Documentation for RnSort.h and RnSort.c

Steven Andrews, © 2003-2012
See the document "LibDoc" for general information about this and other libraries.

Header file

```
#ifndef __RnSort_h
#define __RnSort_h

/************************   sorting functions   ****************************/

void sortV(float *a,float *b,int n);
void sortVdbl(double *a,double *b,int n);
void sortCV(float *a,float *bc,int n);
void sortVliv(long int *a,void **b,int n);
void sortVoid(void **a,int n,int (*compare)(void*,void*));
int sortinsertVsi(char **strlist,char *str,int *xlist,int x,int n);

/************************   locating functions   ****************************/

int locateV(float *a,float x,int n);
int locateVdbl(double *a,double x,int n);
int locateVli(long int *a,long int x,int n);
int locateVstr(char **a,char *x,int n);

/********************   interpolation functions   **************************/

float interpolate1(float *ax,float *ay,int n,int *j,float x);
double interpolate1dbl(double *ax,double *ay,int n,int *j,double x);
float interpolate1Cr(float *ax,float *ayc,int n,int *j,float x);
float interpolate1Ci(float *ax,float *ayc,int n,int *j,float x);
double cubicinterpolate1D(double *xdata,double *ydata,int n,double x);
double cubicinterpolate2D(double *xdata,double *ydata,double *zdata,int nx,int
ny,double x,double y);

/************************   resampling functions   **************************/

void convertxV(float *ax,float *ay,float *cx,float *cy,int na,int nc);
void convertxCV(float *ax,float *ayc,float *cx,float *cyc,int na,int nc);

/************************   histogram functions   ***************************/

void setuphist(float *hist,float *scale,int hn,float low,float high);
void setuphistdbl(double *hist,double *scale,int hn,double low,double high);
int histbin(float value,float *scale,int hn);
int histbindbl(double value,double *scale,int hn);
void data2hist(float *data,int dn,char op,float *hist,float *scale,int hn);

/********************   event analysis functions   **************************/

double maxeventrateVD(double *event,double *weight,int n,double sigma,double
```

```
*tptr);

/*****************   concentration profile functions   **********************/

double *cpxinitializer(int n,double *r,double rlow,double rhigh,double rjump);
double *cpxinitializec(double *r,double *c,int n,double *values,int code);
double cp1integrate(double *r,double *rdf,int n,double rmin,double rmax,int
upperleft);
double cp2integrate(double *r,double *rdf,int n,double rmin,double rmax,int
upperleft);
double cp3integrate(double *r,double *rdf,int n,double rmin,double rmax,int
upperleft);
void cp1diffuse(double *r,double *rdfa,double *rdfd,int n,double rmsstep,double
cminfinity,double cinfinity);
void cp3diffuse(double *r,double *rdfa,double *rdfd,int n,double rmsstep,double
cinfinity);
double cp1absorb(double *r,double *rdf,int n,double rabsorb);
double cp2absorb(double *r,double *rdf,int n,double rabsorb);
double cp3absorb(double *r,double *rdf,int n,double rabsorb);
void cpxaddconc(double *r,double *rdf,int n,double amount,int profile,double
r1,double r2);
void cpxmassactionreact(double *rdfa,double *rdfb,int n,double rate);

#endif
```

Requires: `<math.h>`,`"Rn.h"`
Example program: `SpectFit.c, LibTest.c`

History: Written 12/98. Works with Metrowerks C. Moderate testing. Added complex function 2/02. Added some double precision routines 10/02. Added `maxeventrateVD` 9/05. Added `sortVliv` and `locateVli` 11/29/06. Added `setuphist` and `data2hist` 12/06. Added `histbin` 4/07. Added setuphisdbl and histbindbl 6/17/07. Quick overhaul 9/27/11; also added cubic interpolation functions. Added `sortVoid` 6/3/12. Added `locateVstr` 10/8/13 and `sortinsertVsi` 10/14/13.

These routines work on sorted arrays of floats, doubles, or strings, and may be used in close conjunction with other matrix and vector routines. For virtually all routines, the sorted array is assumed to be in ascending order.

Some functions are for the use of histograms or other specialized lists of sorted numbers.

