JS LIBRARY: GENERAL 1 v. 1.1

by: John A. Schlack

created: September 11, 1990

last modified: November 21, 1990

Introduction

This is a first in a series of libraries of procedures and functions that I will be releasing. This was created using THINK C version 4.0. The libraries should be useful on other platforms (especially THINK Pascal 3.0) since the library consists of object code which is simply linked into the program being developed. Please consult the manual that came with your development environment for information about linking code from other programs. Subjects include complex numbers, dialog utilities, geometry, linear algebra, memory allocation and deallocation, searching, sorting, and string utilities.

Below, I will describe each function, give the prototype, and any other relevant information. With this package, I have also included a header file, the library, and the resources. I have tried to make these libraries bug free. If you find a bug (and you have registered), please feel free to report it to me. Include the software name and version number, procedure, problem, computer that you use, etc. I will send you and all registered users the bug fix.

There are two versions of each library. The normal version: "`JS_General 1.lib`", and "`JS_General 1.881.lib`", a version compiled with the 68020 and 68881 compiler options chosen. If you are using a Mac II series computer with a math coprocessor, use the "`JS_General 1.881.lib`" library.

Shareware Registration

These libraries do not have restricted functions. However, they do not include all of the functions that I developed on the subjects. A complete library can be obtained by paying the Shareware fee. The descriptions below contain all of the functions. Those that are not included and are only available by paying the Shareware fee are noted at the end of this file.

The registration fee is $10 (try to send only checks or money orders). If you pay this fee, you will receive the complete library along with the header file and descriptions. You will receive future upgrades and bug fixes for free.

If you plan on using this library in creation of any program that is not intended and used exclusively for personal use (such as Public Domain, Shareware, Freeware, or a commercial product), you must pay the Shareware fee (see the following paragraph). If you create and use a program for personal use and do NOT distribute the program in any fashion, you do not need to pay the Shareware fee (but you will not receive any support, upgrade, or missing functions), and may skip the next paragraph.

By paying the one time Shareware fee, you have my permission for unlimited use of these functions and procedures in your programs. You may NOT use these functions and procedures in your programs if you fail to pay the fee. Feel free to distribute the non-registered library along with all *documentation that came with the package (especially the registration notice). Do NOT distribute the registered version.*

Please send the Shareware fee to:

John A. Schlack

824 Rhoads Avenue

Jenkintown, Pa. 19046

Other Notes / Getting Started

In order to use these libraries (for THINK C 4.0), the following must be done:

1. make sure that the Toolbox is initialized
2. include `<stdlib.h>` before including "`JS_General 1.h`". If you do not, you will not be able to access the random number macros (JS_RANDOM and JS_RANDOM_2).
3. include the file "`JS_General 1.h`" in your programs
4. copy the included resources from "`JS_General 1.π.rsrc`" into your resource file
5. include the library "`JS_General 1.lib`" or "`JS_General 1.881.lib`" into a new segment in your project file

Array and matrix pointers must be declared and allocated before calling any routines (except for the memory allocation routines). If this is not done, the program being written will certainly crash (the only question is when). Please note the array or matrix format required for each procedure. Some always start at 1, others at 0, still others can vary.

If you are reading the text only file, you will need to look at the PICT file "`Geometry.PICT`" for the pictures that are related to the geometry functions.

Constants

JS_PI = 3.14159265358979
JS_TwoPI = 6.28318530717959
JS_TRUE = 1
JS_FALSE = 0
JS_SUPPRESS = 0
JS_NO_SUPPRESS = 1

Suppress some outputs.

JS_STD_HIGHLIGHT = 1

This is the standard highlight button (the resource in a DLOG with an ID of 1).

JS_CONE_HEIGHT = 0
JS_CONE_LENGTH = 1

These are flags that tell the computer whether the length of a side of the cone or the height of the cone was entered.

JS_FRUSTUM_HEIGHT = 0
JS_FRUSTUM_LENGTH = 1

These are flags that tell the computer whether the length of a side of the frustum of a cone or the height of the frustum of a cone was entered.

JS_BEGIN = -1
JS_ALL = 0
JS_END = 1

These are flags that tell the computer to strip characters from the beginning, entire, or end of the string, respectively.

Macros

Macros generally execute faster than procedures, since there is an overhead on procedure calls. The drawback to the macros is that the more they are used, the larger the object code of your program will be.

SUPPORT

```
#define         JS_MIN(a,b)         (((a) < (b)) ? (a) : (b))
```

This macro returns the minimum of two variables a and b. This procedure has an added advantage of working with what ever type of variable that you are (and does not need to use time to convert to type double as with JS_Min()).

```
#define         JS_MAX(a,b)         (((a) > (b)) ? (a) : (b))
```

This macro returns the maximum of two variables a and b. This procedure has an added advantage of working with what ever type of variable that you are (and does not need to use time to convert to type double as with JS_Max()).

```
#define         JS_SQUARE(x)        ((x) * (x))
```

This macro squares a value x. The drawback to this macro is that if x is an expression, it must be evaluated twice (once for each (x) above). Thus, if a long expression is passed, it might be slower than calling the JS_Square() function). One advantage is that the macro works with the current type and does not need to be converted to type double.

```
#define         JS_CUBE(x)          ((x) * (x) * (x))
```

This macro cubes a value x. The drawback to this macro is that if x is an expression, it must be evaluated three times (once for each (x) above). Thus, if a long expression is passed, it might be slower than calling the JS_Cube() function). One advantage is that the macro works with the current type and does not need to be converted to type double.

```
#define         JS_RANDOM(x)        (rand() % ((x) + 1))
```

This macro returns a random number from 0 up to and including x. x must be an integer or long.

```
#define         JS_RANDOM_2(a,b)    (rand() % ((b) - (a) + 1) + (a))
```

This macro returns a random number from a up to and including b. a and b must be an integers or longs.

