MathArray

Inherits From: NSObject

Conforms To: NSObject (NSObject), NSCoding, NSCopying

Declared In: MathArray/MathArray.h

Class Description

MathArray is a general class for performing mathematical operations on arrays (vectors, matrices, etc) of values. It can operate on any standard 'C' number type plus numbers of complex type. MathArray is a implemented using "class cluster" concept, allowing one to perform mathematical calculations on a number without necessarily being aware of what type (class) of number is being operated on. MathArray knows implicitly what types of operations can be performed on what types of numbers and will automatically cast itself to the correct number type representation to handle the specific operation. One can also explicitly cast the MathArray to a specific type (see method descriptions below) ± effectively changing the number class.

As with standard Objective-C convention, all operations are performed on the MathArray being passed the message (except where noted in the documentation below). Thus the statement

```
c = [b maAdd:[NSNumber numberWithDouble:2.0]]
```

adds 2.0 to the array `b` (replacing whatever was previously stored in `b`). The array `c` is then set equal to `b` (i.e. they are exactly the same object). If you want MathArray to act more like the typical math expression,

c = b + 2.0

where b is unchanged, and c is an array which is equal to the array b with 2.0 added to it, you need to explicitly make a temporary copy of `b` (e.g. `[[b copy] autorelease]`) before performing the operation, or you can use the macro MA_TEMPORARY (which does the same thing):

```
c = [MA_TEMPORARY(b) maAdd:[NSNumber numberWithDouble:2.0]]
```

This would allow you to cascade operations like this:

```
number2 = [NSNumber numberWithDouble:2.0];
c = [[[MA_TEMPORARY(b) maAdd:number2] maMult:b] maMult:number2]
```

Subclassing and Adding Features

To perform more general calculations, or to perform them faster, write them into a function and use the MathArray method **performFunction:userInfo:** to call that function for each element in the array. If you need faster performance,
generally the best way to add features to MathArray is to add a category to the type of number you wish to add features to (say MathFloatArray). MathArray will automatically search for the class that responds to a particular message, and re-cast itself to that type (if possible) in order to perform the new operation. You can also add a new subclass (if you think of a new type of number that I haven't!), by registering it with MathArray's

registerArraySubclass: method. It is likely you will have to override all methods of the super-class in order to deal with your new number type. It is also likely that, since other classes won't know about your class, they won't know how to cast 'values' from your new class, so in essence, they won't be able to interact. This is ok as long as you don't need them too.

Notes on Precision and Promotion

MathArray allows one to operate with numbers and arrays of number while almost ignoring the way numbers are represented. Problems can occur when MathArray tries to operate on two numbers (or arrays) with different precision. For instance, what happens when you add a short to an unsigned long? What type should the result be? MathArray makes the bold assumption that signed data is more 'precise' than unsigned data, so in this case (surprise!), the result is a signed long. Even this can cause problems if the unsigned long overflows its signed long value. You don't like the way this works? Then give me a good reason to change it, or make your own MathArray subclass and rewrite **resolvedTypecast::**. Even better, always make sure you work with types that are well within your desired precision range.

Exceptions

Exception and signal handling is a rather difficult problem. Most architectures allow you to trap math errors only by setting a machine-specific (usually in assembly language) trap flag. Until I figure out the assembly language calls for more architecutres, math error reporting will be rather general. Currently, MathArray checks every number in its array during an operation for an infinite or NaN value, and raises an exception AFTER the entire process is complete. You can catch this exception normally, or choose to ignore it using the MathArray method **setTrap:action:**. Eventually, with better support for singal handling, one could use **setTrap:action:** to selectively ignore (mostly benign) exceptions ± i.e. some exceptions won't be raised if there is no need to handle them (this would cause unexpected results, like overflow errors, but maybe you don't care about that?).

Instance Variables

MAMutableValueData***data**;
MAValueData* **size**;

```
unsigned dimension;
struct _math_flags {
        unsigned zero:1;
        unsigned promote:1;
        unsigned reserved:3;
} math_flags;
```

data	The object that stores the array data.
size	A data array of length **dimension**, where the *i*th element represents the size of the *i*th dimension. data is unsigned.
dimension	The dimension of the array.

Method Types

Helping resolve casting problems	+ resolvedTypecast::
Characteristics of the class	+ promoteToDouble + registerArraySubclass: + setTrap:action: + willPromoteToDouble
Creating instances	- initArrayFrom:ofDimension:size:objCType: - initArrayOfDimension:size:objCType:zero: + maWithMatrixWithCols:rows: + maWithScalar:

	+ maWithVector:
Setting instance characteristics	- promoteIfNeeded: - isPromotable - castToObjCType:
Information about arrays	- dimension - mathData ± objcCType - precision - sizes
Accessing data within the array	- arraySubrange: ± arrayValueAtIndex: - arrayValues: - concatArray: ± setArray:atIndex: ± setValue:atIndex; ± setValues:atLocations: - reformArrayToDimension:size:
Operating on the array	± maOperate:with: - maPerform: ± maPerformFunction:userInfo:

Class Methods

maMatrixWithCols:rows:

+ (MathArray *)**maMatrixWithCols:**(unsigned)*cols*

rows:(unsigned)*rows*
objCType:(const char *)*theType*

Creates and returns a two-dimensional matrix of the proper size. If *theType* is NULL, then the type of matrix remains indeterminate until the first operation is perfomred on it.

