ARL-TN-1200 ● APR 2024

Recursively Creating, Deleting, and Linearly Interpolating *K*-Dimensional Arrays in C++

by Robert J Yager

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-1200 ● APR 2024

Recursively Creating, Deleting, and Linearly Interpolating *K*-Dimensional Arrays in C++

Robert J Yager
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
April 2024	Technical Note	10/1/2022	9/30/2023

4. TITLE AND SUBTITLE
Recursively Creating, Deleting, and Linearly Interpolating *K*-Dimensional Arrays in C++

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Robert J Yager

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-TD Aberdeen Proving Ground, MD 21005	ARL-TN-1200

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
K-dimensional (*K*-D) arrays are arrays that require *K* indices to access array elements. The yKDArray namespace (described in this report) contains a set of recursive functions that can be used to create, delete, and linearly interpolate rectangular *K*-D arrays. Also included is a function for creating jagged 2-D arrays. The yKDArray namespace relies exclusively on standard C++ operators. However, examples make use of the yInterp namespace for performing binary searches, the yRandom namespace for generating pseudorandom numbers, and the yBmp namespace for creating images.

15. SUBJECT TERMS
Terminal Effects, multidimensional, arrays, C++, recursive, interpolation, linear

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	29

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Robert J Yager	(410) 278-6689

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

Acknowledgments

I would like to thank Dr George Vunni and Ms Nancy Simini for their editorial recommendations, which improved the quality of this report.

1. Introduction

Consider some arbitrary function $z(x)$, where the form of $z(x)$ is known. Assuming x is a measurable quantity, determining $z(x)$ is simply a matter of first measuring x, then calculating $z(x)$. However, what if the form of $z(x)$ is not known? A common approach is to measure both x and $z(x)$ repeatedly, and with sufficient resolution, until some function (often a polynomial or exponential) can be generated and used to approximate $z(x)$.

When the number of independent variables grows large, though, the process of generating a relatively simple function to approximate $z(x_0, x_1, \dots)$ can be challenging. An alternative approach is to measure $z(x_0, x_1, \dots)$ for a grid of independent variables, then perform an interpolation between grid intersections. This is the method described here. The approach has a significant downside: rather than storing a few function parameters, an entire grid of data points must be measured/generated and stored. The size of that grid can grow to be exceedingly large for large numbers of independent variables. Thus, it should be considered to be a method of last resort.

The grid of measured $z(x_0, x_1, \dots)$ values can be stored in a *K*-dimensional (*K*-D) array, where *K* is equal to the number of independent variables needed to calculate $z(x_0, x_1, \dots)$, and a *K*-D array is an array that requires *K* indices to access an array element. The yKDArray namespace (presented in this report) contains a set of recursive functions that can be used to create, delete, and linearly interpolate rectangular *K*-D arrays. Also included in this report is a function for creating jagged 2-D arrays.

The yKDArray namespace relies exclusively on standard C++ operations. However, examples make use of the yInterp namespace[1] for performing binary searches, the yRandom namespace[2] for generating pseudorandom numbers, and the yBmp namespace[3] for creating images. Code for the yInterp, yRandom, and yBmp namespaces is included as the Appendix.

Working in *K* dimensions is considerably more difficult than working in one dimension, or even in some higher, but fixed-size, dimension. When the number of dimensions is not known in advance, simple indices are no longer effective. Rather, indices themselves must have indices. If care is not taken to keep track of indices and indices with indices, errors in logic can easily hide in confusions between similar seeming, though actually different, variables.

Writing functions in C++ becomes challenging as well. Data is stored in arrays that do not have a set number of dimensions. So, functions need to be templated to allow

for pointers that are arbitrarily deep (double*, double**, double***, etc.). Furthermore, functions need to be recursive (i.e., they call themselves). Recursive functions inherently contain a level of abstraction that can be difficult to follow for those not accustomed to working with them. Furthermore, their compactness can be deceiving, possibly leading one to believe, at first glance, that a couple of apparently simple lines of code describe a simple algorithm. This is particularly true of the LinInterp() function presented in Section 3. The body of the function is a mere two lines long. However, multiple pages in this report (see Section 2) are dedicated to describing the algorithm behind those two lines of code.

This report is intended to be useful for a reader who is both familiar with the basics of programming in C++, and who has a need of either performing linear interpolations in *K*-dimensions or possibly simply storing data in *K*-D arrays. As such, details of the process of programming in C++ are mostly absent. Readers who wish to skip over the preliminary discussion and go straight to the code should feel free to skip ahead to Section 3, which presents the yKDArray namespace, and Sections 8–11, which present examples of its use.

2. Linear Interpolations

The equation for linearly interpolating in *K* dimensions is similar to the equation for linearly interpolating in one dimension, though with the added complexities of recursion and additional indices (see Eqs. 6 and 28). The following three subsections first present the process of interpolating in one dimension, followed by the process of generalizing to two dimensions, then, finally, the process of fully generalizing to *K* dimensions.

2.1 1-D Linear Interpolations

Suppose there exists some function $z(x)$, where the *form* of $z(x)$ is unknown, but *values* of $z(x)$ *are* known for certain values of x. Let those x and $z(x)$ values be stored in $\vec{X}$ and $\vec{Z}$, respectively. If n is the number of values of x for which $z(x)$ is known, and j is an indexing variable such that $0 \leq j < n$, then

$$\vec{X} \equiv \{X_0\,, X_1\,, \cdots, X_j\,, \cdots, X_{n\text{-}1}\,\} \tag{1}$$

and

$$\vec{Z} \equiv \{Z_0\,, Z_1\,, \cdots, Z_j\,, \cdots, \mathrm{Z}_{n\text{-}1}\,\} \tag{2}$$

where

$$Z_j \equiv z(X_j) \tag{3}$$

To simplify matters, assume that each X_j is unique and all X_j values are stored in increasing order. Thus,

$$X_j < X_{j+1} \tag{4}$$

Next, suppose we want to know the value of $z(x)$ for some value of x for which $z(x)$ is unknown. If $X_0 \leq x < X_{n-1}$, then we can approximate $z(x)$ by performing a linear interpolation, an example of which is shown in Fig. 1.