Sorting functions

`void sortV(float *a,float *b,int n);`

`sortV` sorts vector `a` in order from smallest to largest value. The routine first checks for a forward or backward pre-sorted vector, and then, if necessary, does a heap sort using a routine nearly identical to that given in *Numerical Recipes*. `b` is rearranged in the same manner as `a`, but does not influence the sorting in any way. `b` may be either `NULL` or the same vector as `a`, if a dual vector is not required.

`void sortVii(int *a,int *b,int n);`
`sortVii` is identical to `sortV` except that all numbers are integers.

`void sortVdbl(double *a,double *b,int n);`
`sortVdbl` is identical to `sortV` except that all numbers are in double precision.

`void sortCV(float *a,float *bc,int n);`
`sortCV` is identical to `sortV` except that the required vector `bc` is a complex vector, with 2`n` elements that alternate real and imaginary components. `a` is still real.

`void sortVliv(long int *a,void **b,int n);`
This is identical to `sortV` except that `a` is an array of long integers and `b` is an array of `void*`s. Also, `b` is required.

`void sortVoid(void **a,int n,int (*compare)(void*,void*));`
Sorts a single vector of void*s, entered in `a`, of which there are `n` elements. There is no dual vector. Two elements of `a` are compared with each other using the compare function, which needs to return -1 if the first argument is less than the second, 0 if they are equal, and 1 if the first is greater than the second. This function is essentially the same as `sortV`.

Item locating functions

`int locateV(float *a,float x,int n);`
`locateV` locates the largest element of `a` that is smaller than or equal to `x`, where `a` is a sorted array. It returns the index of that element. If `x` is smaller than any value in `a`, then –1 is returned; if `x` is larger than any value in `a`, then `n-1` is returned. If several elements of `a` are equal to each other and `x` is either equal to them or is between their value and the next higher value, then the element of the collection with the highest index is returned. This routine uses a bisection type routine, which is almost exactly copied from *Numerical Recipes*.

`int locateVdbl(double *a,double x,int n);`
`locateVdbl` is identical to `locateV` except that all numbers are in double precision.

`int locateVli(long int *a,long int x,int n);`
This is similar to `locateV`. Differences are that all numbers are long integers and the function only returns the index if an exact match is found. Otherwise, it returns -1.

`int locateVi(int *a,int x,int n,int mode);`
Similar to above functions, but for integers. If mode is 0, then the function returns -1 if `x` is not found, whereas if mode is 1, then the function returns the largest element of `a` that is smaller than or equal to `x`.

`int locateVstr(char **a,char *x,int n,int mode);`

Similar to above functions, but for strings. Also, if mode is 0, then the function returns -1 if x is not found, whereas if mode is 1, then the function returns the largest element of a that is smaller than or equal to x.

Interpolation functions

`float interpolate1(float *ax,float *ay,int n,int *j,float x);`

interpolate1 does linear interpolation at position x. ax and ay are input *x* and *y* vectors, where the *x* values are sorted in ascending order, and n is the number of elements in each vector (minimum of 1). The routine needs the closest smaller *x* value. If its index or an index close by, but below it, is known, then send that value in as *j. If the index is not known, then send in *j as -2 and the routine will locate it with locateV. Either way, the correct index is returned in *j (identical to locateV). If multiple elements of ax have the same value: if x equals that value, the corresponding ay with the highest index is returned, if x is less than them, the one with the lowest index is used for interpolation, and if x is greater than them, the one with the highest index is used for interpolation. Here is a typical code fragment using interpolate1, which takes a sorted data set and rewrites it using a different set of *x* values.

```
for(j=-2,i=0;i<n2;i++) y2[i]=interpolate1(x1,y1,n1,&j,x2[i]);
```

`double interpolate1dbl(double *ax,double *ay,int n,int *j,double x);`

interpolate1dbl is identical to interpolate1 except that all numbers are in double precision.

`float interpolate1Cr(float *ax,float *ayc,int n,int *j,float x);`

interpolate1Cr is identical to interpolate1 except that the vector ayc is a complex vector, with 2n elements that alternate real and imaginary components. The function returns the interpolated value of the real components.