Structures and Types

COMPLEX NUMBERS

```
typedef struct JS_Complex
{
	double re;
	double im;
} JS_Complex;
```

This is the structure for a complex number. It is double precision with both real and imaginary components.

Functions and Procedures

COMPLEX NUMBERS

```
JS_Complex    JS_Complex_Add( JS_Complex c1, JS_Complex c2);
```

This function adds two complex numbers. It returns a complex number.

```
JS_Complex    JS_Complex_Sub( JS_Complex c1, JS_Complex c2 );
```

This function subtracts the second complex number (c2) from the first (c1). It returns a complex value.

```
JS_Complex    JS_Complex_Mult( JS_Complex c1, JS_Complex c2 );
```

This function returns the result of a multiplication of two complex numbers.

```
JS_Complex    JS_Complex_Div( JS_Complex c1, JS_Complex c2 );
```

This function returns the result of dividing complex number c1 by complex number c2.

```
JS_Complex    JS_Complex_Num( double re, double im );
```

This function uses two double precision numbers re and im to create a complex number. re is the real component and im is the imaginary component of the complex number.

double JS_Complex_Abs(JS_Complex c);

This function returns the magnitude of the complex number c.

JS_Complex JS_Complex_Conjugate(JS_Complex c);

This function returns the complex conjugate of the complex number c.

JS_Complex JS_Complex_Sqrt(JS_Complex c);

This function returns the square root of the complex number c.

JS_Complex JS_Double_Complex_Mult(double x, JS_Complex c);

This function multiplies a real number x with a complex number c and returns a complex number.

JS_Complex JS_Complex_Exp(JS_Complex c);

This function returns the exponential of c (e raised to the c power).

DIALOG UTILITIES

void JS_Fatal_Error_Handler(char *text);

This procedure is used to declare to the user that a fatal error has occurred. It accepts a char * to text. This text is placed in a dialog box and centered on screen. When the OK button is clicked or the user strikes return, the program is TERMINATED.

void JS_Error_Handler(char *text);

This procedure is used to display an error condition to the user. It is the same as JS_Fatal_Error_Handler except that the program will not be terminated, and will proceed normally.

long JS_Get_Dialog_Long(DialogPtr d, int loc, long val);

This function returns the data of type long in the dialog box pointed to by the DialogPtr d and in the editable text box with an ID number of loc. If there is no text in the box, the value val (which the programmer inputs) will be returned.

void JS_Put_Dialog_Long(DialogPtr d, int loc, long val, char sel);

This procedure places the data val of type long into an editable text box with an ID number of loc in a dialog box pointed to by DialogPtr d. If sel is not 0, then the text will be highlighted.

double JS_Get_Dialog_Double(DialogPtr d, int loc, double val);

This function is similar to JS_Get_Dialog_Long except that it returns a value of type double.

void JS_Put_Dialog_Double(DialogPtr d, int loc, double val, char sel);

This function is similar to JS_Put_Dialog_Long except that it puts a value of type double into the editable text box.

int JS_Get_Dialog_Integer(DialogPtr d, int loc, int sel);

This function is similar to JS_Get_Dialog_Long except that it returns a value of type int.

void JS_Put_Dialog_Integer(DialogPtr d, int loc, int val, char sel);

This function is similar to JS_Put_Dialog_Long except that it puts a value of type int into the editable text box.

float JS_Get_Dialog_Float(DialogPtr d, int loc, float val);

This function is similar to JS_Get_Dialog_Long except that it returns a value of type float.

void JS_Put_Dialog_Float(DialogPtr d, int loc, float val, char sel);

This function is similar to JS_Put_Dialog_Long except that it puts a value of type float into the editable text box.

char * JS_Get_Dialog_Text(DialogPtr d, int loc, char *val);

This function is similar to JS_Get_Dialog_Long except that it returns a char * to the data (which will actually be a changed char *val).

void JS_Put_Dialog_Text(DialogPtr d, int loc, char *val, char sel);

This function is similar to JS_Put_Dialog_Long except that it puts the text pointed to by char *val into the editable text box.

void JS_Do_Binary_Control(DialogPtr d, int loc);

This procedure toggles the value of a binary control with an ID of loc(such as a radio button or check box) between on and off. It also takes the DialogPtr d as input.

void JS_Center_Alerts(AlertTHndl a);

This procedure accepts an AlertTHndl a as input and it will change the location of the alert to be the center of the current screen (I have only tried this on single screen Macs). Right after the alert is centered, it should be called by your program since the changes will not be saved. If the alert is purged from memory before it is called, the alert will appear where the resource puts it.

void JS_Current_Device_Size(int *halfH, int *halfV);

This procedure places the horizontal and vertical size of the screen (divided by 2) into *halfH and *halfV respectively. Please note: these values are actually only one half of the screen size.

void JS_Center_Window (WindowPtr w);

This procedure accepts the WindowPtr w and centers the window on the screen (this has only been tested on one screen Macs).