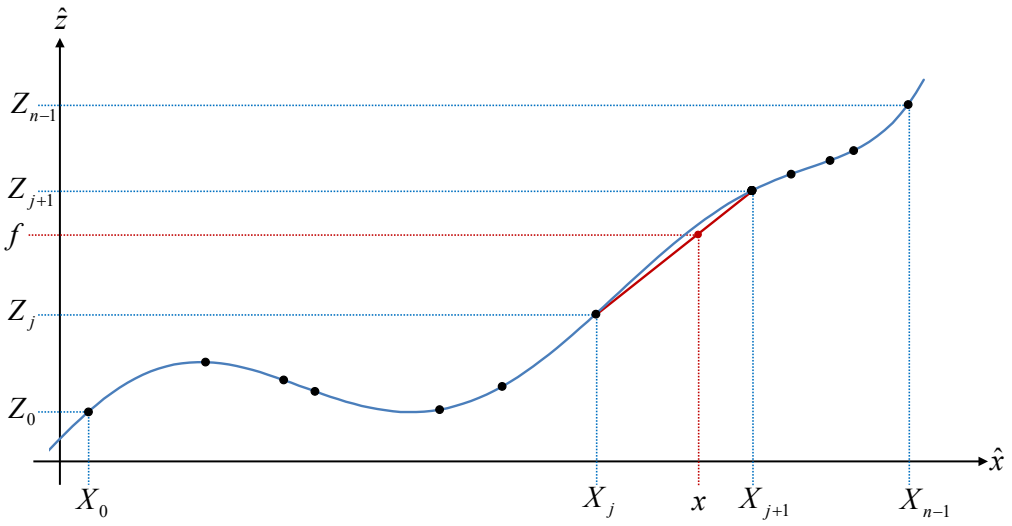

Fig. 1 $z(x)$ **(represented by a curved, blue line) and** $f(x)$ **(represented by a straight, red line)**

Begin by choosing j such that it satisfies the following inequality:

$$X_j \leq x < X_{j+1} \tag{5}$$

Then, if we let $f(x)$ represent a linear interpolation,

$$z(x) \cong f(x) = Z_j + (Z_{j+1} - Z_j)\frac{x - X_j}{X_{j+1} - X_j} \tag{6}$$

2.2 2-D Linear Interpolations

Suppose there exists some function $z(x_0,x_1)$, where the *form* of $z(x_0,x_1)$ is unknown, but *values* of $z(x_0,x_1)$ *are* known for certain values of x_0 and x_1. Let those x_0, x_1, and $z(x_0,x_1)$ values be stored in $\vec{X}_0$, $\vec{X}_1$, and Z, respectively, where Z is now a matrix. Let n_0 be the number of values of x_0 for which $z(x_0,x_1)$ is known, n_1 be the number of values of x_1 for which $z(x_0,x_1)$ is known, j_0 be an indexing variable such that $0 \leq j_0 < n_0$, and j_1 be an indexing variable such that $0 \leq j_1 < n_1$. Then

$$\vec{X}_0 \equiv \{X_{0,0}\,, X_{0,1}\,, \cdots, X_{0,j_0}\,, \cdots, X_{0,\,n_0\text{-}1}\} \tag{7}$$

$$\vec{X}_1 \equiv \{X_{1,0}\,, X_{1,1}\,, \cdots, X_{1,j_1}\,, \cdots, X_{1,\,n_1\text{-}1}\} \tag{8}$$

and

$$Z \equiv \begin{bmatrix} Z_{0,0} & \cdots & Z_{0,j_1} & \cdots & Z_{0,n_1-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ Z_{j_0,0} & \cdots & Z_{j_0,j_1} & \cdots & Z_{j_0,n_1-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ Z_{n_0-1,0} & \cdots & Z_{n_0-1,j_1} & \cdots & Z_{n_0-1,n_1-1} \end{bmatrix} \tag{9}$$

where

$$Z_{j_0,j_1} \equiv z(X_{0,j_0}, X_{1,j_1}) \tag{10}$$

To simplify matters, assume that each X_{0,j_0} value is unique and all X_{0,j_0} are stored in increasing order. Similarly, assume each X_{1,j_1} value is unique and all X_{1,j_1} values are stored in increasing order. Thus,

$$X_{0,j_0} \; < \mathrm{X}_{0,j_0+1} \tag{11}$$

and

$$X_{1,j_1} \; < X_{1,j_1+1} \tag{12}$$

Furthermore, assume that Z_{j_0,j_1} is known for every X_{0,j_0}, X_{1,j_1}pair.

Next, suppose we want to know the value of $z(x_0,x_1)$ for some x_0, x_1 pair for which $z(x_0,x_1)$ is unknown. If $X_{0,0} \le x_0 < X_{0,n_0\text{-}1}$ and $X_{1,0} \le x_1 < X_{1,n_1\text{-}1}$, then we can approximate $z(x_0,x_1)$ by performing three linear interpolations. Begin by choosing j_0 and j_1 such that they satisfy the following inequalities:

$$X_{0,\,j_0} \le x_0 < X_{0,\,j_0+1} \tag{13}$$

and

$$X_{1,\,j_1} \le x_1 < X_{1,\,j_1+1} \tag{14}$$

Let f_0 be an interpolation between Z_{j_0,j_1} and Z_{j_0,j_1+1} at $x_0 = X_{0,\,j_0}$:

$$f_0(x_1) = Z_{j_0,j_1} + \left(Z_{j_0,j_1+1} - Z_{j_0,j_1}\right)\frac{x_1 - X_{1,\,j_1}}{X_{1,\,j_1+1} - X_{1,\,j_1}} \tag{15}$$

Next, let f_1 be an interpolation between Z_{j_0+1,j_1} and Z_{j_0+1,j_1+1} at $x_0 = X_{0,\,j_0+1}$:

$$f_1(x_1) = Z_{j_0+1,j_1} + (Z_{j_0+1,j_1+1} - Z_{j_0+1,j_1})\frac{x_1 - X_{1,j_1}}{X_{1,j_1+1} - X_{1,j_1}} \tag{16}$$

Finally, interpolate between $f_0(x_1)$ and $f_1(x_1)$:

$$f(x_0, x_1) = f_0(x_1) + (f_1(x_1) - f_0(x_1))\frac{x_0 - X_{0,j_0}}{X_{0,j_0+1} - X_{0,j_0}} \tag{17}$$

A graphic representation of the process of interpolating in two dimensions is shown in Fig. 2, where the green lines represent f_0 and f_1 (the interpolations described by Eqs. 15 and 16) and the red line represents f (the interpolation described by Eq. 17).

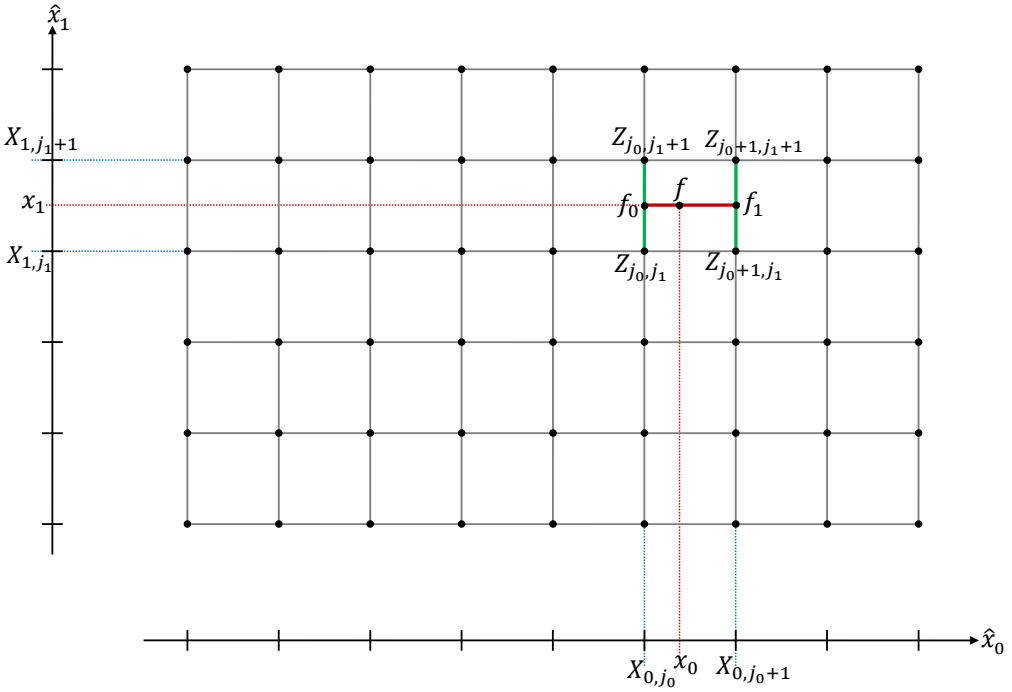

Fig. 2 Interpolating in two dimensions

2.3 *K*-D Linear Interpolations

Let $\vec{x}$ represent a K-element vector that stores some set of independent variables:

