*, int *m*, double *x*, double *result_array[]*, double *result_deriv_array[]*)

These functions are now deprecated and will be removed in a future release; see *gsl_sf_legendre_array()* and *gsl_sf_legendre_deriv_array()*.

int **gsl_sf_legendre_sphPlm_array** (int *lmax*, int *m*, double *x*, double *result_array[]*)

int **gsl_sf_legendre_sphPlm_deriv_array** (int *lmax*, int *m*, double *x*, double *result_array[]*, double *result_deriv_array[]*)

These functions are now deprecated and will be removed in a future release; see *gsl_sf_legendre_array()* and *gsl_sf_legendre_deriv_array()*.

int **gsl_sf_legendre_array_size** (const int *lmax*, const int *m*)

This function is now deprecated and will be removed in a future release.

7.25.3 Conical Functions

The Conical Functions $P^{\mu}_{-(1/2)+i\lambda}(x)$ and $Q^{\mu}_{-(1/2)+i\lambda}$ are described in Abramowitz & Stegun, Section 8.12.

double **gsl_sf_conicalP_half** (double *lambda*, double *x*)
int **gsl_sf_conicalP_half_e** (double *lambda*, double *x*, *gsl_sf_result* * *result*)
These routines compute the irregular Spherical Conical Function $P^{1/2}_{-1/2+i\lambda}(x)$ for $x > -1$.

double **gsl_sf_conicalP_mhalf** (double *lambda*, double *x*)
int **gsl_sf_conicalP_mhalf_e** (double *lambda*, double *x*, *gsl_sf_result* * *result*)
These routines compute the regular Spherical Conical Function $P^{-1/2}_{-1/2+i\lambda}(x)$ for $x > -1$.

double **gsl_sf_conicalP_0** (double *lambda*, double *x*)
int **gsl_sf_conicalP_0_e** (double *lambda*, double *x*, *gsl_sf_result* * *result*)
These routines compute the conical function $P^{0}_{-1/2+i\lambda}(x)$ for $x > -1$.

double **gsl_sf_conicalP_1** (double *lambda*, double *x*)
int **gsl_sf_conicalP_1_e** (double *lambda*, double *x*, *gsl_sf_result* * *result*)
These routines compute the conical function $P^{1}_{-1/2+i\lambda}(x)$ for $x > -1$.

double **gsl_sf_conicalP_sph_reg** (int *l*, double *lambda*, double *x*)
int **gsl_sf_conicalP_sph_reg_e** (int *l*, double *lambda*, double *x*, *gsl_sf_result* * *result*)
These routines compute the Regular Spherical Conical Function $P^{-1/2-l}_{-1/2+i\lambda}(x)$ for $x > -1$ and $l \geq -1$.

double **gsl_sf_conicalP_cyl_reg** (int *m*, double *lambda*, double *x*)
int **gsl_sf_conicalP_cyl_reg_e** (int *m*, double *lambda*, double *x*, *gsl_sf_result* * *result*)
These routines compute the Regular Cylindrical Conical Function $P^{-m}_{-1/2+i\lambda}(x)$ for $x > -1$ and $m \geq -1$.

7.25.4 Radial Functions for Hyperbolic Space

The following spherical functions are specializations of Legendre functions which give the regular eigenfunctions of the Laplacian on a 3-dimensional hyperbolic space H^3. Of particular interest is the flat limit, $\lambda \to \infty$, $\eta \to 0$, $\lambda\eta$ fixed.

double **gsl_sf_legendre_H3d_0** (double *lambda*, double *eta*)
int **gsl_sf_legendre_H3d_0_e** (double *lambda*, double *eta*, *gsl_sf_result* * *result*)
These routines compute the zeroth radial eigenfunction of the Laplacian on the 3-dimensional hyperbolic space,

$$L^{H3d}_0(\lambda, \eta) := \frac{\sin(\lambda\eta)}{\lambda \sinh(\eta)}$$

for $\eta \geq 0$. In the flat limit this takes the form $L^{H3d}_0(\lambda, \eta) = j_0(\lambda\eta)$.

