Documentation for Rn.h and Rn.c

Steven Andrews, © 2004
See the document “LibDoc” for general information about this and other libraries.

Header file

```
#ifndef __Rn_h
#define __Rn_h

#include <stdlib.h>
#include <stdio.h>
#define allocV(n) ((float *) calloc(n,sizeof(float)))
#define allocM(m,n) ((float *) calloc((m)*(n),sizeof(float)))
#define freeV(a) free(a)
#define freeM(a) free(a)
#define itemV(a,i) (a[i])
#define itemM(a,n,i,j) (a[(n)*(i)+(j)])
#define setitemV(a,i,x) ((a[i])=(x))
#define setitemM(a,n,i,j,x) ((a[(n)*(i)+(j)])=(x))

int makeV(float *c,int n,char *s);
int makeM(float *c,int m,int n,char *s);
float *setstdV(float *c,int n,int k);
float *setstdM(float *c,int m,int n,int k);
float *DirCosM(float *c,float theta,float phi,float chi);
double *DirCosMD(double *c,double theta,double phi,double chi);
float *DirCosM2(float *c,float theta);
double *DirCosM2D(double *c,double theta);
float *printV(float *a,int n);
double *printVD(double *a,int n);
float *fprintV(FILE *stream,float *a,int n);
double *fprintVD(FILE *stream,double *a,int n);
float *printM(float *a,int m,int n,char s[]);
float *sprintM(float *a,int m,int n,char s[],char *t,int tn);
float minV(float *a,int n);
float maxV(float *a,int n);
double maxVD(double *a,int n,int *indx);
double minVD(double *a,int n,int *indx);
int equalV(float *a,float *b,int n);
int isevenspV(float *a,int n,float tol);
float detM(float *a,int n);
float traceM(float *a,int n);
double traceMD(double *a,int n);
int issymmetricMD(double *a,int n);
float *columnM(float *a,float *c,int m,int n,int j);
float *copyV(float *a,float *c,int n);
double *copyVD(double *a,double *c,int n);
float *copyM(float *a,float *c,int m,int n);
float *leftrotV(float *a,float *c,int n,int k);
float *sumV(float ax,float *a,float bx,float *b,float *c,int n);
double *sumVD(double ax,double *a,double bx,double *b,double *c,int n);
float *sumM(float ax,float *a,float bx,float *b,float *c,int m,int n);
float *addKV(float k,float *a,float *c,int n);
```

```
float *multKV(float k,float *a,float *c,int n);
float *divKV(float k,float *a,float *c,int n);
float *multV(float *a,float *b,float *c,int n);
float *multM(float *a,float *b,float *c,int m,int n);
float *divV(float *a,float *b,float *c,int n);
float *divM(float *a,float *b,float *c,int m,int n);
float *transM(float *a,float *c,int m,int n);
float dotVV(float *a,float *b,int n);
double dotVVD(double *a,double *b,int n);
void crossVV(float *a,float *b,float *c);
void crossVVD(const double *a,const double *b,float *c);
float distanceVV(float *a,float *b,int n);
double distanceVVD(double *a,double *b,int n);
float normalizeV(float *a,int n);
float normalizeVD(double *a,int n);
float averageV(float *a,int n,int p);
float *dotVM(float *a,float *b,float *c,int m,int n);
float *dotMV(float *a,float *b,float *c,int m,int n);
double *dotMVD(double *a,double *b,double *c,int m,int n);
float *dotMM(float *a,float *b,float *c,int ma,int na,int nb);
double *dotMMD(double *a,double *b,double *c,int ma,int na,int nb);
float minorM(float *a,int n,char *row,char *col);
float invM(float *a,float *c,int n);
float *deriv1V(float *a,float *c,int n);
float *deriv2V(float *a,float *c,int n);
float *integV(float *a,float *c,int n);
float *smoothV(float *a,float *c,int n,int k);
float *convolveV(float *a,float *b,float *c,int na,int nb,int nc,int bz,float
     left,float right);
float *correlateV(float *a,float *b,float *c,int na,int nb,int nc,int bz,float
     left,float right);
int histogramV(float *a,float *h,float lo,float hi,int na,int nh);
int histogramVdbl(double *a,double *h,double lo,double hi,int na,int nh);

#endif
```

Requires: `<math.h>,<stdlib.h>,<stdio.h>,<string.h>,"math2.h","random.h"`
Example program: `LibTest.c`

Originally written 11/10/90, expanded 2/93. Substantially modified 9/10/98. Moderately tested. Modified slightly 1/00. Modified 10/01. Works with Metrowerks C. Updated documention and ported to Linux 10/18/01. Updated 1/8/02, 1/25/02, 3/19/02, 4/10/02. Added `fprintV` 3/6/03. Added `traceM` 7/22/03. Added histogram routines 4/04. Added `normalizeV` 12/2/06. Added `issymmetricMD` 11/19/07. Added crossVVD 3/15/24. Removed major bug from crossVV and crossVVD 10/21/24.