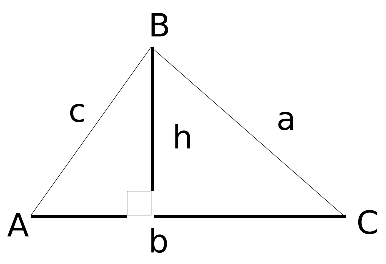
void JS_Highlight_Button (DialogPtr d, int loc);

This procedure highlights the button with ID loc in the dialog pointed to by d. To be safe, use the accompanying constant JS_STD_HIGHLIGHT (which is set to 1).

GEOMETRY

The diagrams below give a visual representation of the data inputs necessary to obtain the correct function values. Please refer to these diagrams to be certain that you are passing the correct parameters to the function.

Triangle



double JS_Triangle_Area_BH(double b, double h);

This function returns the area of a triangle given the base length b and the height h.

double JS_Triangle_Area_BCA(double b, double c, double A);

This function returns the area of a triangle if given two sides (b and c) and the angle between those sides (angle at A).

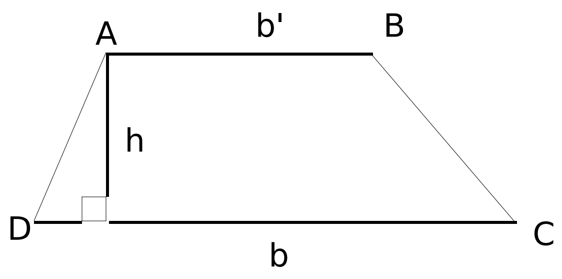
double JS_Deg_to_Rad(double deg);

This function converts degrees deg to radians and returns the radian value of an angle.

double JS_Rad_to_Deg(double rad);

This function returns the value in degrees of an angle rad provided in radians.

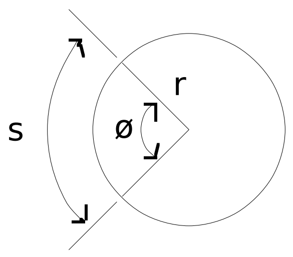
Trapezoids



double JS_Trapazoid_Area(double b, double bPrime, double h);

This function returns the area of a trapezoid if given the two parallel side lengths, b and b', and the height h.

Circle



double JS_Circle_Area(double r);

This function returns the area of a triangle given the radius r.

double JS_Circle_Circumference(double r);

This function returns the circumference of a circle given the radius r.

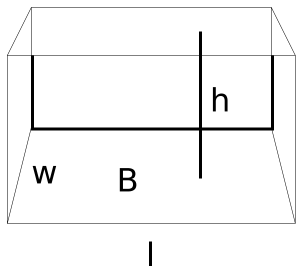
double JS_Circle_Sector_Length(double r, double phi);

This function returns the length of the circular arc s if the radius r and the angle ø (phi) is provided.

double JS_Circle_Sector_Area(double r, double phi);

This function returns the area of the sector s if given the radius r and the angle ø (phi).

Prism



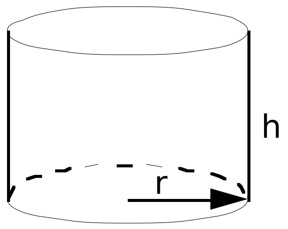
double JS_Prism_Volume_Bh(double B, double h);

This function returns the area of a prism (box) if given the base area B and the height h.

double JS_Prism_Volume_ABC(double l, double w, double h);

This function returns the area of a prism if provided the length l, width w, and height h.

Cylinder



double JS_Cylinder_Volume(double r, double h);

This function calculates the volume of a right circular cylinder if given the radius r and the altitude h.

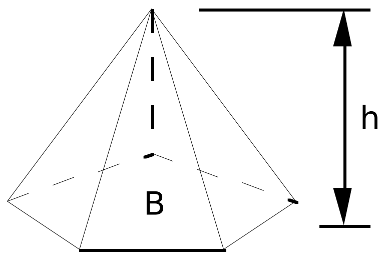
double JS_Cylinder_Lateral_SA(double r, double h);

This function calculates the lateral surface area of a right circular cylinder (not the circles covering the ends) if provided the radius r and height h.

double JS_Cylinder_Total_SA(double r, double h);

This function determines the total surface area of a right circular cyliner (including the ends) if provided the radius r and the height h.

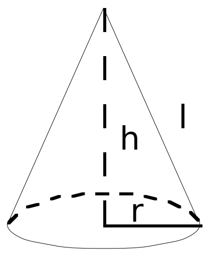
Pyramid



double JS_Pyramid_Volume(double B, double h);

This function finds the volume of a pyramid if given the area of the base B and the height h.

Cone



double JS_Cone_Volume(double r, double alt, short flag);

This function calculates the volume of a right circular cone if the radius r and either the height h or the length of side l is provided (in alt). Set the flag to JS_CONE_HEIGHT (0) if alt = h (height) or JS_CONE_LENGTH (1) if alt = l (length of side). The default is alt = l.