`float interpolate1Ci(float *ax,float *ayc,int n,int *j,float x);`

interpolate1Ci is identical to interpolate1Cr except that the function returns the interpolated value of the imaginary components.

`double cubicinterpolate1D(double *xdata,double *ydata,int n,double x);`

Performs one-dimensional interpolation or extrapolation on tabulated data using an interpolating polynomial. Enter the tabulated *x*-values in xdata, the tabulated *y*-values in ydata, and the sizes of these vectors as n. This function requires n to be at least 4 and will return -1 if this is not the case. Enter the desired *x* value in x. This finds the four data points that are closest to x, calculates the interpolating polynomial for them, and then calculates and returns the *y* value that corresponds to x. The xdata vector does not need to be uniformly spaced, although it must increase monotonically.

This function was copied from SurfaceParam.c, interpolate1D. The following text about a previous version of this function was copied from rxnparam_doc.doc:

Interpolation is done in a few functions here with simple cubic interpolation. This might be formally identical to the cubic spline algorithm that is described in *Numerical Recipes in C*, although it is a completely different implementation and so might lead to different results. My implementation is probably not as fast for long lists of input values, but has more transparent code, is fully contained within a single function, and is probably nearly as fast as their method for only a few input values.

Consider a input vectors X_i and Y_i, which define the known values and the input value x for which the unknown y is wanted. First, the position of x in the X_i vector is found, with the four X_i values that surround x are copied over into x_0 to x_3 such that x is between x_1 and x_2. Then, Lagrange's formula (see Numerical Recipies in C), is used to define the four polynomial coefficients as:

$$z_0 = \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} = -\frac{(x-x_1)(x-x_2)(x-x_3)}{6\Delta x^3}$$

$$z_1 = \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} = \frac{(x-x_0)(x-x_2)(x-x_3)}{2\Delta x^3}$$

$$z_2 = \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} = -\frac{(x-x_0)(x-x_1)(x-x_3)}{2\Delta x^3}$$

$$z_3 = \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} = \frac{(x-x_0)(x-x_1)(x-x_2)}{6\Delta x^3}$$

The first form allows unequally spaced x values, whereas the latter form assumes constant spacing on x of Δx. For interpolation on just one axis, the result is

$$y = z_0 y_0 + z_1 y_1 + z_2 y_2 + z_3 y_3$$

It doesn't matter if the input x value is too close to an edge of the tabulated data for it to be centered among 4 values, because exactly the same method is done if it is just one value in from the edge, or if extrapolation is required to go beyond the edge. For interpolation on two axes, interpolation is done four times on one axis to find 4 new y values; then it is done once on the other axis with these four y values. This is fairly obvious if a grid is drawn. While I haven't proven it, I suspect that the exact same result is gotten regardless of which axis is interpolated on first.

```
double cubicinterpolate2D(double *xdata,double *ydata,double *zdata,int nx,int
    ny,double x,double y);
```

Performs two-dimensional interpolation or extrapolation on tabulated data using an interpolating polynomial. The two independent variables are listed in the vectors

xdata and ydata, which have lengths nx and ny, respectively. The dependent variable is the table zdata, which has nx rows and ny columns (i.e. *y* is the fast-changing index). zdata needs to be a single array, such that zdata[i*ny+j] is the element at row i and column j. Enter the desired coordinate in x and y. This finds the 4x4 grid of xdata and ydata points that surround the desired (*x*,*y*) coordinate and interpolates on both axes to estimate the corresponding *z* value. If either nx or ny is less than 4, this returns the error code of -1.

Internally, this first calculates the polynomial coefficients for *x* (columns). Then, it uses these to interpolate a *z* value for each of the four nearest tabulated *y* values (i.e. for each row). Then, it calculates the polynomial coefficients for *y* and combines them with the 4 interpolated *z* values to yield a final interpolated *z* value, which is returned.

This function was copied from SurfaceParam.c, interpolate2D. See discussion above for cubicinterpolate1D for the method used.