$$\vec{x} \equiv \{x_0, x_1, \cdots, x_k, \cdots, x_{K-1}\} \tag{18}$$

Suppose there exists some function $z(\vec{x})$, where the *form* of $z(\vec{x})$ is unknown, but *values* of $z(\vec{x})$ *are* known for certain values of $x_0, x_1, \ldots, x_k, \ldots, x_{K-1}$. Let those $x_0, x_1, \ldots, x_k, \ldots, x_{K-1}$ values be stored in $\vec{X}_0, \vec{X}_1, \ldots, \vec{X}_k, \ldots, \vec{X}_{K-1}$, respectively, and let $\vec{X}_0, \vec{X}_1, \ldots, \vec{X}_k, \ldots, \vec{X}_{K-1}$ be stored in X:

$$X \equiv \begin{bmatrix} \vec{X}_0 \\ \vdots \\ \vec{X}_{\mathrm{k}} \\ \vdots \\ \vec{X}_{\mathrm{K\text{-}1}} \end{bmatrix} = \begin{bmatrix} X_{0,0} & \cdots & X_{0,j_0} & \cdots & X_{0,n_0-1} \\ \vdots & & & & \\ X_{k,0} & \cdots & X_{k,j_k} & \cdots & X_{k,n_k-1} \\ \vdots & & & & \\ X_{K-1,0} & \cdots & X_{K-1,j_{K-1}} & \cdots & X_{K-1,n_{K-1}-1} \end{bmatrix} \quad (19)$$

where n_k is the number of values of x_k for which $z(\vec{x})$ is known, and j_k is an indexing variable such that $0 \leq j_k < n_k$. Thus,

$$\vec{n} \equiv \{n_0, n_1, \cdots, n_k, \cdots, n_{K\text{-}1}\} \quad (20)$$

and

$$\vec{j} \equiv \{j_0, j_1, \cdots, j_k, \cdots, j_{K\text{-}1}\} \quad (21)$$

The total number of elements in X is the sum of the n_k values:

$$M = \sum_{k=0}^{k<K} n_{\mathrm{k}} \quad (22)$$

In general, since the number of values in each $\vec{X}_k$ need not be equal, X will be a jagged 2-D array rather than a matrix.

To simplify matters, assume that for each $\vec{X}_k$, each X_{k,j_k} value is unique, and for each $\vec{X}_k$, all X_{k,j_k} values are stored in increasing order. Thus, for each $\vec{X}_k$,

$$X_{k,j_k} < \mathrm{X}_{k,j_k+1} \quad (23)$$

Let the $z(X)$ values be stored in Z, with elements $Z_{j_0,j_1,\cdots j_k,\cdots\, j_{K-1}}$. Thus,

$$Z_{j_0,j_1,\cdots,j_k,\cdots,j_{K-1}} \equiv z(X_{0,j_0}, X_{1,j_1}, \cdots, X_{k,j_k}, \cdots, X_{K-1,j_{K-1}}) \quad (24)$$

and Z is now a K-D array. Assume that $Z_{j_0,j_1,\cdots,j_k,\cdots,j_{K-1}}$ is known for every $X_{0,j_0}, X_{1,j_1}, \cdots, X_{k,j_k}, \cdots, X_{K-1,j_{K-1}}$ tuple. The total number of values in Z is the product of the n_k values:

$$N = \prod_{k=0}^{k<K} n_{\mathrm{k}} \quad (25)$$

Suppose we want to know the value of $z(\vec{x})$ for some $\vec{x}$ tuple for which $z(\vec{x})$ is unknown. If $X_{k,0} < x_k < X_{k,n_k\text{-}1}$ for all $0 \leq k < K$, then we can approximate $z(\vec{x})$ by performing $2^K - 1$ linear interpolations. Begin by choosing $J_k = j_k$ for all $0 \leq k < K$ such that they satisfy the following inequality:

$$\mathrm{X}_{k,\,J_k} \leq x_k < \mathrm{X}_{k,\,J_k+1} \quad (26)$$

Then

$$\vec{J} \equiv \{J_0, J_1, \cdots, J_k, \cdots, J_{K\text{-}1}\} \tag{27}$$

Equation 28 can be used to recursively perform K-D interpolations, which approximate $z(\vec{x})$:

$$f(Z, \vec{J}, \vec{x}, X) = \begin{cases} Z & \text{for } K = 0 \\ f\left(Z_{J_0}, \vec{J}', \vec{x}', X'\right) + & \\ \left[f\left(Z_{J_0+1}, \vec{J}', \vec{x}', X'\right) - f\left(Z_{J_0}, \vec{J}', \vec{x}', X'\right)\right] & \text{for } K > 0 \\ \times (x_0 - X_{0,J_0}) / \left(X_{0,J_0} - X_{0,J_0+1}\right) & \end{cases} \tag{28}$$

where $\vec{J}'$ and $\vec{x}'$ equal $\vec{J}$ and $\vec{x}$, respectively, but with the first element removed, and X' equals X, but with the first vector removed:

$$\vec{J}' = \{J_1, \cdots, J_k, \cdots, J_{K-1}\} \tag{29}$$

$$\vec{x}' = \{x_1, \cdots, x_k, \cdots, x_{K-1}\} \tag{30}$$

and

$$X' = \{\vec{X}_1, \cdots, \vec{X}_k, \cdots, \vec{X}_{K-1}\} \tag{31}$$

Furthermore, Z_{J_0} and Z_{J_0+1} are (K-1)-D arrays such that

$$Z_{J_0} \equiv Z_{J_0, j_1, \cdots, j_k, \cdots, j_{K-1}} \tag{32}$$

and

$$Z_{J_0 \mp 1} \equiv Z_{J_0 \mp 1, j_1, \cdots, j_k, \cdots, j_{K-1}} \tag{33}$$

Finally, by rearranging terms, Eq. 28 can be optimized so that $f()$ is only called twice (rather than three times) per recursion:

$$f(Z, \vec{J}, \vec{x}, X) = \begin{cases} Z & \text{for } K = 0 \\ [f(Z_{J_0}, \vec{J}', \vec{x}', X')(x_0 - X_{0,J_0+1}) & \\ -f(Z_{J_0+1}, \vec{J}', \vec{x}', X')(x_0 - X_{0,J_0})] / & \text{for } K > 0 \\ \left(X_{0,J_0} - X_{0,J_0+1}\right) & \end{cases} \tag{34}$$