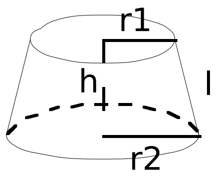
double JS_Cone_Lateral_SA(double r, double alt, short flag);

This function calculates the lateral surface area (area not including circle at bottom) of a right circular cone if the radius r and either the height h or the length of side l is provided (in alt). Set the flag to JS_CONE_HEIGHT (0) if alt = h (height) or JS_CONE_LENGTH (1) if alt = l (length of side). The default is alt = l.

double JS_Cone_Total_SA(double r, double alt, short flag);

This function calculates the total surface area of a right circular cone if the radius r and either the height h or the length of side l is provided (in alt). Set the flag to JS_CONE_HEIGHT (0) if alt = h (height) or JS_CONE_LENGTH (1) if alt = l (length of side). The default is alt = l.

Frustum of a Cone



double JS_Frustum_Cone_Volume(double r1, double r2, double alt, short flag);

This function calculates the volume of a frustum of a right circular cone if the radius r and either the height h or the length of side l is provided (in alt). Set the flag to JS_FRUSTUM_HEIGHT (0) if alt = h (height) or JS_FRUSTUM_LENGTH (1) if alt = l (length of side). The default is alt = l. The order of entering the radii does not matter.

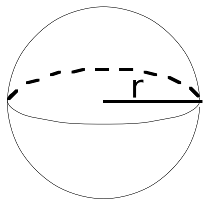
double JS_Frustum_Cone_LSA(double r1, double r2, double alt, short flag);

This function calculates the lateral surface area (not including two sircles) of a frustum of a right circular cone if the radius r and either the height h or the length of side l is provided (in alt). Set the flag to JS_FRUSTUM_HEIGHT (0) if alt = h (height) or JS_FRUSTUM_LENGTH (1) if alt = l (length of side). The default is alt = l. The order of entering the radii does not matter.

double JS_Frustum_Cone_TSA(double r1, double r2, double alt, short flag);

This function calculates the total surface area of a frustum of a right circular cone if the radius r and either the height h or the length of side l is provided (in alt). Set the flag to JS_FRUSTUM_HEIGHT (0) if alt = h (height) or JS_FRUSTUM_LENGTH (1) if alt = l (length of side). The default is alt = l. The order of entering the radii does not matter.

Sphere



double JS_Sphere_Volume(double r);

This function returns the volume of a sphere if given the radius r.

double JS_Sphere_SA(double r);

This function returns the surface area of a sphere if given the radius r.

LINEAR ALGEBRA

void JS_Add_Vectors_No_Destroy(double *v1, double *v2, double *v3, short len, short start, short flag);

This procedure adds two vectors, v1 and v2, and stores the result in the third, v3. v1 and v2 are left intact. The length of the vectors is len, and their starting subscript is start. v1[start .. (len+start-1)]. If the flag is negative, then v2 is subtracted from v1 instead of added.

If the length len is less than 2, JS_Fatal_Error_Handler is called and the program is terminated.

void JS_Add_Vectors(double *v1, double *v2, short len, short start, short flag);

This procedure adds two vectors, v1 and v2, and stores the result in v1 (destroying the previous contents). v2 is left intact. The length of the vectors is len, and their starting subscript is start. v1[start .. (len+start-1)]. If the flag is negative, then v2 is subtracted from v1 and the result is placed in v1.

If the length len is less than 2, JS_Fatal_Error_Handler is called and the program is terminated.

double JS_Vector_Magnitude(double *v, short len, short start);

This function obtains the magnitude of the vector v, whose length is len and initial subscript is start. v[start .. (len+start-1)]. If the length len is less than 2, JS_Fatal_Error_Handler is called and the program is terminated.

void JS_Normalize_Vector(double *v, short len, short start);

This procedure normalizes the vector v. This means that each component of the vector is divided by the magnitude of the vector. Thus, the normal vector will have a length of 1. The length of the vector is len, and the initial subscript is start. v[start .. (len+start-1)]. If the length len is less than 2, JS_Fatal_Error_Handler is called and the program is terminated.

short JS_Vector_Compare(double *v1, double *v2, short len, short start,
double thresh);

This function returns JS_TRUE if the two vectors v1 and v2 are equal to within some threshhold thresh per component or JS_FALSE if they are not. len is the length of the vector and start is the initial subscript. v1[start .. (len+start-1)]. thresh is a programmer input threshhold per component. If two vector components are within thresh of one another, then they are considered equal. Thus, is all components in v1 are within thesh of their corresponding components in v2, then the function will return JS_TRUE.

If the length len is less than 2, JS_Fatal_Error_Handler is called and the program is terminated.

void JS_Add_Matrices(double **m1, double **m2, short si, short sj, short li,
short lj, short flag);

This procedure adds two matrices m1 and m2 of length li by lj whose starting subscripts are si and sj. m1[si .. (si+li-1)][sj .. (sj+lj-1)]. The result of the addition is stored in m1. m2 remains unchanged. If the flag is negaitve, the second matrix is subtracted from the first.

If either length (li or lj) is less than 2, JS_Fatal_Error_Handler is called and the program is terminated.

void JS_Add_Matrices_No_Destroy(double **m1, double **m2, double **m3,
short si, short sj, short li, short lj, short flag);

This procedure adds two matrices m1 and m2 of length li by lj whose starting subscripts are si and sj. m1[si .. (si+li-1)][sj .. (sj+lj-1)]. The result of the addition is stored in m3 (which must be at least as large as m1 and m2). m1 and m2 remain unchanged. If the flag is negaitve, the second matrix is subtracted from the first.