These routines do vector and matrix arithmetic and manipulation on arrays of floats. While previous versions of this library encoded the matrix or vector dimensions in the first element or two of the array, this implementation does not do this, neccessitating the passing of dimension information to most routines. Routines ending with a `V` are for vector operations, those ending with an `M` are for matrix operations, and `VM` and `MV` suffixes are for combined operations. Vectors

are indexed from 0 to n-1 with index i. Matrices have rows from 0 to m-1 with index i and columns from 0 to n-1 with index j. Routines don't check that the dimensions of the vectors or matrices are compatible, so this needs to be watched before calling them. There aren't any limits on the vector or matrix sizes. Every routine is supposed to work for arbitrarily large sizes and for sizes as small as 1 (although a size of 0 is generally not allowed). Vectors may be sent into matrix routines as either single row matrices or as single column matrices, without causing problems. Alternatively, matrices may be sent to vector routines using vector dimension m*n. The matrices and vectors a and b are input parameters, and the result, if there is one, is c. Unless otherwise specified, a pointer to c is returned to allow easy concatenation of routines (but this must be used carefully to ensure correct order of operation). Destination vectors and matrices need to be allocated for the proper size before these routines are called.

Macro routines

allocV allocates space and returns an uninitiallized vector with the requested dimension.
allocM allocates space and returns an uninitiallized matrix with the requested dimensions.
freeV frees a vector.
freeM frees a matrix.
itemV returns the vector element at position i. This routine and the following three are nice conceptually since it means that a calling program can ignore the vector or matrix implementation; however, I virtually never use them, but assign or read from the data structure directly.
itemM returns the matrix element at position i,j.
setitemV sets the element at position i to be x.
setitemM sets the element at position i,j to be x.

Functions

makeV assigns the elements of a vector to be the numbers in the string. Typically the string will have n numbers. Each number is assumed to be separated from the previous by a single space. Values are assigned sequentially from item 0 to n-1. Null, too short, too long, or partially non-numeric strings are all permitted; in those cases, the routine assigns all values that it can and initializes the rest to 0. The number of assigned values is returned.

makeM assigns the elements of a matrix to be the numbers in the string, as above. Typically the string will have m*n numbers. Values are assigned sequentially from (0,0) to (m-1,n-1), filling each row before proceeding to the next row.

setstdV initiallizes a vector to one of a couple standard selections, chosen with the k parameter. k=0 gives all 0's, k=1 gives all 1's, k=2 gives sequential numbering from 0 to n-1, k=3 gives uniformly distributed random values between 0 and 1 in each element, and a negative k value gives all 0's, except for the item at position -k, which is set to 1 (in this case -k must be less than n). It's not defined for k>3.

setstdM initiallizes a matrix to one of a couple standard selections. k=0 gives all 0's, k=1 gives the identity matrix (if the matrix isn't square, it pads the extra region with 0's), k=2 gives a matrix of all 1's, and k=3 gives a matrix of random numbers between 0 and 1. Other numbers are undefined.

`DirCosM` returns the 3x3 direction cosine matrix for the Euler angles `theta`, `phi`, and `chi`, using the Quantum mechanics convention (see Zare or Sakurai). The matrix should be allocated beforehand to be size 3x3. Input angles can be on (–∞,∞).

`DirCosM2` is similar to `DirCosM` but returns the 2x2 direction cosine matrix for the angle `theta`.

`printV` displays a vector to the standard output as a single row using standard float formatting. Since it is conceivable that printf won't work, this command returns 0 if it fails; otherwise it returns the vector address as usual.

`fprintV` is identical to `printV`, except that it prints to filestream `stream`.

`printM` displays a matrix to the standard output with the standard `mxn` shape. Formatting is the standard float style if `s=NULL` or `s=""`; otherwise it uses whatever formatting is requested with `s`, such as `"%1.3f "`, (this string is sent directly to the printf command). As above, this command returns 0 if it fails; otherwise it returns the matrix address.

`sprintM` is just like `printM` except that the matrix is sent to the string `t`, which has been allocated to size `tn`. If `t` isn't large enough to hold the output, then the output is truncated when `t` is full. Each individual formatted number must be 255 characters or less (i.e. don't use absurd `s` inputs).

`minV` returns the minimum value of a vector. It doesn't assume or check for any ordering of elements but requires that the vector have at least one element.

`maxV` is similar but returns the maximum value of a vector.

`maxVD` is the same as `maxV` but for doubles and it also returns the index of the maximum value in `indx`. `indx` is allowed to be `NULL` if the index isn't wanted.

`minVD` is the same as `maxVD` but for the minimum value of a vector.

`equalV` returns 1 if two vectors are identical and 0 otherwise.

`isevenspV` returns 1 if the values of `a` are evenly spaced, within `tol` fractional tolerance, and 0 otherwise. That is, it returns 1 if every `a[i]-a[i-1]` is equal to the average difference, plus or minus `tol` times the average. It returns 0 if `n<2`.

`detM` returns the determinant of a matrix, which is assumed to be square. It uses an internal recursive routine called `detpart`. The only failure mode is if memory for `n char`'s can't be allocated, in which the routine returns a 0 (which is indistinguishable from a legitimate singular matrix).

`traceM` returns the trace of a matrix, which is assumed to be square.

`issymmetricMD` returns 1 if the square matrix is symmetric and 0 if it is not.

Arithmetic and copying functions are summarized in the table below. For functions that use both `a` and `b`, they are always allowed to point to the same

section of memory. c is also allowed to be in the same space, unless there is a "*" in the table below. For example, it is permitted to square the elements of a vector and overwrite the original with multV(v,v,v,n), although a comparable routine won't work for a dot product.

Function	a	b	c	operation
columnM	m,n		m	copies j'th column (starting at 0)
copyV	n		n	copies vector
copyM	m,n		m,n	copies matrix
leftrotV	n		n	item shift or rotate, see below
sumV	n		n	itemwise sum of c=ax*a+bx*b
sumM	m,n		m,n	itemwise sum of c=ax*a+bx*b
addKV	n		n	itemwise sum c=k+a
multKV	n		n	itemwise product c=k*a
divKV	n		n	itemwise division c=k/a
multV	n	n	n	itemwise product c=a*b
multM	m,n	m,n	m,n	itemwise product c=a*b
divV	n	n	n	itemwise division c=a/b
divM	m,n	m,n	m,n	itemwise division c=a/b
transM	m,n *		n,m	transpose $c=a^T$
dotVV	n	n		returns dot product a.b
crossVV	3	3	3	returns cross product axb
crossVVD	3	3	3	returns cross product axb
distanceVV	n	n		returns $\sqrt{(a-b)^2}$
normalizeV	n			returns |a| and normalizes a
averageV	n			mean of elements to power p
dotVM	m *	m,n *	n	returns dot product c=a.b
dotMV	m,n *	n *	m	returns dot product c=a.b
dotMM	ma,na *	na,nb *	ma,nb	returns dot product c=a.b
invM	n,n *		n,n	inverse $c=a^{-1}$, see below
deriv1V	n>=3 *		n	first derivative c=da, see below
deriv2V	n>=3 *		n	second derivative $c=d^2a$, see below
integV	n>=1 *		n	integral $c=\int a$, see below
smoothV	n>=2k *		n	smoothing c=a*gauss, see below
convolveV	na *	nb *	nc	convolve c=a*b, see below

leftrotV rotates a vector to the left by k elements, where k may have any integer value. Negative k values yield rotation to the right.

averageV returns the average of the elements, where elements are first raised to the p power. This is fast for p values of –1, 0, 1, or 2, and uses the math.h routine pow for the rest. Elements should not be equal to zero if negative powers are used.

minorM, which is primarily intended for internal use by invM, returns the minor of a square matrix, where the rows and columns that are struck out are given in row and col. row and col are strings of 0's and 1's ('\0' and '\1'), one character per row or column, where 0 means to include that row or column in the cofactor and 1 means to ignore it. row and col should each have the same number of 0's; these strings are returned as they were sent in.

invM inverts a matrix, using a matrix of cofactors, which it finds from the minors. It also returns the determinant of the input matrix (which is reciprocal of the

determinant of the inverse). If the determinant is 0, the inverse is not attempted. This method is slow and inelegant, but straightforward. It is conceivable that there is insufficient memory for `invM` to allocate the strings `row` and `col`, in which case `invM` returns a 0.