double **gsl_sf_legendre_H3d_1** (double *lambda*, double *eta*)
int **gsl_sf_legendre_H3d_1_e** (double *lambda*, double *eta*, *gsl_sf_result* * *result*)
These routines compute the first radial eigenfunction of the Laplacian on the 3-dimensional hyperbolic space,

$$L^{H3d}_1(\lambda, \eta) := \frac{1}{\sqrt{\lambda^2 + 1}} \left(\frac{\sin(\lambda\eta)}{\lambda \sinh(\eta)} \right) (\coth(\eta) - \lambda \cot(\lambda\eta))$$

for $\eta \geq 0$ In the flat limit this takes the form $L^{H3d}_1(\lambda, \eta) = j_1(\lambda\eta)$.

double **gsl_sf_legendre_H3d** (int *l*, double *lambda*, double *eta*)
int **gsl_sf_legendre_H3d_e** (int *l*, double *lambda*, double *eta*, *gsl_sf_result* * *result*)
These routines compute the `l`-th radial eigenfunction of the Laplacian on the 3-dimensional hyperbolic space $\eta \geq 0$ and $l \geq 0$. In the flat limit this takes the form $L^{H3d}_l(\lambda, \eta) = j_l(\lambda\eta)$.

int **gsl_sf_legendre_H3d_array** (int *lmax*, double *lambda*, double *eta*, double *result_array[]*)
This function computes an array of radial eigenfunctions $L^{H3d}_l(\lambda, \eta)$ for $0 \leq l \leq lmax$.

7.26 Logarithm and Related Functions

Information on the properties of the Logarithm function can be found in Abramowitz & Stegun, Chapter 4. The functions described in this section are declared in the header file `gsl_sf_log.h`.

double **gsl_sf_log** (double *x*)
int **gsl_sf_log_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the logarithm of `x`, $\log(x)$, for $x > 0$.

double **gsl_sf_log_abs** (double *x*)
int **gsl_sf_log_abs_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the logarithm of the magnitude of `x`, $\log(|x|)$, for $x \neq 0$.

int **gsl_sf_complex_log_e** (double *zr*, double *zi*, *gsl_sf_result* * *lnr*, *gsl_sf_result* * *theta*)
This routine computes the complex logarithm of $z = z_r + iz_i$. The results are returned as `lnr`, `theta` such that $\exp(lnr + i\theta) = z_r + iz_i$, where θ lies in the range $[-\pi, \pi]$.

double **gsl_sf_log_1plusx** (double *x*)
int **gsl_sf_log_1plusx_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute $\log(1 + x)$ for $x > -1$ using an algorithm that is accurate for small `x`.

double **gsl_sf_log_1plusx_mx** (double *x*)
int **gsl_sf_log_1plusx_mx_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute $\log(1 + x) - x$ for $x > -1$ using an algorithm that is accurate for small `x`.

7.27 Mathieu Functions

The routines described in this section compute the angular and radial Mathieu functions, and their characteristic values. Mathieu functions are the solutions of the following two differential equations:

$$\frac{d^2y}{dv^2} + (a - 2q\cos 2v)y = 0$$
$$\frac{d^2f}{du^2} - (a - 2q\cosh 2u)f = 0$$

The angular Mathieu functions $ce_r(x, q)$, $se_r(x, q)$ are the even and odd periodic solutions of the first equation, which is known as Mathieu's equation. These exist only for the discrete sequence of characteristic values $a = a_r(q)$ (even-periodic) and $a = b_r(q)$ (odd-periodic).

The radial Mathieu functions $Mc_r^{(j)}(z, q)$ and $Ms_r^{(j)}(z, q)$ are the solutions of the second equation, which is referred to as Mathieu's modified equation. The radial Mathieu functions of the first, second, third and fourth kind are denoted by the parameter j, which takes the value 1, 2, 3 or 4.

For more information on the Mathieu functions, see Abramowitz and Stegun, Chapter 20. These functions are defined in the header file `gsl_sf_mathieu.h`.

7.27.1 Mathieu Function Workspace

The Mathieu functions can be computed for a single order or for multiple orders, using array-based routines. The array-based routines require a preallocated workspace.

gsl_sf_mathieu_workspace
Workspace required for array-based routines

gsl_sf_mathieu_workspace * **gsl_sf_mathieu_alloc** (size_t *n*, double *qmax*)
This function returns a workspace for the array versions of the Mathieu routines. The arguments n and qmax specify the maximum order and q-value of Mathieu functions which can be computed with this workspace.

void **gsl_sf_mathieu_free** (*gsl_sf_mathieu_workspace* * *work*)
This function frees the workspace work.