If either length (li or lj) is less than 2, JS_Fatal_Error_Handler is called and the program is terminated.

```
void    JS_Multiply_Matrices( double **m1, double **m2, double **m3, short m,
        short n, short p, short flag );
```

This procedure multiplies two matrices m1 and m2 and stores the result in m3. This procedure is more strict than the previous. The subscript of all matrices must either start at 0 or 1. Set flag to either 0 or 1 according to the starting subscript of the matrix. The default is 0. matrices m1 and m2 are not altered.

The size of the matrices being multiplies is also important. The sizes must be: m1[m][p], m2[p][n], and m3[m][n]. If m, n, or p is less than 1, the procedure will call JS_Fatal_Error_Handler and the program will terminate.

```
char    JS_Handle_Eigenvalue( double **m, int i, double *e, char suppress );
```

This routine calculates the eigenvalues and eigenvectors for a real, symmetric matrix m[1..i][1..i], where i > 1. e is a vector [1..i] which are initially empty. On return, it will contain the eigenvalues. m will contain the eigenvectors for those eigenvalues. The vectors will be stored in the columns, where the kth eigenvalue will have eigenvectors in the kth column of matrix m.

If the length i is less than 2, JS_Fatal_Error_Handler will be called and the program will terminate. If the matrix is not summetric, JS_Error_Handler will be called, and the program will return JS_FALSE (indicating that the eigenvalues and eigenvectors were not obtained. Otherwise, JS_TRUE will be returned.

suppress is a variable used to prevent JS_Error_Handler from being called. Pass in the value JS_SUPPRESS to prevent the display of the error message if the matrix is not symmetric, otherwise, enter JS_NO_SUPPRESS for suppress. The default is no suppression.

```
double JS_Matrix_Determinant( double **m, int n );
```

This function takes an N x N matrix m[1..n][1..n] and obtains its determinant (which is the value returned). JS_Fatal_Error_Handler is called and the program terminated if n < 2.

```
void    JS_Matrix_Inverse( double **m1, double **m2, int n );
```

This procedure obtains the inverse of matrix m1[1..n][1..n] and places it in matrix m2[1..n][1..n]. m1 will remain intact. The inverse of the inverse will yield the original matrix. S_Fatal_Error_Handler is called and the program terminated if n < 2.

```
void    JS_Solve_Linear_System( double **A, double *B, int n );
```

This procedure solves the linear system of equations A • X = B. The matrix A[1..n][1..n] must be provided along with the vector B[1..n]. The solution vector X[1..n] will be placed in B. A remains intact. JS_Fatal_Error_Handler is called and the program terminated if n < 2.

NOTE: This routine is not accurate if the matrix is singular (which is not checked for).

void JS_Matrix_Transpose(double **m, int n);

This procedure obtains the transpose of the matrix m[1..n] and stores it in m. JS_Fatal_Error_Handler is called and the program terminated if n < 2.

MEMORY

float * JS_Float_Vector(long first, long last, char suppress);

This function is used to allocate an array of floats. The arguments are first, which is the index of the first subscript and last, the last subscript. The function returns a float pointer. The array will be of size: array[first..last]. If first > last, then last is defaulted to equal first (for an array[first..first]).

IF suppress = JS_NO_SUPPRESS and there is not enough memory to allocate the array, JS_Fatal_Error_Handler will be called, and the program will terminate. Otherwise, if suppress = JS_SUPPRESS, the function will return a 0L if there is not enough memory to allocate an array.

char *JS_Char_Vector(long first, long last, char suppress);

This function is used to allocate an array of characters (a string). The arguments are first, which is the index of the first subscript and last, the last subscript. The function returns a char pointer. The array will be of size: array[first..last]. If first > last, then last is defaulted to equal first (for an array[first..first]).

IF suppress = JS_NO_SUPPRESS and there is not enough memory to allocate the array, JS_Fatal_Error_Handler will be called, and the program will terminate. Otherwise, if suppress = JS_SUPPRESS, the function will return a 0L if there is not enough memory to allocate an array.

int * JS_Int_Vector(long first, long last, char suppress);

This function is used to allocate an array of integers. The arguments are first, which is the index of the first subscript and last, the last subscript. The function returns an integer pointer. The array will be of size: array[first..last]. If first > last, then last is defaulted to equal first (for an array[first..first]).

IF suppress = JS_NO_SUPPRESS and there is not enough memory to allocate the array, JS_Fatal_Error_Handler will be called, and the program will terminate. Otherwise, if suppress = JS_SUPPRESS, the function will return a 0L if there is not enough memory to allocate an array.

long * JS_Long_Vector(long first, long last, char suppress);

This function is used to allocate an array of long. The arguments are first, which is the index of the first subscript and last, the last subscript. The function returns a long pointer. The array will be of size: array[first..last]. If first > last, then last is defaulted to equal first (for an array[first..first]).

IF suppress = JS_NO_SUPPRESS and there is not enough memory to allocate the array, JS_Fatal_Error_Handler will be called, and the program will terminate. Otherwise, if suppress = JS_SUPPRESS, the function will return a 0L if there is not enough memory to allocate an array.

double *JS_Double_Vector(long first, long last, char suppress);

This function is used to allocate an array of doubles. The arguments are first, which is the index of the first subscript and last, the last subscript. The function returns a pointer to the double array. The array will be of size: array[first..last]. If first > last, then last is defaulted to equal first (for an array[first..first]).