3. y_kd_array.h

The following is the yKDArray namespace, presented in its entirety. Sections dedicated to describing each of the four functions that are contained in the yKDArray namespace follow, as do examples.

```
#ifndef YKDARRAY_GUARD//    See Yager, R.J. "Recursively Creating, Deleting, and
#define YKDARRAY_GUARD//     Linearly Interpolating K-Dimensional Arrays in C++"
namespace yKDArray{//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 template<class T>void New(T Z,int*n,int l){}//<==================DUMMY OVERLOAD
 template<class T>void New(//<===================ALLOCATE A RECTANGULAR K-D ARRAY
   T*&Z,//<-------------------------------------------A POINTER TO A K-D ARRAY
   int*n,//<------------ARRAY DIMENSIONS (n SHOULD POINT TO AT LEAST K ELEMENTS)
   int l=1){//<---------------------------------NOT A USER-MODIFIABLE PARAMETER
  for(New(*(Z=new T[l**n]),n+1,l**n);--l;*(&Z+l)=*&Z+*n*l);}
 template<class T>void New2D(//<============ALLOCATE A 2-D, NONRECTANGULAR ARRAY
   T**&Z,//<-----------------------------------------A POINTER TO A 2-D ARRAY
   int K,//<-----------------------------------------------NUMBER OF ROWS IN Z
   int*n){//<------------------ROW SIZES (n SHOULD POINT TO AT LEAST K ELEMENTS)
  int N=0;/*<-*/for(int i=0;i<K;++i)N+=n[i];
  Z=new T*[K],*Z=new T[N];/*<-*/for(int i=1,l=*n;i<K;l+=n[i++])Z[i]=*Z+l;}
 template<class T>void Delete(T Z){}//<============================DUMMY OVERLOAD
 template<class T>void Delete(//<================================DELETE AN ARRAY
   T*Z){//<----------------A POINTER TO AN ARRAY ALLOCATED WITH New() or New2D()
  Delete(*Z),delete[]Z;}
 template<class T>T LinInterp(T Z,int*j,T*x,T**X){return Z;}//<=========K=0 CASE
 template<class T,class U>U LinInterp(//<==============A K-D LINEAR INTERPOLATOR
   T Z,//<--------------------A POINTER TO AN ARRAY OF DEPENDENT-VARIABLE VALUES
   int*J,//<----------INTERPOLATION INDICES (FIND USING yInterp::BinarySearch())
   U*x,//<-----------------------------------KEY VALUES FOR INDEPENDENT VARIABLES
   U**X){//<----------STORED VALUES FOR INDEPENDENT VARIABLESS AT KNOWN Z-VALUES
  return(LinInterp(Z[*J],J+1,x+1,X+1)*(*x-X[0][*J+1])-
   LinInterp(Z[*J+1],J+1,x+1,X+1)*(*x-X[0][*J]))/(X[0][*J]-X[0][*J+1]);}
}//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#endif//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ YAGENAUT@GMAIL.COM
```

4. yKDArray::New()

yKDArray::New() can be used to allocate memory for rectangular *K*-D arrays, where all elements are stored in single block of contiguous memory. As a result, *K*-D arrays are intrinsically mapped to 1-D arrays, as is demonstrated in Sections 10 and 11.

yKDArray::New() can be used to create arrays of any data type. However, care should be taken when creating (and deleting) arrays of pointers.

4.1 yKDArray::New() Arguments

Z points to a *K*-D array of type-T elements.

n points to an array of integers that specifies the sizes of the dimensions of Z. For example, n = {4, 6} specifies the dimensions of a 4 × 6 matrix. The array that n points to must contain at least K elements.

l is not intended to be a user-modifiable parameter.

4.2 yKDArray::New() Example

The following example uses yKDArray::New() to create a 3-D array with each element set to equal the sum of its indices.

4.2.1 Example Code

```
#include<cstdio>//.....................................................printf()
#include"y_kd_array.h"//.............................yKDArray (see ARL-TN-1200)
int main(){//<==================EXAMPLE: CREATE AND DELETE A 3-DIMENSIONAL ARRAY
 int n[]={2,2,2};//.........................................array dimensions
 int***Z;/*<-*/yKDArray::New(Z,n);//...............................create array
 for(int i=0;i<*n;++i)for(int j=0;j<n[1];++j)for(int k=0;k<n[2];++k)
  printf("Z[%d][%d][%d] = %d\n",i,j,k,Z[i][j][k]=i+j+k);//......populate & print
 yKDArray::Delete(Z);}//.............................................cleanup
```

4.2.2 Example Output

```
Z[0][0][0] = 0
Z[0][0][1] = 1
Z[0][1][0] = 1
Z[0][1][1] = 2
Z[1][0][0] = 1
Z[1][0][1] = 2
Z[1][1][0] = 2
Z[1][1][1] = 3
```

5. yKDarray::New2D()

yKDArray::New2D() can be used to allocate memory for non-rectangular 2-D arrays. As with *K*-D arrays allocated using yKDArray::New(), all elements are stored in a single block of contiguous memory and, thus, are intrinsically mapped to 1-D arrays.

5.1 yKDArray::New2D() Arguments

Z points to a 2-D array of type-T elements.

K specifies the first-dimension size of Z, which is the same as the number of elements in the array that n points to.

n points to an array of integers, which specifies the sizes of the 1-D arrays that Z points to. For example, n = {4, 6} specifies an array where Z[0] points to a 1-D

array with four elements and Z[1] points to a 1-D array with six elements. The array that n points to must contain at least K elements.

5.2 yKDArray::New2D() Example

The following example uses yKDArray::New2D() to create a nonrectangular 2-D array with randomly chosen row sizes and populated with random values.

5.2.1 Example Code

```
#include <cstdio>//.....................................................printf()
#include"y_kd_array.h"//.............................yKDArray (see ARL-TN-1200)
#include"y_random.h"//.................................yRandom (see ARL-TN-613)
int main(){//<=============EXAMPLE: CREATE AND DELETE A NONRECTANGULAR 2-D ARRAY
 unsigned I[625];/*<-*/yRandom::Seed(I,1);//...........state of Mersenne twister
 int K=6,*n=new int[K];/*<-*/for(int i=0;i<K;++i)n[i]=yRandom::Integer(I,3,8);
 double**Z;/*<-*/yKDArray::New2D(Z,K,n);//...................create a 2-D array
 printf("Z=\n");
 for(int i=0;i<K;++i)for(int j=0;j<n[i];++j)printf("%8f%s",
  Z[i][j]=yRandom::Uniform(I,1,9),j==n[i]-1?"\n":",");//........populate & print
 delete[]n,yKDArray::Delete(Z);}//.....................................cleanup
```

5.2.2 Example Output

```
Z=
3.418661,8.992324,2.174047,2.888712,1.738709
4.172646,2.490082,4.103286,3.764486,6.357968,4.174140,8.484313,5.310534
7.770487,4.353556,3.506188,6.481756,5.196385,2.635618,4.547623
8.024939,2.836618,1.219101,5.275311,6.363740,8.311696,4.338438,4.657638
5.469519,4.445589,2.123095
8.513022,2.584812,7.227114
```

6. yKDarray::Delete()

yKDArray::Delete() deallocates memory that was previously allocated using either yKDArray::New() or yKDArray::New2D(). yKDArray::Delete() may be required to avoid memory leaks. All examples in this report use yKDArray::Delete() to clear previously allocated memory prior to program termination.

yKDArray::Delete() Arguments

Z points to an array that was previously allocated using either yKDArray::New() or yKDArray::New2D().