7.27.2 Mathieu Function Characteristic Values

int **gsl_sf_mathieu_a** (int *n*, double *q*)
int **gsl_sf_mathieu_a_e** (int *n*, double *q*, *gsl_sf_result* * *result*)
int **gsl_sf_mathieu_b** (int *n*, double *q*)
int **gsl_sf_mathieu_b_e** (int *n*, double *q*, *gsl_sf_result* * *result*)
These routines compute the characteristic values $a_n(q)$, $b_n(q)$ of the Mathieu functions $ce_n(q,x)$ and $se_n(q,x)$, respectively.

int **gsl_sf_mathieu_a_array** (int *order_min*, int *order_max*, double *q*, *gsl_sf_mathieu_workspace* * *work*, double *result_array[]*)
int **gsl_sf_mathieu_b_array** (int *order_min*, int *order_max*, double *q*, *gsl_sf_mathieu_workspace* * *work*, double *result_array[]*)
These routines compute a series of Mathieu characteristic values $a_n(q)$, $b_n(q)$ for n from order_min to order_max inclusive, storing the results in the array result_array.

7.27.3 Angular Mathieu Functions

int **gsl_sf_mathieu_ce** (int *n*, double *q*, double *x*)
int **gsl_sf_mathieu_ce_e** (int *n*, double *q*, double *x*, *gsl_sf_result* * *result*)
int **gsl_sf_mathieu_se** (int *n*, double *q*, double *x*)
int **gsl_sf_mathieu_se_e** (int *n*, double *q*, double *x*, *gsl_sf_result* * *result*)
These routines compute the angular Mathieu functions $ce_n(q,x)$ and $se_n(q,x)$, respectively.

int **gsl_sf_mathieu_ce_array** (int *nmin*, int *nmax*, double *q*, double *x*, *gsl_sf_mathieu_workspace* * *work*, double *result_array[]*)
int **gsl_sf_mathieu_se_array** (int *nmin*, int *nmax*, double *q*, double *x*, *gsl_sf_mathieu_workspace* * *work*, double *result_array[]*)
These routines compute a series of the angular Mathieu functions $ce_n(q,x)$ and $se_n(q,x)$ of order n from nmin to nmax inclusive, storing the results in the array result_array.

7.27.4 Radial Mathieu Functions

int **gsl_sf_mathieu_Mc** (int *j*, int *n*, double *q*, double *x*)
int **gsl_sf_mathieu_Mc_e** (int *j*, int *n*, double *q*, double *x*, *gsl_sf_result* * *result*)
int **gsl_sf_mathieu_Ms** (int *j*, int *n*, double *q*, double *x*)
int **gsl_sf_mathieu_Ms_e** (int *j*, int *n*, double *q*, double *x*, *gsl_sf_result* * *result*)
These routines compute the radial j-th kind Mathieu functions $Mc_n^{(j)}(q,x)$ and $Ms_n^{(j)}(q,x)$ of order n.

The allowed values of j are 1 and 2. The functions for $j = 3, 4$ can be computed as $M_n^{(3)} = M_n^{(1)} + iM_n^{(2)}$ and $M_n^{(4)} = M_n^{(1)} - iM_n^{(2)}$, where $M_n^{(j)} = Mc_n^{(j)}$ or $Ms_n^{(j)}$.

int **gsl_sf_mathieu_Mc_array** (int *j*, int *nmin*, int *nmax*, double *q*, double *x*, *gsl_sf_mathieu_workspace* * *work*, double *result_array[]*)
int **gsl_sf_mathieu_Ms_array** (int *j*, int *nmin*, int *nmax*, double *q*, double *x*, *gsl_sf_mathieu_workspace* * *work*, double *result_array[]*)
These routines compute a series of the radial Mathieu functions of kind j, with order from nmin to nmax inclusive, storing the results in the array result_array.