IF suppress = JS_NO_SUPPRESS and there is not enough memory to allocate the array, JS_Fatal_Error_Handler will be called, and the program will terminate. Otherwise, if suppress = JS_SUPPRESS, the function will return a 0L if there is not enough memory to allocate an array.

void JS_Free_Float_Vector(float *v, long first, long last);

This procedure frees the space occupied by a float array. The parameters are the pointer to the array v, and the subscripts of the first and last elements, first and last. Call this procedure to free space allocated by JS_Float_Vector().

void JS_Free_Int_Vector(int *v, long first, long last);

This procedure frees the space occupied by a integer array. The parameters are the pointer to the array v, and the subscripts of the first and last elements, first and last. Call this procedure to free space allocated by JS_Int_Vector().

void JS_Free_Long_Vector(long *v, long first, long last);

This procedure frees the space occupied by an array of long. The parameters are the pointer to the array v, and the subscripts of the first and last elements, first and last. Call this procedure to free space allocated by JS_Long_Vector().

void JS_Free_Double_Vector(double *v, long first, long last);

This procedure frees the space occupied by an array of doubles. The parameters are the pointer to the array v, and the subscripts of the first and last elements, first and last. Call this procedure to free space allocated by JS_Double_Vector().

void JS_Free_Char_Vector(char *v, long first, long last);

This procedure frees the space occupied by a char array (string). The parameters are the pointer to the array v, and the subscripts of the first and last elements, first and last. Call this procedure to free space allocated by JS_Char_Vector().

float **JS_Float_Matrix(long a1, long a2, long b1, long b2, char suppress);

This function is used to allocate a matrix of float. The matrix will be of size: matrix[a1 .. a2][b1 .. b2]. If a2 < a1, then a2 will be set equal to a1. The same holds true for b1 and b2. A double pointer to the matrix of type float will be returned.

IF suppress = JS_NO_SUPPRESS and there is not enough memory to allocate the matrix, JS_Fatal_Error_Handler will be called, and the program will terminate. Otherwise, if suppress = JS_SUPPRESS, the function will return a 0L if there is not enough memory to allocate an matrix.

double **JS_Double_Matrix(long a1, long a2, long b1, long b2, char suppress);

This function is used to allocate a matrix of double. The matrix will be of size: matrix[a1 .. a2][b1 .. b2]. If a2 < a1, then a2 will be set equal to a1. The same holds true for b1 and b2. A double pointer to the matrix of type double will be returned.

IF suppress = JS_NO_SUPPRESS and there is not enough memory to allocate the matrix, JS_Fatal_Error_Handler will be called, and the program will terminate. Otherwise, if suppress = JS_SUPPRESS, the function will return a 0L if there is not enough memory to allocate an matrix.

int **	JS_Int_Matrix(long a1, long a2, long b1, long b2, char suppress);

This function is used to allocate a matrix of integers. The matrix will be of size: matrix[a1 .. a2][b1 .. b2]. If a2 < a1, then a2 will be set equal to a1. The same holds true for b1 and b2. A double pointer to the matrix of type integer will be returned.

IF suppress = JS_NO_SUPPRESS and there is not enough memory to allocate the matrix, JS_Fatal_Error_Handler will be called, and the program will terminate. Otherwise, if suppress = JS_SUPPRESS, the function will return a 0L if there is not enough memory to allocate an matrix.

long **JS_Long_Matrix(long a1, long a2, long b1, long b2, char suppress);

This function is used to allocate a matrix of long. The matrix will be of size: matrix[a1 .. a2][b1 .. b2]. If a2 < a1, then a2 will be set equal to a1. The same holds true for b1 and b2. A double pointer to the matrix of type long will be returned.

IF suppress = JS_NO_SUPPRESS and there is not enough memory to allocate the matrix, JS_Fatal_Error_Handler will be called, and the program will terminate. Otherwise, if suppress = JS_SUPPRESS, the function will return a 0L if there is not enough memory to allocate an matrix.

char **JS_Char_Matrix(long a1, long a2, long b1, long b2, char suppress);

This function is used to allocate a matrix of char (array of strings). The matrix will be of size: matrix[a1 .. a2][b1 .. b2]. If a2 < a1, then a2 will be set equal to a1. The same holds true for b1 and b2. A double pointer to the matrix of type char will be returned.

IF suppress = JS_NO_SUPPRESS and there is not enough memory to allocate the matrix, JS_Fatal_Error_Handler will be called, and the program will terminate. Otherwise, if suppress = JS_SUPPRESS, the function will return a 0L if there is not enough memory to allocate an matrix.

void JS_Free_Float_Matrix(float **v, long a1, long a2, long b1, long b2);

This procedure frees the space occupied by an matrix of float. The parameters are the pointer to the matrix v, and the subscripts of the first and last elements for the matrix[a1 .. a2][b1 .. b2]. Call this procedure to free space allocated by JS_Float_Matrix().

void JS_Free_Double_Matrix(double **v, long a1, long a2, long b1, long b2);

This procedure frees the space occupied by an matrix of double. The parameters are the pointer to the matrix v, and the subscripts of the first and last elements for the matrix[a1 .. a2][b1 .. b2]. Call this procedure to free space allocated by JS_Double_Matrix().

void JS_Free_Int_Matrix(int **v, long a1, long a2, long b1, long b2);

This procedure frees the space occupied by an matrix of integers. The parameters are the pointer to the matrix v, and the subscripts of the first and last elements for the matrix[a1 .. a2][b1 .. b2]. Call this procedure to free space allocated by JS_Int_Matrix().

void JS_Free_Long_Matrix(long **v, long a1, long a2, long b1, long b2);

This procedure frees the space occupied by an matrix of long. The parameters are the pointer to the matrix v, and the subscripts of the first and last elements for the matrix[a1 .. a2][b1 .. b2]. Call this procedure to free space allocated by JS_Long_Matrix().

void JS_Free_Char_Matrix(char **v, long a1, long a2, long b1, long b2);

This procedure frees the space occupied by an matrix of char. The parameters are the pointer to the matrix v, and the subscripts of the first and last elements for the matrix[a1 .. a2][b1 .. b2]. Call this procedure to free space allocated by JS_Char_Matrix().