Fitting functions

```
double fitoneparam(double *xdata,double *ydata,int nlo,int nhi,int
    function,double *constant);
```

This simple function performs linear least squares fits to data with functions that have only a single fitting parameter, although potentially several constants. Enter the *x*-values of the function in xdata and the *y*-values in ydata. Enter the indicies of the data that you want to fit using the nlo and nhi bounds (to fit all of the data, set nlo to 0 and nhi to the number of elements in the vectors). Data element nlo is included, while data element nhi is not included. Enter the function code number in function; the current available functions are listed in the table below. If you want to include one or more constant parameters in the fitting function that differs from the defaults listed below, enter it or them in constant. On the other hand, if you want to use the listed defaults, just enter constant as NULL.

function	function	defaults
1	$a+c_0$	$c_0 = 0$
2	$ax+ c_0$	$c_0 = 0$
3	$a/x+ c_0$	$c_0 = 0$

The math behind this function is based on the "Modeling of Data" chapter of *Numerical Recipies in C*. In this case, we are fitting to a single basis function $X(x_i)$. To minimize the function

$$\chi^2 = \sum_{i=nlo}^{nhi-1} \left[y_i - a_0 X(x_i) \right]^2 ,$$

we calculate

$$\alpha = \sum_{i=nlo}^{nhi-1} \left[X(x_i) \right]^2 \qquad \text{and} \qquad \beta = \sum_{i=nlo}^{nhi-1} \left[y_i X(x_i) \right]$$

and then

$$a_0 = \frac{\beta}{\alpha} .$$

Resampling functions

`void convertxV(float *ax,float *ay,float *cx,float *cy,int na,int nc);`
`convertxV` takes a sorted *x*,*y* data set in `ax` and `ay` and interpolates the data to a different set of *x* values, from `cx`, outputing the result to `cy`. `ax` and `cx` should be sorted beforehand. This just does what the code fragment above shows, except that this routine is a little faster and easier to use. Also, it checks first to see if all terms in `cx` are equal to those in `ax`, in which case the data are copied directly. `n` needs to be at least 2.

`void convertxCV(float *ax,float *ayc,float *cx,float *cyc,int na,int nc);`
`convertxCV` is identical to `convertxV` except that `ayc` and `cyc` are complex vectors with 2`na` and 2`nc` elements respectively.

Histogram functions

`void setuphist(float *hist,float *scale,int hn,float low,float high);`
Sets up arrays for a histogram. `hist` will contain the histogram data, and has size `hn`. `scale` is the histogram `scale`, which also has size `hn`. The histogram will be set up for the range from `low` to `high`, plus a bin at each end for values that are below `low` and that are above `high`. For example, consider 5 bins from 0 to 10 in steps of 2: bin 0 is for (–∞,0), bin 1 is for [0,2), bin 2 is for [2,4), …, bin 5 is for [8,10), and bin 6 is for [10,∞). This would be setup with `low` as 0, `high` as 10, and `hn` as 7. Both `hist` and `scale` would need to be pre-allocated to size 7. `hist` would be returned with all 0s and `scale` would be returned with the maximum value for each bin: 0, 2, 4, 6, 8, 10, `FLT_MAX`.

`void setuphistdbl(double *hist,double *scale,int hn,double low,double high);`
Identical to `setuphist`, but for doubles instead of floats.

`int histbin(float value,float *scale,int hn);`
Returns the bin number of a histogram that corresponds to `value`. `scale` and `hn` are as in `setuphist`.

`int histbindbl(double value,double *scale,int hn);`
Identical to `histbin`, but for doubles instead of floats.

`void data2hist(float *data,int dn,char op,float *hist,float *scale,int hn);`

Takes an unsorted list of dn data values in data and sorts them into histogram bins in hist. If op is '=', any prior histogram counts are cleared from hist; if op is '+', these data are added to any prior counts; and if op is '-', these data are subtracted from any prior counts. There are hn total histogram bins with upper bounds given with scale, exactly as defined as in setuphist.