See also: ± initArrayOfDimension:size:objCType:zero:

maWithScalar:

+ (MathArray *)**maWithScalar:**(NSValue *)*value*

Creates and returns a MathArray representing the scalar *value*. This is a special case of an array with a dimension of zero (but a size of 1).

See also: ± initArrayOfDimension:size:objCType:zero:

maWithVector:

+ (MathArray *)**maWithVector:**(unsigned)*size*
objCType:(const char *)*theType*

Creates and returns a one-dimensional vector of length size. If *theType* is NULL, then the type of matrix remains indeterminate until the first operation is perfomred on it.

See also: ± initArrayOfDimension:size:objCType:zero:

promoteToDouble:

+ (void)**promoteToDouble:**(BOOL)*flag*

If *flag* is yes, then when MathArray is forced to promote an array to float (complex float) type, it will promote it

instead to double (complex double) type. This increases the precision of the calculations at the expense of doubling the size of the array.

See also: ± promoteIfNeeded:

resolvedTypecast::

+ (const char *)**resolvedTypecast:**(const char *)*firstType*
:(const char *)*secondType*

Tries to pick the type with the best precicion. Currently it picks the type with the higher precision, unless there is a conflict between signed and unsigned types. In this case, it picks the signed version of the higher precision type.

See also: ± castToObjCType:, **± maOperate:with:**

registerArraySubclass:

+ (void)**registerArraySubclass:**(MathArray *)*subclass*

Registers the *subclass* as a numeric type that MathArray can search to find a given operation. If MathArray is sent a message that is only defined in this subclass, it will automatically promote itself to this type of subclass in order to perform the operation. *Subclass* should especially override the method **precision** to return a unique number which places the class in relation to the other classes based on the 'precision' of the representation (e.g. float is considered more precise then integer). When searching each subclass for a method, MathArray searches the list of subclasses in order of increasing precision.

See also: ± precision:

setTrap:action:

+ (void)**setTrap:**(unsigned)*mathTrap*
action:(ExceptionMask)*mask*

Determines the action taken (as defined by the **MaskedException** enumerated type *ExceptionMask*) when a math error occurs. The value *mathTrap* is a bitwise OR'd combination of any of the *ma_trap_t* enumerated types (see the MathArray header file for a list of these types). One can use this method to selectively ignore or catch specific math erros such as underflow or division by 0.

See also:

willPromoteToDouble

+ (BOOL)**willPromoteToDouble**

Returns YES if arrays are normally promoted to double instead of float.

See also: + promoteToDouble:

Instance Methods

arraySubrange:

- (MathArray *)**arraySubrange:**(NSRange *)*arrayRange*

Creates and returns a new array which is a subset of the original array based on the list of *arrayRange* parameters (one for each dimension). Raises an **MARangeException** if the list of ranges don't match the number of dimensions in the array or if any of the specified ranges lies outside the limits of the array.

See also:

arrayValueAtIndex:

- (id <NSNumber,ComplexNumber>)**arrayValueAtIndex:**(unsigned *)*index*

Returns a new number whose values corresponds to the value of the original array at the location specified by *index*. The *i*th value of index specifies the index of the *i*th dimension of the array. Raises an **MARangeException** if the *index* array lies outside the limit of the array.

See also:

arrayValues:

- (MathArray *)**arrayValues:**(MathArray *)*arrayLocations*

Creates and returns a new array whose values corresponds to the values of the original array at the locations specified by *arrayLocations*. Each value in *arrayLocations* is an *orderedIndex*. An *orderedIndex* is the one-dimensional equivalent index of the array, i.e.

orderedIndex = *index*[1] * *size*[0]^(*dimensions*-1) + *index*[2] * *size*[1]^(*dimensions*-2) + ...

where *index* is the array of index locations from *arrayLocations*, and *size* is the array of sizes for each *dimension* of the array. The *arrayLocations* can be of any type, but it is converted to an unsigned type before determining which locations each value specifies. Raises an **MARangeException** if any of the array locations are outside the limit of the array.

See also: - **setValues:atLocations:**

castToObjCType:

- **castToObjCType:**(const char *)*newType*

Forces the array to be cast to the specified type. Precision may be lost if the array is cast to a type with less precision (sort of obvious, huh?). This method works even if the array is not promotable.

See also: ± isPromotable

concatArray:

- **concatArray:**(MathArray *)*otherArray*

Concatenates *otherArray* to the end of the receiver. Raises an **MADimensionException** if the dimensions of *otherArray* are incompatible with the receiver.

See also:

dimension

- (unsigned)**dimension**

Returns the dimension of the array.