SEARCH

void JS_Array_Find_Float(float *v, int len, float val, int *index);

This procedure searches for a specified float value in an increasing or decreasing array. Given the array v[1..len], this procedure returns the index j such the value val is between v[index] and v[index+1]. If the value is equal to one in the vector, that index is returned. If j=0 or j=len, then x is not in the range of the vector v. JS_Fatal_Error_Handler is called is len < 2. The array must be either always increasing or always decreaing.

void JS_Array_Find_Double(double *v, int len, double val, int *index);

This procedure searches for a specified double value in an increasing or decreasing array. Given the array v[1..len], this procedure returns the index j such the value val is between v[index] and v[index+1]. If the value is equal to one in the vector, that index is returned. If j=0 or j=len, then x is not in the range of the vector v. JS_Fatal_Error_Handler is called is len < 2. The array must be either always increasing or always decreaing.

void JS_Array_Find_Int(int *v, int len, int val, int *index);

This procedure searches for a specified integer value in an increasing or decreasing array. Given the array v[1..len], this procedure returns the index j such the value val is between v[index] and v[index+1]. If the value is equal to one in the vector, that index is returned. If j=0 or j=len, then x is not in the range of the vector v. JS_Fatal_Error_Handler is called is len < 2. The array must be either always increasing or always decreaing.

void JS_Array_Find_Long(long *v, int len, long val, int *index);

This procedure searches for a specified long value in an increasing or decreasing array. Given the array v[1..len], this procedure returns the index j such the value val is between v[index] and v[index+1]. If the value is equal to one in the vector, that index is returned. If j=0 or j=len, then x is not in the range of the vector v. JS_Fatal_Error_Handler is called is len < 2. The array must be either always increasing or always decreaing.

SORT

void JS_Heap_Sort_Double(double *v, int len);

The heap sort algorithm is a fast algorithm for small arrays (under 1000 to 10000 in size). The procedure accepts an array of doubles v[1..len] and sorts them into ascending order. The result is stored in v. If len < 2, JS_Fatal_Error_Handler is called and the program is terminated.

void JS_Heap_Sort_Float(float *v, int len);

The heap sort algorithm is a fast algorithm for small arrays (under 1000 to 10000 in size). The procedure accepts an array of float v[1..len] and sorts them into ascending order. The result is stored in v. If len < 2, JS_Fatal_Error_Handler is called and the program is terminated.

void	JS_Heap_Sort_Integer(int *v, int len);

The heap sort algorithm is a fast algorithm for small arrays (under 1000 to 10000 in size). The procedure accepts an array of integers v[1..len] and sorts them into ascending order. The result is stored in v. If len < 2, JS_Fatal_Error_Handler is called and the program is terminated.

void	JS_Heap_Sort_Long(long *v, int len);

The heap sort algorithm is a fast algorithm for small arrays (under 1000 to 10000 in size). The procedure accepts an array of long v[1..len] and sorts them into ascending order. The result is stored in v. If len < 2, JS_Fatal_Error_Handler is called and the program is terminated.

void	JS_Index_Double_Array(int len, double *v, int *indx);

This procedure is similar to the sorting procedures. Instead of sorting the array, it produces an index array indx. The first element of the index (indx[1]) is the subscript of the array element that is the lowest value in v. The second element of the index is the subscript of the array element that is the second lowest value in v. And so on... v is unchanged. v[indx[j]] for j=1 to len is in ascending order.

v and indx are of size [1..len]. If len < 2, JS_Fatal_Error_Handler is called, and the program is terminanted.

void	JS_Index_Float_Array(int len, float *v, int *indx);

This is similar to JS_Index_Double_Array, except that it works on an array of float.

void	JS_Index_Integer_Array(int len, int *v, int *indx);

This is similar to JS_Index_Double_Array, except that it works on an array of float.

void	JS_Index_Long_Array(int len, long *v, int *indx);

This is similar to JS_Index_Double_Array, except that it works on an array of float.

void	JS_Quick_Sort_Int(int *v, int len);

This algorithm uses the quick sort algorithm to sort an array. It should only be used for large arrays (1000 to 10000 or greater). The procedure accepts an array of integers v[1..len] and sorts them into ascending order. The result is stored in v. If len < 2, JS_Fatal_Error_Handler is called and the program is terminated.

void   JS_Quick_Sort_Float(float *v, int len);

This algorithm uses the quick sort algorithm to sort an array. It should only be used for large arrays (1000 to 10000 or greater). The procedure accepts an array of float v[1..len] and sorts them into ascending order. The result is stored in v. If len < 2, JS_Fatal_Error_Handler is called and the program is terminated.

void   JS_Quick_Sort_Double(double *v, int len);