7. yKDArray::LinInterp()

yKDArray::LinInterp() performs linear interpolations in K dimensions. A K-D linear interpolation can be defined as a linear interpolation between the results of two (K-1)-D linear interpolations. Performing a K-D linear interpolation requires a total of 2^K-1 1-D linear interpolations.

7.1 yKDArray::Lininterp() Arguments

`Z` points to a *K*-D array that is used to specify *Z* (see Eq. 24).

`J` points to an array that is used to specify $\vec{J}$ (see Eq. 27). BinarySearch() from the yInterp namespace can be used to populate `J`.

`x` points to an array that is used to specify $\vec{x}$, the point at which the interpolation is performed (see Eq. 18).

`X` points to an array that is used to specify *X* (see Eq. 19). The elements of each row of `X` must be both unique and stored in increasing order (see the inequality presented in Eq. 23).

7.2 yKDArray::Lininterp() Example

The following example code uses yKDArray::LinInterp() to interpolate a 3-D array that is based on Eq. 35.

$$z(\vec{x}) = e^{-\sum_{i=0}^{i<K} x_i^2} \tag{35}$$

BinarySearch() from the yInterp namespace is used to find $\vec{J}$. Output is calculated for both *z* and *f* at $\vec{x}$ = {0.32, 0.46, -0.11}.

7.2.1 Example Code

```
#include<cstdio>//.....................................................printf()
#include"y_interp.h"//.................................yInterp (see ARL-TN-0657)
#include"y_kd_array.h"//..............................yKDArray (see ARL-TN-1200)
int main(){//<================================EXAMPLE: 3-D LINEAR INTERPOLATIONS
 int n[]={11,11,11},j[3],J[3];//...............array dimensions & array indices
 double**X;/*<-*/yKDArray::New2D(X,3,n);//...........create & polulate "X" array
 for(int i=0;i<3;++i)for(j[i]=0;j[i]<n[i];++j[i])X[i][j[i]]=-.5+.1*j[i];
 double***Z;/*<-*/yKDArray::New(Z,n);//.............create & polulate "Z" array
 for(*j=0;*j<*n;++*j)for(j[1]=0;j[1]<n[1];++j[1])for(j[2]=0;j[2]<n[2];++j[2])
  Z[*j][j[1]][j[2]]=exp(-pow(X[0][*j],2)-pow(X[1][j[1]],2)-pow(X[2][j[2]],2));
 double x[]={.32,.46,-.11};//..............................interpolation point
 for(int i=0;i<3;++i)J[i]=yInterp::BinarySearch(X[i],X[i]+n[i]-1,x[i])-X[i];
 printf("z(%f,%f,%f)=%f\n",x[0],x[1],x[2],exp(-x[0]*x[0]-x[1]*x[1]-x[2]*x[2]));
 printf("f(%f,%f,%f)=%f\n",x[0],x[1],x[2],yKDArray::LinInterp(Z,J,x,X));
 yKDArray::Delete(Z),yKDArray::Delete(X);}//............................cleanup
```

7.2.2 Example Output

```
z(0.320000,0.460000,-0.110000)=0.721733
f(0.320000,0.460000,-0.110000)=0.719214
```

8. Example: Interpolating a Rectilinear-Grid-Based Array

The following example code begins by creating a 2-D rectilinear grid of *Z* values that are calculated using Eq. 35. Next, yKDArray::LinInterp() is used to calculate values for the output image shown in Fig. 3. Black dots represent locations of stored *Z* values. Colored pixels represent interpolated values (lighter colors imply larger numbers).

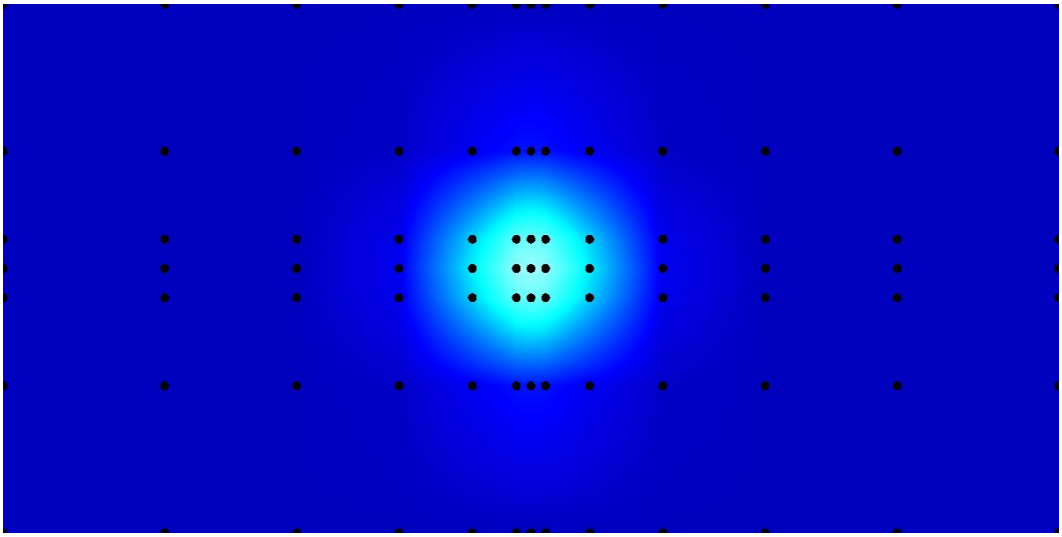

Fig. 3 "rectilinear.bmp"

Example Code

```
#include"y_bmp.h"//.....................................yBmp (see ARL-TN-559)
#include"y_interp.h"//.................................yInterp (see ARL-TN-0657)
#include"y_kd_array.h"//...............................yKDArray (see ARL-TN-1200)
int main(){//<==============================EXAMPLE: CREATE AN IMAGE OF A SURFACE
 int n[]={13,7},r=100,j[2],J[2];//.dimension sizes, pixel scaling, array indices
 double**X;/*<-*/yKDArray::New2D(X,2,n);
 for(int i=0;i<2;++i)for(j[i]=0;j[i]<n[i];++j[i])
  X[i][j[i]]=2./(n[i]-1)*pow(j[i]-(n[i]-1)/2.,2.)*(j[i]<((n[i]-1)/2.)?-1:1);
 double**Z;/*<-*/yKDArray::New(Z,n);
 for(*j=0;*j<*n;++*j)for(j[1]=0;j[1]<n[1];++j[1]){
  Z[*j][j[1]]=1;
  for(int i=0;i<2;++i)Z[*j][j[1]]*=exp(-pow(X[i][j[i]],2));}
 unsigned char*I=yBmp::New((*n-1)*r,(n[1]-1)*r),*P;
 for(int p=0,q,i;p<(*n-1)*r;++p)for(q=0;q<(n[1]-1)*r;++q){//....loop thru pixels
  double x[2]={p*(X[0][*n-1]-**X)/((*n-1)*r-1)+**X,
   q*(X[1][n[1]-1]-*X[1])/((n[1]-1)*r-1)+*X[1]};
  for(i=0;i<2;++i)J[i]=yInterp::BinarySearch(X[i]+1,X[i]+n[i]-1,x[i])-X[i];
  double f=yKDArray::LinInterp(Z,J,x,X),c=3*(f+.4)/1.6;//.c is for color scaling
  for(i=0;i<2;++i)if(x[i]>(X[i][J[i]]+X[i][J[i]+1])/2)++J[i];
  if(hypot(*x-X[0][*J],x[1]-X[1][J[1]])>.05)//.....................black dots
   for(i=0,P=yBmp::Pixel(I,p,q);i<3;++i)P[i]=(c<i?0:c<i+1?c-i:1)*255;}
 FILE*F=fopen("rectilinear.bmp","wb");/*<-*/yBmp::Write(F,I);
 yKDArray::Delete(X),yKDArray::Delete(Z),delete[]I;}//.................cleanup
```

9. Example: Interpolating a Curvilinear-Grid-Based Array

The following example code begins by creating a 2-D polar grid of *Z* values that are calculated using Eq. 35. Next, yKDArray::LinInterp() is used to calculate values for the output image shown in Fig. 4. Black dots represent locations of stored *Z* values. Colored pixels represent interpolated values (lighter colors imply larger numbers).

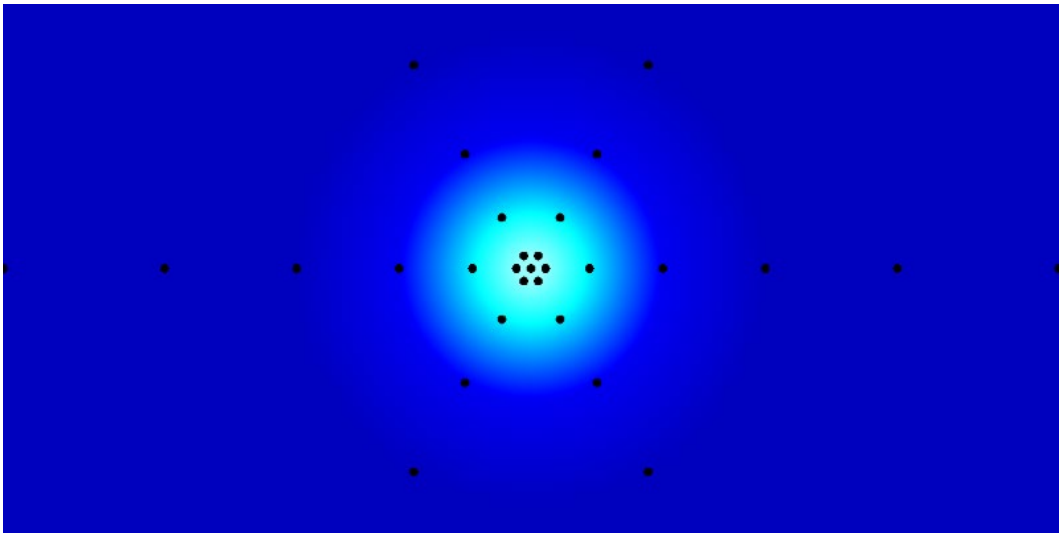

Fig. 4 "curvilinear.bmp"

Example Code

```
#include"y_bmp.h"//.....................................yBmp (see ARL-TN-559)
#include"y_interp.h"//.................................yInterp (see ARL-TN-0657)
#include"y_kd_array.h"//.............................yKDArray (see ARL-TN-1200)
int main(){//<============================EXAMPLE: CREATE AN IMAGE OF A SURFACE
 int n[]={7,7},r=100,j[2],J[2];//..dimension sizes, pixel scaling, array indices
 double pi=3.14159265358979;
 double**X;/*<-*/yKDArray::New2D(X,2,n);
 for(*j=0;*j<*n;++*j)X[0][*j]=*j**j/(*n-1.);
 for(j[1]=0;j[1]<n[1];++j[1])X[1][j[1]]=j[1]*2*pi/(n[1]-1)-pi;
 double**Z;/*<-*/yKDArray::New(Z,n);
 for(*j=0;*j<*n;++*j)for(j[1]=0;j[1]<n[1];++j[1])
  Z[*j][j[1]]=exp(-X[0][*j]*X[0][*j]);
 unsigned char*I=yBmp::New(2*(*n-1)*r,(n[1]-1)*r),*P;
 for(int p=0,q,i;p<2*(*n-1)*r;++p)for(q=0;q<(n[1]-1)*r;++q){//..loop thru pixels
  double x[2]={p*2*(X[0][*n-1])/(2*(*n-1)*r-1)-X[0][*n-1],
   q*(X[0][*n-1])/((*n-1)*r-1)-X[0][*n-1]/2};
  double xp[2]={sqrt(*x**x+x[1]*x[1]),atan2(x[1],*x)};
  for(i=0;i<2;++i)J[i]=yInterp::BinarySearch(X[i]+1,X[i]+n[i]-1,xp[i])-X[i];
  double f=yKDArray::LinInterp(Z,J,xp,X),c=3*(f+.4)/1.6;//c is for color scaling
  for(i=0;i<2;++i)if(xp[i]>(X[i][J[i]]+X[i][J[i]+1])/2)++J[i];
  if(hypot(*x-X[0][*J]*cos(X[1][J[1]]),x[1]-X[0][*J]*sin(X[1][J[1]]))>.05)
   for(i=0,P=yBmp::Pixel(I,p,q);i<3;++i)P[i]=(c<i?0:c<i+1?c-i:1)*255;}
 FILE*F=fopen("curvilinear.bmp","wb");/*<-*/yBmp::Write(F,I);
 yKDArray::Delete(X),yKDArray::Delete(Z),delete[]I;}//..................cleanup
```

10. Example: Storing Arrays in Text Files

Arrays that have been created using either yKDArray::New() or yKDArray::New2D() can easily be stored in (and retrieved from) text files.

The following example code uses yKDArray::New() and yKDArray::New2D() to create the same X and Z arrays that were created in Section 7.2. However, in this example the arrays are written to a file. For file-writing purposes, the arrays are treated as 1-D.