7.28 Power Function

The following functions are equivalent to the function *gsl_pow_int()* with an error estimate. These functions are declared in the header file `gsl_sf_pow_int.h`.

double **gsl_sf_pow_int** (double *x*, int *n*)
int **gsl_sf_pow_int_e** (double *x*, int *n*, *gsl_sf_result* * *result*)
These routines compute the power x^n for integer `n`. The power is computed using the minimum number of multiplications. For example, x^8 is computed as $((x^2)^2)^2$, requiring only 3 multiplications. For reasons of efficiency, these functions do not check for overflow or underflow conditions. The following is a simple example:

```
#include <gsl/gsl_sf_pow_int.h>
/* compute 3.0**12 */
double y = gsl_sf_pow_int(3.0, 12);
```

7.29 Psi (Digamma) Function

The polygamma functions of order n are defined by

$$\psi^{(n)}(x) = \left(\frac{d}{dx}\right)^n \psi(x) = \left(\frac{d}{dx}\right)^{n+1} \log(\Gamma(x))$$

where $\psi(x) = \Gamma'(x)/\Gamma(x)$ is known as the digamma function. These functions are declared in the header file `gsl_sf_psi.h`.

7.29.1 Digamma Function

double **gsl_sf_psi_int** (int *n*)
int **gsl_sf_psi_int_e** (int *n*, *gsl_sf_result* * *result*)
These routines compute the digamma function $\psi(n)$ for positive integer `n`. The digamma function is also called the Psi function.

double **gsl_sf_psi** (double *x*)
int **gsl_sf_psi_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the digamma function $\psi(x)$ for general `x`, $x \neq 0$.

double **gsl_sf_psi_1piy** (double *y*)
int **gsl_sf_psi_1piy_e** (double *y*, *gsl_sf_result* * *result*)
These routines compute the real part of the digamma function on the line $1 + iy$, $\Re[\psi(1 + iy)]$.

7.29.2 Trigamma Function

double **gsl_sf_psi_1_int** (int *n*)
int **gsl_sf_psi_1_int_e** (int *n*, *gsl_sf_result* * *result*)
These routines compute the Trigamma function $\psi'(n)$ for positive integer n.

double **gsl_sf_psi_1** (double *x*)
int **gsl_sf_psi_1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the Trigamma function $\psi'(x)$ for general `x`.

7.29.3 Polygamma Function

double **gsl_sf_psi_n** (int *n*, double *x*)
int **gsl_sf_psi_n_e** (int *n*, double *x*, *gsl_sf_result* * *result*)
These routines compute the polygamma function $\psi^{(n)}(x)$ for $n \geq 0$, $x > 0$.

7.30 Synchrotron Functions

The functions described in this section are declared in the header file `gsl_sf_synchrotron.h`.

double **gsl_sf_synchrotron_1** (double *x*)
int **gsl_sf_synchrotron_1_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the first synchrotron function $x \int_x^\infty dt K_{5/3}(t)$ for $x \geq 0$.

double **gsl_sf_synchrotron_2** (double *x*)
int **gsl_sf_synchrotron_2_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the second synchrotron function $xK_{2/3}(x)$ for $x \geq 0$.

7.31 Transport Functions

The transport functions $J(n, x)$ are defined by the integral representations

$$J(n, x) = \int_0^x t^n e^t / (e^t - 1)^2 dt$$

They are declared in the header file `gsl_sf_transport.h`.

double **gsl_sf_transport_2** (double *x*)
int **gsl_sf_transport_2_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the transport function $J(2, x)$.

double **gsl_sf_transport_3** (double *x*)
int **gsl_sf_transport_3_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the transport function $J(3, x)$.

double **gsl_sf_transport_4** (double *x*)
int **gsl_sf_transport_4_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the transport function $J(4, x)$.

double **gsl_sf_transport_5** (double *x*)
int **gsl_sf_transport_5_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the transport function $J(5, x)$.

7.32 Trigonometric Functions

The library includes its own trigonometric functions in order to provide consistency across platforms and reliable error estimates. These functions are declared in the header file `gsl_sf_trig.h`.

7.32.1 Circular Trigonometric Functions

double **gsl_sf_sin** (double *x*)

int **gsl_sf_sin_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the sine function $\sin(x)$.

double **gsl_sf_cos** (double *x*)
int **gsl_sf_cos_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute the cosine function $\cos(x)$.

double **gsl_sf_hypot** (double *x*, double *y*)
int **gsl_sf_hypot_e** (double *x*, double *y*, *gsl_sf_result* * *result*)
These routines compute the hypotenuse function $\sqrt{x^2 + y^2}$ avoiding overflow and underflow.

double **gsl_sf_sinc** (double *x*)
int **gsl_sf_sinc_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute $\mathrm{sinc}(x) = \sin(\pi x)/(\pi x)$ for any value of `x`.

7.32.2 Trigonometric Functions for Complex Arguments

int **gsl_sf_complex_sin_e** (double *zr*, double *zi*, *gsl_sf_result* * *szr*, *gsl_sf_result* * *szi*)
This function computes the complex sine, $\sin(z_r + iz_i)$ storing the real and imaginary parts in `szr`, `szi`.

int **gsl_sf_complex_cos_e** (double *zr*, double *zi*, *gsl_sf_result* * *czr*, *gsl_sf_result* * *czi*)
This function computes the complex cosine, $\cos(z_r + iz_i)$ storing the real and imaginary parts in `czr`, `czi`.

int **gsl_sf_complex_logsin_e** (double *zr*, double *zi*, *gsl_sf_result* * *lszr*, *gsl_sf_result* * *lszi*)
This function computes the logarithm of the complex sine, $\log(\sin(z_r + iz_i))$ storing the real and imaginary parts in `lszr`, `lszi`.

7.32.3 Hyperbolic Trigonometric Functions

double **gsl_sf_lnsinh** (double *x*)
int **gsl_sf_lnsinh_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute $\log(\sinh(x))$ for $x > 0$.

double **gsl_sf_lncosh** (double *x*)
int **gsl_sf_lncosh_e** (double *x*, *gsl_sf_result* * *result*)
These routines compute $\log(\cosh(x))$ for any `x`.

7.32.4 Conversion Functions

int **gsl_sf_polar_to_rect** (double *r*, double *theta*, *gsl_sf_result* * *x*, *gsl_sf_result* * *y*)
This function converts the polar coordinates (`r`, `theta`) to rectilinear coordinates (`x`, `y`), $x = r\cos(\theta)$, $y = r\sin(\theta)$.

int **gsl_sf_rect_to_polar** (double *x*, double *y*, *gsl_sf_result* * *r*, *gsl_sf_result* * *theta*)
This function converts the rectilinear coordinates (`x`, `y`) to polar coordinates (`r`, `theta`), such that $x = r\cos(\theta)$, $y = r\sin(\theta)$. The argument `theta` lies in the range $[-\pi, \pi]$.

7.32.5 Restriction Functions

double **gsl_sf_angle_restrict_symm** (double *theta*)
int **gsl_sf_angle_restrict_symm_e** (double * *theta*)
These routines force the angle `theta` to lie in the range $(-\pi, \pi]$.

Note that the mathematical value of π is slightly greater than `M_PI`, so the machine numbers `M_PI` and `-M_PI` are included in the range.

double **gsl_sf_angle_restrict_pos** (double *theta*)
int **gsl_sf_angle_restrict_pos_e** (double * *theta*)
These routines force the angle `theta` to lie in the range $[0, 2\pi)$.

Note that the mathematical value of 2π is slightly greater than `2*M_PI`, so the machine number `2*M_PI` is included in the range.

7.32.6 Trigonometric Functions With Error Estimates

int **gsl_sf_sin_err_e** (double *x*, double *dx*, *gsl_sf_result* * *result*)
This routine computes the sine of an angle `x` with an associated absolute error `dx`, $\sin(x \pm dx)$. Note that this function is provided in the error-handling form only since its purpose is to compute the propagated error.

int **gsl_sf_cos_err_e** (double *x*, double *dx*, *gsl_sf_result* * *result*)
This routine computes the cosine of an angle `x` with an associated absolute error `dx`, $\cos(x \pm dx)$. Note that this function is provided in the error-handling form only since its purpose is to compute the propagated error.

7.33 Zeta Functions

The Riemann zeta function is defined in Abramowitz & Stegun, Section 23.2. The functions described in this section are declared in the header file `gsl_sf_zeta.h`.

7.33.1 Riemann Zeta Function

The Riemann zeta function is defined by the infinite sum

$$\zeta(s) = \sum_{k=1}^{\infty} k^{-s}$$

double **gsl_sf_zeta_int** (int *n*)
int **gsl_sf_zeta_int_e** (int *n*, *gsl_sf_result* * *result*)
These routines compute the Riemann zeta function $\zeta(n)$ for integer `n`, $n \neq 1$.

double **gsl_sf_zeta** (double *s*)
int **gsl_sf_zeta_e** (double *s*, *gsl_sf_result* * *result*)
These routines compute the Riemann zeta function $\zeta(s)$ for arbitrary `s`, $s \neq 1$.