Event analysis functions

```
double maxeventrateVD(double *event,double *weight,int n,double sigma,double
    *tptr);
```

maxeventrateVD inputs an unsorted vector of event times in event, which has n elements and has respective weights in weight. It finds the time at which the rate of weighted events was highest, where this is the time that maximizes $\sum_i w_i/[\sigma\sqrt{(2\pi)}]$ $\exp[-(t-a_i)^2/(2\sigma^2)]$. Send in weight equal to NULL if all events are to be weighted equally. This is a Gaussian smoothing of the delta functions that represent the events with standard deviation σ = sigma. The rate is returned. If tptr is not input as NULL, the time of the maximum rate is returned in tptr. Note that the maximum event rate depends on σ. This scans the whole event list and then does two more scans over progressively narrower regions. A faster but less reliable algorithm would use a binary search for the zero of the derivative.

Concentration profile functions

```
double *cpxinitializer(int n,double *r,double rlow,double rhigh,double rjump);
```

Initializes **r** vector of independent radius values for any dimension system. Enter r with NULL if memory should be allocated or with an existing vector if it should be overwritten. Returns **r**, or NULL if memory could not be allocated. **r** is set up with n evenly spaced values, of which the first value, r[0], equals rlow and the last one, r[n-1], is approximately equal to rhigh. If rjump is between rlow and rhigh, this function adds extra points around rjump to enable high precision around a function discontinuity. The result is symmetric about rjump, but does not include rjump. This symmetry was required for studying 1-D adsorption. In this case, the last point is roughly but not exactly equal to rhigh. To disable this extra precision, set rjump larger than rhigh; in this case, the last point will exactly equal rhigh.

For example, if one is analyzing a 3-D Smoluchowski reaction system with binding radius of 1, then a typical function call would be cpxinitializer(200,NULL,0,10,1);.

This function was modified from code in rxnparam.c, rdfmaketable.

```
double *cpxinitializec(double *r,double *c,int n,double *values,int code);
```

Initializes concentration vector, **c**, for any dimensional system. Enter c with NULL if memory should be allocated or with an existing vector if it should be overwritten.

Returns **c** or `NULL` if memory could not be allocated. Enter `r` with the radius values, `n` with the number of data points, `values` with the values listed below, and `code` with the desired initialization type.

code	result
0	`c[i] = 0` for all `i`
1	`c[i] = values[0]` for all `i`
else	same as code of 0

```
double cp1integrate(double *r,double *rdf,int n,double rmin,double rmax,int
    upperleft);
```

Integrates a 1-dimensional concentration profile, in `rdf` with *r*-values in `r`, from `rmin` to `rmax`. The trapezoid rule is used. The integral is returned. If `upperleft` is 1 (and `rmax` is less than `r[n-1]`), the sum includes only the upper left triangle of the last trapezoid and this last trapezoid extends past `rmax` to the next tabulated data point. `rmin` and `rmax` can be any finite values, including negative values, with `rmin` > `rmax`, etc. If one or both limits are outside of the tabulated *r*-values, then the function is assumed to be constant between these values and the nearest tabulated *r*-value. If `rmin` and/or `rmax` do not equal a tabulated *r*-value, then the integral includes the correct fraction of the end trapezoids.

The trapezoid rule is simple in this one-dimensional case. The area of the trapezoid that has *r*-values from r_0 to r_1, and respective heights f_0 and f_1, is

$$A = \frac{f_0 + f_1}{2}\left(r_1 - r_0\right)$$

For each trapezoid, the function calculates r_0, f_0, r_1, and f_1. For terminal trapezoids that are between the tabulated points j–1 and j, linear interpolation is used with the equation

$$y = y_{j-1} + \left(y_j - y_{j-1}\right)\frac{r - r_{j-1}}{r_j - r_{j-1}}$$

For the last trapezoid if `upperleft` is 1, then the area is simply

$$A = \frac{f_0}{2}\left(r_1 - r_0\right)$$

Complications arise in the code because it allows `rmin` and `rmax` to both be between the same pair of *r*-values (in same trapezoid), and/or to be above or below all of the *r*-values.

```
double cp2integrate(double *r,double *rdf,int n,double rmin,double rmax,int
    upperleft);
```

Integrates a 2-dimensional concentration profile, in `rdf` with r-values in `r`, from `rmin` to `rmax`. The trapezoid rule is used. The integral is returned. If `upperleft` is 1 (and `rmax` is less than `r[n-1]`), the sum includes only the upper left triangle of the last trapezoid and this last trapezoid extends past `rmax` to the next tabulated data point. `rmin` and `rmax` can be any non-negative values, including with `rmin` > `rmax`. If one or both are outside of the tabulated r-values, then the function is assumed to be constant between these values and the nearest tabulated r-value. If `rmin` and/or `rmax` do not equal a tabulated r-value, then the integral includes the correct fraction of the end trapezoids.

Integration uses a circular version of the trapezoid rule: at positions r_0 and r_1, the function f has values f_0 and f_1, leading to the linear interpolation

$$f = \frac{(r - r_0) f_1 + (r_1 - r) f_0}{r_1 - r_0}$$

$$A = \int_{r_0}^{r_1} 2\pi r f(r)\, dr$$

$$= \frac{\pi}{3}(r_1 - r_0)\left[f_0(2r_0 + r_1) + f_1(2r_1 + r_0) \right]$$

For the last trapezoid if `upperleft` is 1, then the area is

$$A == \frac{\pi}{3}(r_1 - r_0) f_0(2r_0 + r_1)$$

This function is identical to `cp1integrate`, except for this 2-D trapezoid calculation.

```
double cp3integrate(double *r,double *rdf,int n,double rmin,double rmax,int
    upperleft);
```

Integrates a 3-dimensional concentration profile, in `rdf` with r-values in `r`, from `rmin` to `rmax`. The trapezoid rule is used. The integral is returned. If `upperleft` is 1 (and `rmax` is less than `r[n-1]`), the sum includes only the upper left triangle of the last trapezoid and this last trapezoid extends past `rmax` to the next tabulated data point. `rmin` and `rmax` can be any non-negative values, including with `rmin` > `rmax`. If one or both are outside of the tabulated r-values, then the function is assumed to be constant between these values and the nearest tabulated r-value. If `rmin` and/or `rmax` do not equal a tabulated r-value, then the integral includes the correct fraction of the end trapezoids.

Integration uses a spherical version of the trapezoid rule: at positions r_0 and r_1, the function f has values f_0 and f_1, leading to the linear interpolation

$$f = \frac{(r - r_0) f_1 + (r_1 - r) f_0}{r_1 - r_0}$$

$$A = \int_{r_0}^{r_1} 4\pi r^2 f(r)\,dr$$
$$= \frac{4\pi}{r_1 - r_0}\left[\frac{(f_1 - f_0)(r_1^4 - r_0^4)}{4} + \frac{(r_1 f_0 - r_0 f_1)(r_1^3 - r_0^3)}{3}\right]$$
$$= \pi(f_1 - f_0)(r_1 + r_0)(r_1^2 + r_0^2) + \frac{4\pi}{3}(r_1 f_0 - r_0 f_1)(r_1^2 + r_1 r_0 + r_0^2)$$

For the last trapezoid if `upperleft` is 1, then the area is

$$A == \pi(-f_0)(r_1 + r_0)(r_1^2 + r_0^2) + \frac{4\pi}{3}(r_1 f_0)(r_1^2 + r_1 r_0 + r_0^2)$$

This function is identical to `cp1integrate`, except for this 3-D trapezoid calculation.

```
void cp1diffuse(double *r,double *rdfa,double *rdfd,int n,double rmsstep,double
     cminfinity,double cinfinity);
```

Diffuses a concentration profile, in `rdfa`, over a fixed time step, returning the result in `rdfd`. `r` is the vector of radius values and `n` is the number of elements in `r`, `rdfa`, and `rdfd`. `rmsstep` is the mean diffusion distance during this time step, which equals $(2D\Delta t)^{1/2}$, where D is the diffusion coefficient and Δt is the time step. This function assumes that all input rdf values below the lowest tabulated value equal `cminfinity` and all input rdf values above the highest tabulated value equal `cinfinity`.