10.1 Example Code

```
#include<cmath>//.....................................................exp(),pow()
#include<cstdio>//....................................FILE,fopen(),fprintf()
#include"y_kd_array.h"//.............................yKDArray (see ARL-TN-1200)
int main(){//<==============================EXAMPLE: STORE ARRAYS IN A TEXT FILE
 FILE*F=fopen("3D_array.txt","w");
 int n[]={11,11,11};//.................allocate, populate, & write array indices
 for(int i=0;i<3;++i)fprintf(F,"%d ",n[i]);/*&*/fprintf(F,"\n");
 double**X;/*<-*/yKDArray::New2D(X,3,n);//.allocate, populate, & write "X" array
 for(int i=0,j;i<3;++i)for(j=0;j<n[i];++j)X[i][j]=-.5+.1*j;
 for(int i=0,N=*n+n[1]+n[2];i<N;++i)fprintf(F,"%.1f ",(*X)[i]);fprintf(F,"\n");
 double***Z;/*<-*/yKDArray::New(Z,n);//....allocate, populate, & write "Z" array
 for(int j0=0,j1,j2;j0<*n;++j0)for(j1=0;j1<n[1];++j1)for(j2=0;j2<n[2];++j2)
  Z[j0][j1][j2]=exp(-pow(X[0][j0],2)-pow(X[1][j1],2)-pow(X[2][j2],2));
for(int i=0,N=*n*n[1]*n[2];i<N;++i)fprintf(F,"%E ",(**Z)[i]);
 yKDArray::Delete(Z),yKDArray::Delete(X);}//.............................cleanup
```

10.2 Example Output (3D_array.txt)

```
11 11 11
-0.5 -0.4 -0.3 -0.2 -0.1 0.0 0.1 0.2 0.3 0.4 0.5 -0.5 -0.4 -0.3 -0.2 -0.1 0.0 0.
4.723666E-001 5.168513E-001 5.543273E-001 5.827483E-001 6.004956E-001 6.065307E-
```

Note that in the above output there are more X and Z values than are shown; only the first 80 characters of each line are present.

11. Example: Reading Arrays from Text Files

The following example code reads and parses the file that was created by the example code presented in Section 10. The parsed text is then used to populate n, X, and Z arrays. Finally, the arrays are used to recreate the interpolation that was presented in Section 7.2.

11.1 Example Code

```
#include<cstdio>//.............................FILE,fopen(),fscanf(),printf()
#include"y_interp.h"//.................................yInterp (see ARL-TN-0657)
#include"y_kd_array.h"//..............................yKDArray (see ARL-TN-1200)
int main(){//<=========================EXAMPLE: RETRIEVE ARRAYS FROM A TEXT FILE
 FILE*F=fopen("3D_array.txt","r");
 int n[3];/*<-*/fscanf(F,"%d%d%d",n,n+1,n+2); //..............read array indices
 double**X;/*<-*/yKDArray::New2D(X,3,n);//.......................read "X" array
 for(int i=0;i<n[0]+n[1]+n[2];++i)fscanf(F,"%lf",*X+i);
 double***Z;/*<-*/yKDArray::New(Z,n);//..........................read "Z" array
 for(int i=0;i<n[0]*n[1]*n[2];++i)fscanf(F,"%lf",**Z+i);
 double x[]={.32,.46,-.11};//................................interpolate at "x"
 int J[3];
 for(int i=0;i<3;++i)J[i]=yInterp::BinarySearch(X[i],X[i]+n[i]-1,x[i])-X[i];
 printf("f(%f,%f,%f)=%f\n",x[0],x[1],x[2],yKDArray::LinInterp(Z,J,x,X));
 yKDArray::Delete(X),yKDArray::Delete(Z);}//............................cleanup
```

11.2 Example Output

```
f(0.320000,0.460000,-0.110000)=0.719214
```

12. Conclusions

The code presented here provides a versatile set of tools for working with KD arrays. Examples demonstrate that KD arrays can be easily created, manipulated, written to text files, and retrieved from those same text files. Furthermore, linear interpolations can be performed on data that needn't conform to evenly spaced (or even rectilinear) grids.

13. References

1. Yager RJ. Basic searching, interpolating, and curve-fitting algorithms in C++. Army Research Laboratory (US); 2015 Jan. Report No.: ARL-TN-0657.

2. Yager RJ. Generating pseudorandom numbers from various distributions using C++. Army Research Laboratory (US); 2014 June. Report No.: ARL-TN-613.

3. Yager RJ. Reading, writing, and modifying BMP files using C++. Army Research Laboratory (US); 2013 Aug. Report No.: ARL-TN-559.

Appendix. Supporting Code

The yInterp Namespace (y_interp.h)

```
#ifndef YINTERP_GUARD//         See Yager, R.J. "Basic Searching, Interpolating,
#define YINTERP_GUARD//                      and Curve-Fitting Algorithms in C++"
#include<cmath>//.....................................................fmod(),pow()
namespace yInterp{//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 template<class T>T*BinarySearch(//<=========================FIND p|*p<=k<*(p+1)
   T*a,T*b,//<-----ARRAY START & END POINTS (ARRAY MUST BE SORTED IN INC. ORDER)
   T k){//<-----KEY (FOR ARRAYS WITH NON-UNIQUE ENTRIES *p=k=*(p+1) IS POSSIBLE)
  if(k<*a)return a-1;//............note that a-1 may point to an invalid address
  for(T*c;k<*--b;k>*c?a=c:b=c+1)c=a+(b-a)/2;/*->*/return b;}
template<class T>T*PeriodicSearch(//<========FOR ARRAYS WITH PERIODIC VARIABLES
   T*a,T*b,//<-----STARTING & ENDING POINTS (ARRAY MUST BE SORTED IN INC. ORDER)
   T&k,//<-----------------------------------------KEY (WILL BE SET TO k'=k+np)
   T p){//<-------------------------------------------------------------PERIOD (p>0)
  return BinarySearch(a,b,k=fmod(k-*a,p)+*a+(k-*a<0?p:0));}
 template<class T>T NNInterp(//<====================NEAREST-NEIGHBOR INTERPOLATOR
   const T*x,const T*Y//<========BRACKETING VALS (*X,X[1],*Y,Y[1] MUST BE VALID)
   T x){//<-----------------VALUE TO INTERPOLATE AT (TYPICALLY, *X <= x <= X[1])
  return x>(*X+X[1])/2?Y[1]:*Y;}
 template<class T>T LinInterp(//<============================LINEAR INTERPOLATOR
   const T*x,const T*Y//<========BRACKETING VALS (*X,X[1],*Y,Y[1] MUST BE VALID)
   T x){//<-----------------VALUE TO INTERPOLATE AT (TYPICALLY, *X <= x <= X[1])
  return*Y+(Y[1]-*Y)*(x-*X)/(X[1]-*X);}
 template<class T>T CubeInterp(//<===========CUBIC (HERMITE SPLINE) INTERPOLATOR
   const T*x,const T*Y//<========BRACKETING VALS (*X,X[1],*Y,Y[1] MUST BE VALID)
   T x,//<------------------VALUE TO INTERPOLATE AT (TYPICALLY, *X <= x <= X[1])
   T m0,T m1){//<-----------------------------------------SLOPES AT *X AND X[1]
  T t=(x-*X)/(X[1]-*X);
  return*Y+(x-*X)*(((m0+m1)*t-(2*m0+m1))*t+m0)+(Y[1]-*Y)*(3-2*t)*t*t;}
 template<class T>T CardinalSlope(//<=========CALCULATES SLOPES FOR CubeInterp()
   const T*x,const T*Y//<=BRACKETING VALS (X[-1],X[1],Y[-1],Y[1] MUST BE VALID)
   T t){//<-----------------------------------------------------TENSION PARAMETER
  return(1-t)*(Y[1]-Y[-1])/(X[1]-X[-1]);}
 template<class T>void PolyFit(//<==========FITS A POLYNOMIAL TO A SET OF POINTS
   const T*X,//<----------INDEPENDENT-VARIABLE VALUES (EACH X[i] MUST BE UNIQUE)
   const T*Y,//<----------------------DEPENDENT-VARIABLE VALUES (SAME SIZE AS X)
   int n,//<-----------------------------------------NUMBER OF ELEMENTS IN X OR Y
   int m,//<-------------------NUMBER OF COEFFICIENTS IN THE BEST-FIT POLYNOMIAL
   T*C){//<---------STORAGE FOR COEFFICIENTS (SIZE=m) y=C[0]+C[1]*x+C[2]*x^2+...
  T**A=new T*[m];/*<-*/for(int i=0,j,k;i<m;++i){
   for(A[i]=new T[m+1],j=0;j<m+1;++j)A[i][j]=0;
   for(k=0;k<n;++k)for(A[i][m]+=pow(X[k],i)*Y[k],j=0;j<m;++j)
    A[i][j]+=pow(X[k],i+j);}//.................................augmented matrix
  for(int i=1,j,k;i<m;++i)for(k=0;k<i;++k)for(j=m;j>=0;--j)
    A[i][j]-=A[i][k]*A[k][j]/A[k][k];//.....................Gaussian elimination
  for(int i=m-1,j;i>=0;--i)for(C[i]=A[i][m]/A[i][i],j=i+1;j<m;++j)
   C[i]-=C[j]*A[i][j]/A[i][i];//..........................backward substitution
  for(int i=0;i<m;++i)delete[]A[i];/*&*/delete[]A;}
}//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#endif//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ YAGENAUT@GMAIL.COM
```