7.33.2 Riemann Zeta Function Minus One

For large positive argument, the Riemann zeta function approaches one. In this region the fractional part is interesting, and therefore we need a function to evaluate it explicitly.

double **gsl_sf_zetam1_int** (int *n*)
int **gsl_sf_zetam1_int_e** (int *n*, *gsl_sf_result* * *result*)
These routines compute $\zeta(n) - 1$ for integer `n`, $n \neq 1$.

double **gsl_sf_zetam1** (double *s*)
int **gsl_sf_zetam1_e** (double *s*, *gsl_sf_result* * *result*)
These routines compute $\zeta(s) - 1$ for arbitrary `s`, $s \neq 1$.

7.33.3 Hurwitz Zeta Function

The Hurwitz zeta function is defined by

$$\zeta(s,q) = \sum_0^\infty (k+q)^{-s}$$

double **gsl_sf_hzeta** (double *s*, double *q*)
int **gsl_sf_hzeta_e** (double *s*, double *q*, *gsl_sf_result* * *result*)
These routines compute the Hurwitz zeta function $\zeta(s,q)$ for $s > 1$, $q > 0$.

7.33.4 Eta Function

The eta function is defined by

$$\eta(s) = (1 - 2^{1-s})\zeta(s)$$

double **gsl_sf_eta_int** (int *n*)
int **gsl_sf_eta_int_e** (int *n*, *gsl_sf_result* * *result*)
These routines compute the eta function $\eta(n)$ for integer `n`.

double **gsl_sf_eta** (double *s*)
int **gsl_sf_eta_e** (double *s*, *gsl_sf_result* * *result*)
These routines compute the eta function $\eta(s)$ for arbitrary `s`.

7.34 Examples

The following example demonstrates the use of the error handling form of the special functions, in this case to compute the Bessel function $J_0(5.0)$,

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_sf_bessel.h>

int
main (void)
{
  double x = 5.0;
  gsl_sf_result result;

  double expected = -0.17759677131433830434739701;

  int status = gsl_sf_bessel_J0_e (x, &result);

  printf ("status  = %s\n", gsl_strerror(status));
  printf ("J0(5.0) = %.18f\n"
          "      +/- % .18f\n",
          result.val, result.err);
  printf ("exact   = %.18f\n", expected);
  return status;
}
```

Here are the results of running the program,

```
status  = success
J0(5.0) = -0.177596771314338264
      +/-  0.000000000000000193
exact   = -0.177596771314338292
```

The next program computes the same quantity using the natural form of the function. In this case the error term `result.err` and return status are not accessible.

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>

int
main (void)
{
  double x = 5.0;
  double expected = -0.17759677131433830434739701;

  double y = gsl_sf_bessel_J0 (x);

  printf ("J0(5.0) = %.18f\n", y);
  printf ("exact   = %.18f\n", expected);
  return 0;
}
```

The results of the function are the same,

```
J0(5.0) = -0.177596771314338264
exact   = -0.177596771314338292
```

7.35 References and Further Reading

The library follows the conventions of the following book where possible,

- Handbook of Mathematical Functions, edited by Abramowitz & Stegun, Dover, ISBN 0486612724.

The following papers contain information on the algorithms used to compute the special functions,

- Allan J. MacLeod, MISCFUN: A software package to compute uncommon special functions. ACM Trans. Math. Soft., vol.: 22, 1996, 288–301
- Bunck, B. F., A fast algorithm for evaluation of normalized Hermite functions, BIT Numer. Math, 49: 281-295, 2009.
- G.N. Watson, A Treatise on the Theory of Bessel Functions, 2nd Edition (Cambridge University Press, 1944).
- G. Nemeth, Mathematical Approximations of Special Functions, Nova Science Publishers, ISBN 1-56072-052-2
- B.C. Carlson, Special Functions of Applied Mathematics (1977)
- N. M. Temme, Special Functions: An Introduction to the Classical Functions of Mathematical Physics (1996), ISBN 978-0471113133.
- W.J. Thompson, Atlas for Computing Mathematical Functions, John Wiley & Sons, New York (1997).
- Y.Y. Luke, Algorithms for the Computation of Mathematical Functions, Academic Press, New York (1977).

- S. A. Holmes and W. E. Featherstone, A unified approach to the Clenshaw summation and the recursive computation of very high degree and order normalised associated Legendre functions, Journal of Geodesy, 76, pg. 279-299, 2002.