This algorithm uses the quick sort algorithm to sort an array. It should only be used for large arrays (1000 to 10000 or greater). The procedure accepts an array of double v[1..len] and sorts them into ascending order. The result is stored in v. If len < 2, JS_Fatal_Error_Handler is called and the program is terminated.

void   JS_Quick_Sort_Long(long *v, int len);

This algorithm uses the quick sort algorithm to sort an array. It should only be used for large arrays (1000 to 10000 or greater). The procedure accepts an array of long v[1..len] and sorts them into ascending order. The result is stored in v. If len < 2, JS_Fatal_Error_Handler is called and the program is terminated.

void   JS_Alphabetize(char **m, int siz, int len);

This procedure places strings (of maximum length len) into alphabetical order. Capital and lower case letters are treated as equal. The procedure accepts the matrix m[0..(siz-1)][0..(len-1)]. The matrix is trated an an array [0..(siz-1)] of strings. If either siz or len is less than 2, then JS_Fatal_Error_Handler is called, and the program is terminated.

STRINGS

void   JS_Char_Strip(char *str, char c, short flag);

This procedure removes occurrances of the character c in the string pointer to by char *str. If flag = JS_BEGIN (-1), c is removed from the beginning until the first non c character is located. If flag = JS_ALL (0), then all occurrances of c are removed. If flag = JS_END (1; this is default), then all c is stripped off the string from the end until the first non-c character is found. str will contain the modified string.

void   JS_String_Strip(char *txt, char *str);

This procedure strips all occurrances of the string str from the string txt. If str does not occur in txt, nothing happens. txt will contain the modified string.

void JS_String_Insert(char *txt, char *str, short loc);

This procedure places the string str into the string txt at the position loc. 0 is the first position in each string. If loc is beyond the end of the string, the string is added to the end. If it is 0 or below, it is added to the begining. No text is destroyed. All text displaced will be placed directly after str.

SUPPORT

double JS_Sign(double x, double y);

This function returns the magnitude of x with the sign of y. If y = 0, then then x is assigned a positive value.

double JS_Max(double x1, double x2);

This function returns the greater value of either x1 or x2.

double JS_Min(double x1, double x2);

This function returns the lesser value of either x1 or x2.

double JS_Mod(double x, double y);

This function returns x - (floor(x/y)*y). That's the best way I can explain the function.

double JS_Min_Array(double *v, int len);

This function returns the minimum value of all of the elements of the array v[1..len]. If len < 2, JS_Fatal_Error_Handler is called and the program is terminated.

double JS_Max_Array(double *v, int len);

This function returns the maximum value of all of the elements of the array v[1..len]. If len < 2, JS_Fatal_Error_Handler is called and the program is terminated.

double JS_Square(double x);

This function returns the square of x.

double JS_Cube(double x);

This function returns the cube of x.

double JS_General_Log(long base, double arg);

This function returns the log to the base base of arg.

void JS_Pop_Sci_Notation(double val, double *arg, long *expo);

This function converts val to scientific notation the components arg and base: arg x 10^expo, -10 < arg < 10.

void JS_Push_Sci_Notation(double arg, long expo, double *val);

This function converts the scientific notation components arg and base (arg x 10^expo) into a double precision number val.

USER UTILITIES

char JS_Check_Command_Period(EventRectord *e);

This procedure accepts an EventRecord pointer and checks the next event to see if a command period has been pressed. If it has, JS_TRUE is returned, otherwise, JS_FALSE is returned. The event will be lost.

void JS_Init_Scale_Dialog(DialogPtr dPtr, char fix);

This procedure puts a modeless dialog box on screen with a scale which will show the progress of a long computer calculation. This dialog is similar to the Mac's scale when copying files. The argument is a DialogPtr which will contain the pointer to the dialog and a character fix. This procedure does not call any of the modeless dialog box procedures (IsDialogEvent and DialogSelect), so fix needs to be JS_TRUE to display words in the dialog. If it is JS_FALSE, it is the programmer's responsibility for calling the modeless dialog procedures. If you place a dialog (or something else) over the scale dialog, it is up to you to update the dialog (by some of the modeless dialog utilities).

void JS_Update_Scale_Dialog(DialogPtr dPtr, long current, long total);

This procedure updates the progress dialog box (described in InitScaleDialog). Current is the current value of the maximum total value. If current is one half of total, then half of the scale will be filled in.

void JS_Kill_Scale_Dialog(DialogPtr dPtr);

This procedure releases the resources and memory take up by the scale dialog box.

Items Excluded in Non-Registered Version

COMPLEX

JS_Complex_Sqrt
JS_Complex_Exp

LINEAR ALGEBRA

JS_Handle_Eigenvalue
JS_Matrix_Inverse
JS_Solve_Linear_System

SORT

JS_Alphabetize

STRINGS

JS_Char_Strip
JS_String_Strip
JS_String_Insert

SUPPORT

JS_Min_Array
JS_Max_Array
JS_General_Log

Revision History

version 1.0 (September 11, 1990): Original release.

version 1.01 (September 12, 1990): JS_Check_Command_Period now returns correct values (TRUE for a stop request and FALSE for all other events; the original version had it reversed).

version 1.02 (September 15, 1990): Corrects resource for scale by changing cancel contouring to cancel process.

version 1.1 (November 21, 1990): Changes some procedures in Support to macros, but still contains the procedures (in lowercase type). Adds macros for random numbers. Library with code for 68020 and 68881. Other minor changes.