Documentation for Cn.h and Cn.c

Steven Andrews, © 2003

See the document “LibDoc” for general information about this and other libraries.

```
#include <math.h>
#define CMPXmag(a,b) (sqrt((a)*(a)+(b)*(b)))
#define CMPXang(a,b)
     ((b)?((a)?atan((b)/(a)):((b)>0?1.57079632679:4.71238898037)):((a)<0?3.1415
     9265358:0))

typedef struct {float r;float i;} complex;

float *makecmplx(float *ar,float *ai,float *c,int n);
float *real(float *a,float *cr,int n);
float *imag(float *a,float *ci,int n);
float *magnitude(float *a,float *cr,int n);
float *cmplxphase(float *a,float *cr,int n);
float *CompConj(float *a,float *c,int n);
float *rotateCV(float *a,float *c,int n,int p);
float *rotate2CV(float *a,float *c,int n,float phi);
float *multeikx(float *xr,float *a,float *c,int n,float k);
float *multCV(float *a,float *b,float *c,int n);
float *deriv2CV(float *a,float *c,int n,int p);
float *integCV(float *a,float *c,int n);
float FTStartDflt(float *xr,int n);
float *fourier(float *xr,float *a,float *kr,float *c,int nx,int nk,int isign);
float *realcosft(float *xr,float *ar,float *kr,float *cr,int nx,int nk);
float *hankel(float *xr,float *ar,float *kr,float *cr,int nx,int nk,int samp);
float *fft(float *xr,float *a,float *kr,float *c,int nn,int isign);
```

Requires: `<math.h>,"math2.h","Rn.h","RnSort.h"`
Example program: `PlotTest.c, Quantum.c`

History: Written 5/99, updated 1/00. Moderate testing. Works with Metrowerks C. Added magnitude and phase 1/25/02.

This library complements the Rn.c library with routines to manipulate complex vectors. Complex vectors need twice the storage space as real vectors, so allocate them with `allocV(2*n)`. The real part of element `i` is stored in address `a[2*i]` with `0≤i≤n-1` and the imaginary part is stored in the next element, `a[2*i+1]`. The notation for most function parameters is that input vectors are labeled `a` and `b`, and output vectors are called `c`. No letter following the parameter letter implies a complex vector, with size `2*n`, and an `r` or `i` implies a real or imaginary vector, with size `n`. The address of the result is also returned by the routines, solely for convenience in cascaded opertations. Many Rn.c routines are useful with complex vectors, such as `sumV` and `copyV`, but remember to use `2*n` for the vector size. A complex vector may also be treated as an `nx2` matrix.

Macros and structures

`complex` is potentially useful elsewhere for single complex numbers, but is not used anywhere in this library (or anywhere else currently).

`CMPXmag` returns the magnitude of a complex number, where `a` and `b` are the real and imaginary components, respectively.

`CMPXang` returns the complex angle of a complex number, where `a` and `b` are the real and imaginary components, respectively. It works correctly for either `a` or `b` equal to zero and returns 0 if both inputs are equal to 0.

Functions

Most of the functions are summarized in the following table, a few of which are explained more fully below. Listed are the sizes of the vectors that are expected. In general, the input vectors, `a` and `b`, may occupy the same space in memory as each other and as the result, `c`, for situations where they have the same sizes. Exceptions are shown below with an asterisk.

Function	a	b	c	operation
`makecmplx`	n	n	2*n	copies to form complex vector
`real`	2*n		n	real portion
`imag`	2*n		n	imaginary portion
`magnitude`	2*n		n	complex magnitude of values
`cmplxphase`	2*n		n	complex phase of values
`CompConj`	2*n		2*n	complex conjugate
`rotateCV`	2*n		2*n	rotate phase of elements by `p`*π/2
`rotate2CV`	2*n		2*n	rotate phase of elements by `phi`
`multeikx`	2*n		2*n	multiply elements by exp(i k xr_j)
`multCV`	2*n	2*n	n	multiply complex elements
`deriv2CV`	2*n *		2*n	second derivative
`integCV`	2*n *		2*n	integral starting from element 0
`fourier`	2*nx *		2*nk	slow fourier transform
`realcosft`	nx *		nk	slow cosine transform
`hankel`	nx *		nk	slow real hankel transform
`fft`	2*nn		2*nn	fast fourier transform

`makecmplx` combines a real and an imaginary vector, of size `n` each, and combines them to give a complex vector, of size `2*n`. Either or both of `ar` and `ai` may be `NULL`, in which case the real, imaginary part, or both parts of `c` are set to zeros.

`deriv2CV` returns the second derivative of a complex vector. If `p` is 1, then periodic boundary conditions are used for the endpoints; otherwise the endpoints are interpolated. See Rn.c documentation for more details.

`integCV` integrates a complex vector. See the documentation for Rn.c for more details.

`FTStartDft` is a tiny routine that returns the default starting point values for a fourier transform, from a list of real x values and the number of values. It uses the equations given below, but with some effort to minimize round-off error.