The yRandom Namespace (y_random.h)

```
#ifndef YRANDOM_GUARD//         See Yager, R.J. "Generating Pseudorandom Numbers
#define YRANDOM_GUARD//                    from Various Distributions Using C++"
#include<cmath>//.....................................cos(),exp(),log(),sqrt()
namespace yRandom{//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 void Seed(unsigned*I,unsigned s){//<===INITIALIZE 625-ELEMENT STATE WITH SEED s
  for(I[1]=s,*I=s=1;s<626;++s)I[s+1]=1812433253*(I[s]^I[s]>>30)+s;}
 unsigned MT(unsigned*I){//<=======================MERSENNE TWISTER (19937) PRNG
  unsigned i=*I,j=i<624?i+1:1,y=I[i]&0x80000000|I[j]&0x7fffffff;
  y=I[i]=I[i<228?i+397:i-227]^y>>1^(y&1)*0x9908b0df,*I=j;
  return y^=y>>11,y^=y<<7&0x9d2c5680,y^=y<<15&0xefc60000,y^y>>18;}
 double Uniform(unsigned*I,double a,double b){//<========UNIFORM DOUBLE IN [a,b)
  return a+MT(I)*(b-a)/4294967296;}
 double Normal(unsigned*I,double m,double s){ //<========MEAN (m), STD. DEV. (s)
  return m+s*sqrt(2*log(4294967296/(MT(I)+1.)))*cos(1.46291807926716E-9*MT(I));}
 int Integer(unsigned*I,int a,int b){ //<====================UNIFORM INT IN [a,b]
  return a+int(MT(I)/4294967296.*(b-a+1));}
 unsigned Poisson(unsigned*I,double m){ //<====================POISSON INT (m>0)
  int k=-1;/*->*/for(m=exp(-m);m<1;m/=MT(I)/4294967296.)++k;/*->*/return k;}
}//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#endif//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ YAGENAUT@GMAIL.COM
```

The yBmp Namespace (y_bmp.h)

```
#ifndef YBMP_GUARD//                        See Yager, R.J. "Reading, Writing, and
#define YBMP_GUARD//                               Modifying BMP Files Using C++"
#include<cstdio>//.................................FILE,fread(),fwrite(),rewind()
#include<cstring>//.............................................memcpy(),memset()
namespace yBmp{//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 unsigned char*New(int w,int h,//<====CREATE A NEW BMP (WIDTH, HEIGHT IN PIXELS)
   unsigned char f=0){//------------------------------FILL (0=>BLACK,255=>WHITE)
  int N=h*(w*3+w%4)+26;
  unsigned char*B=new unsigned char[N],H[26]={66,77,N,N>>8,N>>16,N>>24,0,0,0,0,
   26,0,0,0,12,0,0,0,w,w>>8,h,h>>8,1,0,24};
  memset(B,f,N),memcpy(B,H,26);/*->*/return B;}
 int Size(const unsigned char*B){//<==================SIZE OF PIXEL DATA (BYTES)
  return (B[2]|B[3]<<8|B[4]<<16|B[5]<<24)-B[10];}
 int Width(const unsigned char*B){return B[18]|B[19]<<8;}//<======WIDTH (PIXELS)
 int Height(const unsigned char*B){//<============================HEIGHT (PIXELS)
  return Size(B)/(Width(B)*3+B[18]%4);}
 unsigned char*Pixel(unsigned char*B,//<==============RETURN A POINTER TO A PIXEL
   int i,int j){//<-----------------------------------------PIXELS RIGHT, UP
  return B+B[10]+(Width(B)*3+B[18]%4)*j+3*i;}
 void Write(FILE*F,const unsigned char*B){//<==============WRITE A BMP TO A FILE
  fwrite(B,1,B[2]|B[3]<<8|B[4]<<16|B[5]<<24,F);}
 unsigned char*Read(FILE*F){//<============================READ A BMP FROM A FILE
  unsigned char H[10];/*<-*/fread(H,1,10,F),rewind(F);
  unsigned char*B=new unsigned char[Size(H)+H[10]];
  fread(B,1,Size(H)+H[10,F);/*->*/return B;}
 void Copy(unsigned char*D,unsigned char*S,//<=======COPY PIXEL DATA FROM S TO D
   int W,int H){//<---------------------------------------WIDTH, HEIGHT OFFSETS
  int wd=Width(D),w=Width(S)+W,hd=Height(D),h=Height(S)+H,i=W>0?W:0,I=wd<w?wd:w;
  if(i<I)for(int j=H>0?H:0,J=hd<h?hd:h;j<J;++j)
   memcpy(Pixel(D,i,j),Pixel(S,i-W,j-H),3*(I-i));}
}//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#endif//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ YAGENAUT@GMAIL.COM
```

List of Symbols, Abbreviations, and Acronyms

1-D	one-dimensional
2-D	two-dimensional
3-D	three-dimensional
K-D	*K*-dimensional

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

1 (PDF) DEVCOM ARL FCDD RLB CI TECH LIB

29 (PDF) DEVCOM ARL FCDD RLA TD
A BARD
B CHESTER
N BRUCHEY
M DEVINE
R DONEY
E ELDRIDGE
J FLENIKEN
R GUPTA
S HALSEY
M KEELE
D KLEPONIS
B KRZEWINSKI
B LINNE
K MASSER
F MURPHY
J PERRELLA
D PETTY
C RANDOW
S SCHRAML
J SENGUPTA
R SPINK
J STEWART
K STOFFEL
R STRICKLAND
N TEE
D WEBB
R YAGER
M ZELLNER
FCDD RLA TE
G VUNNI