`deriv1V` takes the first derivative of a vector, using no smoothing. At each point in `a`, a parabola is fit to that point and the two neighboring points (or the closest two other points in the case of the ends) and that point in `c` is given the first derivative of the parabola. It assumes that the points in the vector are spaced with unit distance. See `integV` for more discussion. Here are the equations:

```
c[0]=-1.5*a[0]+2.0*a[1]-0.5*a[2];
c[i]=(a[i+1]-a[i-1])/2.0;
c[n-1]=0.5*a[n-3]-2.0*a[n-2]+1.5*a[n-1];
```

`deriv2V` is identical to `deriv1V` except that it takes the second derivative. Here are the equations:

```
c[0]=a[0]+a[2]-2.0*a[1];
c[i]=a[i-1]+a[i+1]-2.0*a[i];
c[n-1]=a[n-3]+a[n-1]-2.0*a[n-2];
```

`integV` integrates a vector, again assuming equally spaced points. The value at a point is the sum of all previous points, plus half the value at that point. Thus, `c[0]=a[0]/2` and `c[n-1]=a[0]+…+a[n-2]+a[n-1]/2`. If a vector is integrated, then differentiated, the result is the original with minimal smoothing (identical to `smooth` routine with `k`=1). If a vector is differentiated, then integrated, the result is again the original with minimal smoothing, but also with a constant offset due to derivative extrapolation. To get the first derivative of a pair of vectors, which represent *x* and *y* values, the new *x* values are the old *x* values, and the new *y* values are the differentiated *y* values (dy) divided by the differentiated *x* values (dx). To get the second derivative, divide the second derivative y values (d^2y) by the squared dx values (dx^2). To integrate, multiply the *y* integral ($\int y$) by dx.

`smoothV` smooths a vector by convolving it with a normalized Pascal's triangle function (a discrete Gaussian). The parameter `k` gives the number of points on each side to include in the smoothing. For example, if `k`=2, then `c[i]=(a[i-2]+4*a[i-1]+6*a[i]+4*a[i+1]+a[1+2])/16`. At the ends, the smoothing just considers the points that are available. The number of points in `a` may be less than `2k`. It returns a `NULL` pointer if `k`<0 or if it can't allocate the memory for the smoothing function.

`convolveV` does a discrete vector convolution using the equation `c[i]=∑ a[i-j]*b[j-bz]` where the sum is conceptually from $-\infty$ to ∞. It is best to think of `a` as the input function and `b` as the convolution kernal, although they are largely interchangeable. Outside the defined domain, `b` is assumed to be 0 and `a` has the constant value `left` for indicies <0 and `right` for indicies ≥`na`. `bz` is the index of `b` that is considered to be the its "zero" point; this index may have any integer value, not just between 0 and `nb-1`. To use this function to approximate a continuous convolution, multiply the result by dx, where dx is the data point spacing for all three vectors. The lowest *x* value of the `c` vector is the same as that for the `a` vector (plus the *x* value of `bz`, which is typically 0).

`correlateV` does a distcrete cross-correlation of a pair of vectors, and, of course, can also be used for an auto-correlation. It is nearly identical to `convolveV`, except that the equation is `c[i]=∑ a[i+j]*b[j-bz]`. As above, to approximate a continuous correlation, multiply by *dx*. Neither this routine nor `convolveV` allow periodic boundary conditions, which might be worth adding.

`histogramV` creates a histogram in `h` for the vector of data `a`. These need to be pre-allocated to size `na` and `nh`, respectively. Each bin has width *w*=`(hi-lo)/(nh-1)`. Bin 0 has a count of all `a` elements less than `lo`, bin 1 counts elements between `lo` and `lo`+*w*, bin *j* counts elements between `lo`+*w*(*j*–1) and `lo`+*wj*, and bin `nh` counts elements between `hi`–*w* and `hi`. There is no count of elements above `hi`. However, the function returns the total number of elements that were counted, so `na` minus the return value is the number of elements larger than `hi`. All values of `h` are integers, although the vector is a `float*` to allow easy scaling afterwards.

`histogramVdbl` is identical to `histogramV` except that it is for doubles rather than floats.