`fourier` computes the numerical result of the continuous fourier transform of a complex vector, using a straightforward summing (not FFT). `xr` is the input vector of x values, corresponding to the complex `a` data values, and has `nx` elements; `xr` must be uniformly spaced (this could be easily modified, if useful). `kr` is also required and is the independent variable of the transformed result, `c`. `kr` has `nk` elements, but they do not need to be equally spaced, nor do they need to satisfy any sampling or aliasing criteria. `isign` is the sign of the exponential, where I typically consider a negative sign to be for a forward transform and a positive one to be for an inverse transform, although the opposite convention is occaisionally used. The equation approximated is $c(k)=1/\sqrt{(2\pi)} \int a(x)\, e^{\pm ikx}\, dx$. By the basic nature of a discrete fourier transform over a finite domain, periodic boundary conditions are always assumed.

The actual equation executed is $c_i = \Delta x/\sqrt{(2\pi)}\ \sum_j a_j \exp(\pm i x_j k_i)$. See below for more discussion.

`realcosft` computes the real cosine transform from data. However, it is not as easy to use as a fourier transform due to the neccessity of getting the endpoint and spacing correct.

`hankel` computes the Hankel transform of a real vector, which is typically used to compute the radial Fourier transform of a two dimensional radially symmetric function. The approximated equation is $c(k)=\int a(x)\ J_0(kx)\ dx$, which is both the forward and inverse Hankel transform (J_0 is the J_0 Bessel function). By experimentation, it appears that repeated Hankel transforms using a simple discrete version of the equation don't yield the original function. Instead the routine implemented here integrates the function as directly as possible, using `samp` x steps per input x data point and interpolating as needed (`samp`=10 is reasonably good). Results are reasonably good for a half Gaussian peaked at the origin and less good for other things. The `xr` input vector needs to be sorted in ascending order.

`fft` computes the fast fourier transform of a complex vector. The parameters are the same as for fourier, except that `nn` is the number of both x and k values and it must be an integer power of 2. Much of this routine is copied from *Numerical Recipes*. `kr[0]` is used to request a starting point of the k vector. However, the actual starting point has to be an integer number of dk steps away from 0, so the nearest value is used, which is within $dk/2$ of the requested value. The `kr` vector is set correctly.

<u>Fourier transform discussion</u>

While the fourier transform routines used here are simple, the correct input k vector is remarkably confusing, due to discreteness and endpoint issues. Here are some general relations and suggested k vectors, all of which satisfy the Nyquist relation. As above, n is the number of points, which are numbered from 0 to n–1.

$(n-1)\Delta x = x_{\max}-x_{\min}$ $\qquad$ $(n-1)\Delta k = k_{\max}-k_{\min}$

$x_i = x_{\min}+i\Delta x \quad 0\le i\le(n-1)$ $\qquad$ $k_i = k_{\min}+i\Delta k \quad 0\le i\le(n-1)$

<u>$a(x)$ is non-symmetric</u>

$n_k=n_x=n$

$n\Delta x$ = 1 full period

$\Delta k=2\pi/n\Delta x$

$k_{\min}=0$, $k_{\max}=(n-1)\Delta k$

or if n even: $k_{\min}=-\pi/\Delta x$, $k_{\max}=\pi/\Delta x-\Delta k$

or if n odd: $k_{\min}=-\pi/\Delta x+\Delta k/2$, $k_{\max}=\pi/\Delta x-\Delta k/2$

<u>$a(x)$ is even, and only half of period is given (e.g. 0 to $x_{\max}$)</u>

The following assumes the lowest point of a is the reflection point.

$n_k=2n_x-1$

$n_k\Delta x$ = 1 full period

option 1: create mirror image for $a(x)$ for $-x_{\max}$ to 0, then transform as above

option 2: do cosine transform with double resolution, shown below

$\Delta k=2\pi/n_k\Delta x$

$k_{\min}=0$, $k_{\max}=(n_k-1)\Delta k$

or: $k_{\min}=-\pi/\Delta x+\Delta k/2$, $k_{\max}=\pi/\Delta x-\Delta k/2$

$c(k)$ will be entirely real.

$a(x)$ is odd, and only half of period is given (e.g. 0 to x_{max})

$n_k=2n_x-1$

$n_k\Delta x$ = 1 full period

option 1: create mirror image for $a(x)$ for $-x_{\text{max}}$ to 0, then transform as above

option 2: do sine transform with double resolution, shown below

$\Delta k=2\pi/n_k\Delta x$

$k_{\text{min}}=0$, $k_{\text{max}}=(n_k-1)\Delta k$

or: $k_{\text{min}}=-\pi/\Delta x+\Delta k/2$, $k_{\text{max}}=\pi/\Delta x-\Delta k/2$

$c(k)$ will be entirely imaginary.

$a(x)$ is even, and only half of period is given (e.g. $\Delta x/2$ to x_{max})

Now assume the lowest point of a is $\Delta x/2$ + the reflection point.

$n_k=2n_x$

$n_k\Delta x$ = 1 full period

option 1: create mirror image for $a(x)$ for $-x_{\text{max}}$ to $-\Delta x/2$, then transform as above

option 2: do cosine transform with double resolution, shown below

$\Delta k=2\pi/n_k\Delta x$

$k_{\text{min}}=0$, $k_{\text{max}}=(n_k-1)\Delta k$

or: $k_{\text{min}}=-\pi/\Delta x$, $k_{\text{max}}=\pi/\Delta x-\Delta k$

or if a is real, just do half the k values: $k_{\text{min}}=\Delta k/2$, $k_{\text{max}}=\pi/\Delta x-\Delta k/2$

This final one is recommended for `realcosft`.

$c(k)$ will be entirely real.