This integrates the product of `rdfa` and the Green's function for simple diffusion to implement diffusion. The equation for the integral is:

$$d(r) = \int_{-\infty}^{\infty} a(r')\,\mathrm{grn}(r,r')\,dr'$$
$$\mathrm{grn}(r,r') = G_s(r - r')$$
$$G_s(r) = \frac{1}{s\sqrt{2\pi}}\exp\left(-\frac{r^2}{2s^2}\right)$$

For $r' <$ `r[0]`, the integral is

$$\int_{-\infty}^{r_0} C_{-\infty}\cdot \mathrm{grn}(r,r')\,dr' = \frac{C_{-\infty}}{2}\left(\mathrm{erfc}\,\frac{r - r_0}{s\sqrt{2}}\right)$$

For `r[0]` $\le r' \le$ `r[n-1]`, the integral is performed with the trapezoid rule, as shown in `cp1integrate`. For $r' >$ `r[n-1]`, the integral is

$$\int_{r_{n-1}}^{\infty} C_\infty \cdot \mathrm{grn}(r,r')dr' = \frac{C_\infty}{2}\left(1+\mathrm{erf}\frac{r-r_{n-1}}{s\sqrt{2}}\right)$$

```
void cp3diffuse(double *r,double *rdfa,double *rdfd,int n,double rmsstep,double
cinfinity);
```

Diffuses a concentration profile, in `rdfa`, over a fixed time step, returning the result in `rdfd`. `r` is the vector of radius values and `n` is the number of elements in `r`, `rdfa`, and `rdfd`. `rmsstep` is the mean diffusion distance during this time step, which equals $(2D\Delta t)^{1/2}$, where D is the diffusion coefficient and Δt is the time step. This function assumes that all input rdf values below the lowest tabulated value equal `rdfa[0]` and that the input rdf approaches `cinfinity` on the high side according to the extrapolation function `cinfinity`+a_2/r, where this function fits a_2 to the final 10% of the data.

This integrates the product of the radial distribution function with the Green's function for radially symmetric diffusion. The equation for this integral is

$$d(r) = \int_0^{\infty} 4\pi r'^2 a(r')\,\mathrm{grn}(r,r')dr'$$

$$\mathrm{grn}(r,r') = \frac{1}{4\pi r r'}\left[G_s(r-r') - G_s(r+r')\right]$$

$$G_s(r) = \frac{1}{s\sqrt{2\pi}}\exp\left(-\frac{r^2}{2s^2}\right)$$

This function scales input data in a couple of ways before processing it, which both simplifies the function some and should improve its accuracy. First, all values with length units (r, r', a_2, etc.) are divided by s (`rmsstep`) to give unitless results. Second, the input rdf is offset by its local value and then divided by C_∞ to make it unitless as well and, moreover, to reduce round-off errors. These scalings are removed as the output rdf is calculated so as to remove their effects. With the scalings, the equations are:

$$d(r) = a(r) + C_\infty \int_0^{\infty} 4\pi r'^2 \frac{a(r')-a(r)}{C_\infty}\mathrm{grn}(r,r')dr'$$

$$\mathrm{grn}(r,r') = \frac{1}{4\pi r r'}\left[G_1(r-r') - G_1(r+r')\right]$$

$$= \frac{1}{4\pi\sqrt{2\pi} r r'}\left\{\exp\left[-\frac{(r-r')^2}{2}\right] - \exp\left[-\frac{(r+r')^2}{2}\right]\right\}$$

Consider the latter half of the function first, where `rr` = `r[i]/rmsstep` = $r/s > 0$. The first trapezoid of the integral goes from `r0` = $r'/s = 0$ to `r1` = $(r'+dr)/s$ = `r[0` or `1]/rmsstep`. Because $r' = 0$, the Green's function is

$$\mathrm{grn}(r,0) = \frac{1}{2\pi\sqrt{2\pi}}\exp\left(-\frac{r^2}{2}\right)$$

Subsequent trapezoids follow the basic equations shown previously. The final portion of the integral is

$$\int_{r_1}^{\infty} 4\pi r'^2 \frac{\left(C_\infty - \frac{a_2}{r'}\right) - a(r)}{C_\infty}\mathrm{grn}(r,r')\,dr'$$

This is calculated in my Cambridge notes p. B-7.61, using unscaled parameters, as

$$\int_{r_1}^{\infty} 4\pi r'^2\left(C_\infty + \frac{a_2}{r'}\right)\mathrm{grn}(r,r')\,dr'$$

$$= 4\pi C_\infty \sigma^2 r_1 \mathrm{grn}(r,r_1) + \frac{C_\infty}{2}\left(\mathrm{erfc}\frac{r_1 - r}{\sigma\sqrt{2}} + \mathrm{erfc}\frac{r_1 + r}{\sigma\sqrt{2}}\right) + \frac{a_2}{2r}\left(\mathrm{erfc}\frac{r_1 - r}{\sigma\sqrt{2}} - \mathrm{erfc}\frac{r_1 + r}{\sigma\sqrt{2}}\right)$$

$$= C_\infty\left[4\pi r_1 \mathrm{grn}(r,r_1) + \frac{1}{2}\left(\mathrm{erfc}\frac{r_1 - r}{\sqrt{2}} + \mathrm{erfc}\frac{r_1 + r}{\sqrt{2}}\right)\right] + \frac{a_2}{2r}\left(\mathrm{erfc}\frac{r_1 - r}{\sqrt{2}} - \mathrm{erfc}\frac{r_1 + r}{\sqrt{2}}\right)$$

The final equality assumes $\sigma = 1$.

Next, move on to the first half of the function, which is executed only for `rdfd[0]` and only if `r[0]` = 0. This portion of the function is just like the latter portion, except that the Green's functions incorporate the fact that $r = 0$ here. The Green's function could be calculated for the first trapezoid where $r' = 0$ (it equals $(2\pi)^{-3/2}$), but it isn't because it's irrelevant. This is because `f1` ~ `rdfa[0]-rdfa[i]`, which equals 0. For subsequent trapezoids, the Green's function is

$$\mathrm{grn}(0,r') = \frac{1}{2\pi\sqrt{2\pi}}\exp\left(-\frac{r'^2}{2}\right)$$

The final portion of the integral is

$$\int_{r_1}^{\infty} 4\pi r'^2\left(C_\infty + \frac{a_2}{r'}\right)\mathrm{grn}(0,r')\,dr' = 4\pi\left(C_\infty\frac{r_1}{\sigma} + \frac{a_2}{\sigma}\right)\mathrm{grn}(r,r_1)\sigma^3 + C_\infty\mathrm{erfc}\frac{r_1}{\sigma\sqrt{2}}$$

This is also from my Cambridge notes, p. B-7.61. This includes a correction made 9/29/11 that was not in the original rxnparam.c file due to a mathematical mistake. Before, I left out the a_2 term. It also includes a small equation fix due to its being entered incorrectly before. These prior mistakes are unlikely to affect the prior results due to the fact that this portion of the integral carries essentially 0 area.

`double cp1absorb(double *r,double *rdf,int n,double rabsorb);`
Integrates the `rdf` from `r[0]` to `rabsorb` (using the upper left final trapezoid method), sets all `rdf` points up to `rabsorb` equal to 0, and returns the integral.

`double cp2absorb(double *r,double *rdf,int n,double rabsorb);`
Integrates the `rdf` from 0 to `rabsorb` (using the upper left final trapezoid method), sets all `rdf` points up to `rabsorb` equal to 0, and returns the integral.

`double cp3absorb(double *r,double *rdf,int n,double rabsorb);`
Integrates the `rdf` from 0 to `rabsorb` (using the upper left final trapezoid method), sets all `rdf` points up to `rabsorb` equal to 0, and returns the integral.

`void cpxaddconc(double *r,double *rdf,int n,double amount,int profile,double r1,double r2);`
Adds (or subtracts) concentration from a concentration profile in `rdf`. The result depends on the `profile` code.

profile	behavior
0	adds `amount` to every `rdf` value
1	adds `amount` to every `rdf` value with `r[i]` < `r1`
2	adds `amount` to every `rdf` value with `r[i]` ≥ `r1`
3	adds `amount` to every `rdf` value with `r1` ≤ `r[i]` < `r2`

`void cpxmassactionreact(double *rdfa,double *rdfb,int n,double rate);`
Performs mass action bimolecular reactions on the concentrations in `rdfa` and `rdfb`, using rate `rate`. This function is very simple. At each radius value, it subtracts `rate*rdfa[i]*rdfb[i]` from both `rdfa[i]` and `rdfb[i]`. To use this function, `rate` should be your rate constant times the time step.