See also: ± sizes

initArrayFrom:ofDimension:size:objCType:

- **initArrayFrom:**(NSData *)*data*
  ofDimension:(unsigned)*numDimensions*
  size:(const unsigned *)*sizes*
  objCType:(const char *)*type*

Initializes a MathArray from data stored in the NSData object *data*. The array will be treated as an array of the specified dimension and sizes. The *data* object should contain values of the Objective-C type *type*. A

MAParameterException is raised if the specified *sizes* of the array don't match the size of the data stored in *data*. This is the designated initializer for the MathArray class. If *data* is nil, this method will act just like **initArrayOfDimension:size:objCType:zero:**. If *type* is NULL, the array will initially be of indeterminate type until the first mathematical operation is performed on it.By default, the array is specified as *promotable*, which means the array will automatically be re-cast to the appropriate (more precise) type for a given mathematical operation if needed.

See also:

initArrayOfDimension:size:objCType:zero:

- **initArrayOfDimension:**(unsigned)*numDimensions*
  size:(const unsigned *)*sizes*
  objCType:(const char *)*type*
  zero:(BOOL)*doZero*

Initializes a MathArray with the specified dimension and sizes. The array will (at least initially) contain numbers of the Objective-C type *type*. If *doZero* is set, then all array values are set to zero, otherwise array values may be undetermined (and even invalid).

See also: ± initArrayFrom:ofDimension:size:objCType:, ± isPromotable

isPromotable

- (BOOL)**isPromotable**

Returns YES if the array is promotable..

See also: ± setPromotable

maOperate:with:

- **maOperate:***(*ma_operator_t*)operator* **with:***value*

Performs the logical or mathematical operation (specified by *operator*)on the reciever using *value*, which may be an number or a MathArray. If *value* is a scalar, then *value* used on each element of the receiver array. If *value* is a MathArray, then element 1 of *value* is operated with element 1 of the receiver, and so on. Raises an **MASizeException** or **MADimensionException** if the sizes or the the dimensions of the two arrays don't match. Raises an **MAFloatingPointException** if a floating-point exception occurred during the processing.

See also:

maPerform:

- **maPerform:**(double (*)*(double))mathFunction*

Performs the function *mathFunction* on each element of the receiver. The receiver is automatically promoted to at least a float if necessary (and if possible). Raises an **MAFloatingPointException** if a floating-point exception occurs during the operation.

See also:

maPerformFunction:useInfo:

- **maPerformFunction:**(perform_func_t*)perform_func*
  userInfo:(void *)*info*

For each element of the receiver, calls the function *perform_func*. The functions is passed a number which contains the value of the current element, an array containing the current index of the element, and a pointer to user supplied values given in *info*. The function should return a new number which contains the result of the operation. Raises an **MAFloatingPointException** if a floating-point exception occurs during the operation. The fucntion definition *perform_func_t* is defined as "id <NSNumber,ComplexNumber>(***perform_func_t**) (id

<NSNumber,ComplexNumber>, unsigned *_index_, void *_info_);"

See also:

mathData

- (MAMutableValueData *)**mathData**

Returns the MAMutableValueData object which contains the values stored in the receiver.

See also:

objCType

- (const char *)**objCType**

Returns the current type/class of number that the array stores (as if using the Objective-C keyword @encode)

See also:

precision

- (unsigned)**precision**

Returns a unique number which specifies the precision of the numbers stored by the receiver.

See also:

promoteIfNeeded:

- (void)**promoteIfNeeded:**(BOOL)*doPromote*

By default, MathArrays will all promote themselves to the numerical type required for the precision of the mathematical operation. If *doPromote* is set to NO, then the array will not be promoted, even if this results in errornous results for the mathematical operation. For instance taking the cosine of a non-promotable integer array will return 0 for every element value (depending on the underlying math library). Generally this is a bad idea unless you are performing specific math operations that don't depend on precision.

See also:

reformArrayToDimension:size:

- **reformArrayToDimension:**(unsigned)*newDimension*
  size:(unsigned *)*newSizes*

Treats the receiver as if it had the dimension and sizes specified. Raises an **MAParameterException** if the new dimension and sizes are incompatible with the actual size of the array. If *newDimension* is 0, then all dimensions of size = 1 are removed (e.g. a 1×100 element "two-dimensional" array would be reformed to a 100 element one-dimensional array).

See also:

setArray:atIndex:

- (void) **setArray:**(MathArray *)*otherArray*
  atIndex:(unsigned *)*startIndex*

Replaces the subarray in the reciever starting at the specified *startIndex* by *otherArray*.

See also:

setValue:atIndex:

- (void) **setValue:** *value*
  atIndex:(unsigned *)*index*

Replaces the value in the reciever at the specified *index* by *value*.

See also:

setValues:atLocations:

- (void) **setValues:**(MathArray *)*otherArray*
  atLocations:(MathArray *)*arrayLocations*

Assuming that *arrayLocations* specifies ordered index locations in the receiver, replace the value at each location with the value specified in *otherArray*.

See also: - **arrayValues:**

sizes

- (const unsigned *)**sizes**

Returns a list which contains the sizes of each dimension of the receiver.

See also: