# MATLAB®

## C/C++, Fortran, Java, and Python API Reference

MATLAB®

R2021a

MathWorks®

## How to Contact MathWorks


Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098


*MATLAB® C/C++, Fortran, Java®, and Python® API Reference*

**Revision History**

| | | |
|---|---|---|
| December 1996 | First Printing | New for MATLAB 5 (Release 8) |
| May 1997 | Online only | Revised for MATLAB 5.1 (Release 9) |
| January 1998 | Online Only | Revised for MATLAB 5.2 (Release 10) |
| January 1999 | Online Only | Revised for MATLAB 5.3 (Release 11) |
| September 2000 | Online Only | Revised for MATLAB 6.0 (Release 12) |
| June 2001 | Online only | Revised for MATLAB 6.1 (Release 12.1) |
| July 2002 | Online only | Revised for MATLAB 6.5 (Release 13) |
| January 2003 | Online only | Revised for MATLAB 6.5.1 (Release 13SP1) |
| June 2004 | Online only | Revised for MATLAB 7.0 (Release 14) |
| October 2004 | Online only | Revised for MATLAB 7.0.1 (Release 14SP1) |
| March 2005 | Online only | Revised for MATLAB 7.0.4 (Release 14SP2) |
| September 2005 | Online only | Revised for MATLAB 7.1 (Release 14SP3) |
| March 2006 | Online only | Revised for MATLAB 7.2 (Release 2006a) |
| September 2006 | Online only | Revised for MATLAB 7.3 (Release 2006b) |
| March 2007 | Online only | Revised and renamed for MATLAB 7.4 (Release 2007a) |
| September 2007 | Online only | Revised and renamed for MATLAB 7.5 (Release 2007b) |
| March 2008 | Online only | Revised and renamed for MATLAB 7.6 (Release 2008a) |
| October 2008 | Online only | Revised and renamed for MATLAB 7.7 (Release 2008b) |
| March 2009 | Online only | Revised for MATLAB 7.8 (Release 2009a) |
| September 2009 | Online only | Revised for MATLAB 7.9 (Release 2009b) |
| March 2010 | Online only | Revised and renamed for MATLAB 7.10 (Release 2010a) |
| September 2010 | Online only | Revised for MATLAB 7.11 (Release 2010b) |
| April 2011 | Online only | Revised for MATLAB 7.12 (Release 2011a) |
| September 2011 | Online only | Revised for MATLAB 7.13 (Release 2011b) |
| March 2012 | Online only | Revised for MATLAB 7.14 (Release 2012a) |
| September 2012 | Online only | Revised for MATLAB 8.0 (Release 2012b) |
| March 2013 | Online only | Revised for MATLAB 8.1 (Release 2013a) |
| September 2013 | Online only | Revised for MATLAB 8.2 (Release 2013b) |
| March 2014 | Online only | Revised for MATLAB 8.3 (Release 2014a) |
| October 2014 | Online only | Revised for MATLAB 8.4 (Release 2014b) |
| March 2015 | Online only | Revised for MATLAB 8.5 (Release 2015a) |
| September 2015 | Online only | Revised for MATLAB 8.6 (Release 2015b) |
| March 2016 | Online only | Revised for MATLAB 9.0 (Release 2016a) |
| September 2016 | Online only | Revised for MATLAB 9.1 (Release 2016b) |
| March 2017 | Online only | Revised for MATLAB 9.2 (Release 2017a) |
| September 2017 | Online only | Revised for MATLAB 9.3 (Release 2017b) |
| March 2018 | Online only | Revised for MATLAB 9.4 (Release 2018a) |
| September 2018 | Online only | Revised for MATLAB 9.5 (Release 2018b) |
| March 2019 | Online only | Revised for MATLAB 9.6 (Release 2019a) |
| September 2019 | Online only | Revised for MATLAB 9.7 (Release 2019b) |
| March 2020 | Online only | Revised for MATLAB 9.8 (Release 2020a) |
| September 2020 | Online only | Revised for MATLAB 9.9 (Release 2020b) |
| March 2021 | Online only | Revised for MATLAB 9.10 (Release 2021a) |

# Contents

1

# API Reference

# matlab::data::ArrayDimensions

Type specifying array dimensions

## Description

Use the `ArrayDimensions` type to specify the size of an array. `ArrayDimensions` is specified as:

```
using ArrayDimensions = std::vector<size_t>;
```

## Free Function

### `getNumElements`

```
inline size_t getNumElements(const ArrayDimensions& dims)
```

#### Description

Determine the number of elements based on the `ArrayDimensions`.

#### Parameters

| | |
|---|---|
| `const ArrayDimensions& dims` | Array dimensions. |

#### Returns

| | |
|---|---|
| `inline size_t` | Number of elements. |

#### Throws

None

## See Also

**Topics**
"MATLAB Data API Types"

**Introduced in R2017b**

# matlab::data::ArrayFactory

C++ class to create arrays

## Description

Use `ArrayFactory` to create `matlab::data::Array` objects.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `ArrayFactory.hpp` |

## Constructors

### Default Constructor

```
ArrayFactory()
```

#### Throws

| | |
|---|---|
| `matlab::data::FailedToLoadLibMatlabDataArrayException` | Concrete implementation not loaded. |

## Destructor

```
~ArrayFactory()
```

## Member Functions

- “createArray” on page 1-3
- “createScalar” on page 1-5
- “createCellArray” on page 1-5
- “createCharArray” on page 1-6
- “createStructArray” on page 1-7
- “createEnumArray” on page 1-8
- “createSparseArray” on page 1-8
- “createEmptyArray” on page 1-9
- “createBuffer” on page 1-10
- “createArrayFromBuffer” on page 1-10

### `createArray`

```
template <typename T>
TypedArray<T> createArray(ArrayDimensions dims)

template <typename ItType, typename T>
TypedArray<T> createArray(ArrayDimensions dims,
```

```
    ItType begin,
    ItType end)

template <typename T>
TypedArray<T> createArray(ArrayDimensions dims,
    const T* const begin,
    const T* const end)

template <typename T>
TypedArray<T> createArray(ArrayDimensions dims,
    std::initializer_list<T> data)
```

**Description**

Creates a `TypedArray<T>` with the given dimensions. If specified, `createArray` fills the array with data. The data is copied and must be in column-major order.

**Template Parameters**

- `ItType` - Iterator types, specified as `std::iterator`.
- `T` - Element types, specified as one of the following C++ data types.

| | | | | | |
|---|---|---|---|---|---|
| `bool` | `int8_t` | `int16_t` | `int32_t` | `int64_t` | `uint8_t` |
| `uint16_t` | `uint32_t` | `uint64_t` | `float` | `double` | `char16_t` |
| `matlab::data::String` | `std::complex<double>` | `std::complex<float>` | `std::complex<int8_t>` | `std::complex<uint8_t>` | `std::complex<int16_t>` |
| `std::complex<uint16_t>` | `std::complex<int32_t>` | `std::complex<uint32_t>` | `std::complex<int64_t>` | `std::complex<uint64_t>` | `matlab::data::MATLABString` |

To create an array of `matlab::data::Object` element types, use the `TypedArray<T> createArray(ArrayDimensions dims, ItType begin, ItType end)` syntax.

**Parameters**

| | |
|---|---|
| `ArrayDimensions dims` | Dimensions for the array. |
| `ItType begin`<br>`ItType end` | Start and end of the user supplied data. The `value_type` of the iterator determines the data type. |
| `const T* const begin`<br>`const T* const end` | Start and end of the user supplied data specified as C-style pointer. This syntax supports all primitive types, complex types, and string types. |
| `std::initializer_list<T>` | Initializer list containing the data. |

**Throws**

| | |
|---|---|
| `matlab::OutOfMemoryException` | Unable to allocate the array. |
| `matlab::data::NumberOfElementsExceedsMaximumException` | Number of elements is greater than `size_t`. |
| `matlab::data::InvalidArrayTypeException` | Input type of `matlab::data::ObjectArray` does not match the type of `TypedArray<T>`. |

## createScalar

```
template <typename T>
TypedArray<T> createScalar(const T val)

TypedArray<String> createScalar(const String val)

TypedArray<String> createScalar(const std::string val)

ObjectArray createScalar(const Object& val);
```

#### Description

Creates a scalar `TypedArray<T>` with the given value. This method supports arithmetic types, complex types, and string types.

#### Parameters

| | |
|---|---|
| `const T val`<br>`const String val`<br>`const std::string val`<br>`const Object& val` | Value to be inserted into the scalar.<br><br>For `std::string` parameters, if `val` is 7-bit ASCII data, then the method converts it to UTF16. |

#### Throws

| | |
|---|---|
| `matlab::OutOfMemoryException` | Unable to allocate the array. |
| `matlab::data::NonAsciiCharInInputDataException` | Input is `std::string` and contains non-ASCII characters. |

#### Examples

```
#include "MatlabDataArray.hpp"

int main() {
    matlab::data::ArrayFactory factory;

    // Create a vector containing two scalar values
    std::vector<matlab::data::Array> args({
    factory.createScalar<int16_t>(100),
    factory.createScalar<int16_t>(60)});
    return 0;
}
```

#### Related Topics

"Call Function with Single Returned Argument"

## createCellArray

```
CellArray createCellArray(ArrayDimensions dims)

template <typename ...Targs>
CellArray createCellArray(ArrayDimensions dims, Targs... data)
```

#### Description

Creates a `CellArray` with the specified `data`. The data is in column-major order.

#### Template Parameters

| | |
|---|---|
| `...Targs` | Variadic template of:<br>• arithmetic type<br>• complex type<br>• `matlab::data::String`<br>• `std::string`<br>• `matlab::data::Array` |

#### Parameters

| | |
|---|---|
| `ArrayDimensions dims` | Dimensions of the cell array. |
| `Targs... data` | Elements to be inserted into the cell array, specified as a primitive complex type, string, or `Array`. |

#### Throws

| | |
|---|---|
| `matlab::OutOfMemoryException` | Unable to allocate the array. |
| `matlab::data::NonAsciiCharInInputDataException` | Input is `std::string` and contains non-ASCII characters. |
| `matlab::data::NumberOfElementsExceedsMaximumException` | Number of elements is greater than `size_t`. |

#### Examples

```
#include "MatlabDataArray.hpp"

int main()
{
    using namespace matlab::data;
    ArrayFactory f;
    CellArray myArray = f.createCellArray({ 1,2 },
        f.createCharArray("MATLAB Cell Array"),
        f.createArray<double>({ 2,2 }, { 1.2, 2.2, 3.2, 4.2 }));

    return 0;
}
```

### createCharArray

```
CharArray createCharArray(String str)
```

```
CharArray createCharArray(std::string str)
```

#### Description

Creates a 1xn `CharArray` from the specified input, where n is the string length.

#### Parameters

| | |
|---|---|
| `matlab::data::String str` | Data to be filled into the array. |
| `std::string str` | |

**Throws**

| | |
|---|---|
| `matlab::OutOfMemoryException` | Unable to allocate the array. |
| `matlab::data::NonAsciiCharInInputDataException` | Input is `std::string` and contains non-ASCII characters. |

**Examples**

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    CharArray A = factory.createCharArray("This is a char array");
    return 0;
}
```

### createStructArray

```
StructArray createStructArray(ArrayDimensions dims,
    std::vector<std::string> fieldNames)
```

**Description**

Creates a `StructArray` with the given dimensions and field names.

**Parameters**

| | |
|---|---|
| `ArrayDimensions dims` | Dimensions for the array. |
| `std::vector<std::string> fieldNames` | Vector of the field names for the structure. |

**Throws**

| | |
|---|---|
| `matlab::OutOfMemoryException` | Unable to allocate the array. |
| `matlab::data::DuplicateFieldNameInStructArrayException` | Duplicate field names specified. |
| `matlab::data::NumberOfElementsExceedsMaximumException` | Number of elements is greater than `size_t`. |

**Examples**

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory f;

    // Create StructArray equivalent to MATLAB structure s:
    // s = struct('loc', {'east', 'west'}, 'data', {[1, 2, 3], [4., 5., 6., 7., 8.]})
    StructArray S = f.createStructArray({ 1,2 }, { "loc", "data" });
    S[0]["loc"] = f.createCharArray("east");
    S[0]["data"] = f.createArray<uint8_t>({ 1, 3 }, { 1, 2, 3 });
    S[1]["loc"] = f.createCharArray("west");
    S[1]["data"] = f.createArray<double>({ 1, 5 }, { 4., 5., 6., 7., 8. });

    // Access the value defined by the MATLAB statement:
    // s(1).data
    Reference<Array> val = S[0]["data"];

    return 0;
}
```

### createEnumArray

```
EnumArray createEnumArray(ArrayDimensions dims,
    std::string className,
    std::vector<std::string> enums)
```

```
EnumArray createEnumArray(ArrayDimensions dims,
    std::string className)
```

#### Description

Creates an `EnumArray` of type `className`, which is a defined class. If specified, the method initializes the array with the list of enumeration names.

#### Parameters

| | |
|---|---|
| `ArrayDimensions dims` | Dimensions for the array. |
| `std::string className` | Class name of the enumeration array. |
| `std::vector<std::string> enums` | List of the enumeration names. |

#### Throws

| | |
|---|---|
| `matlab::OutOfMemoryException` | Unable to allocate the array. |
| `matlab::data::MustSpecifyClassNameException` | Class name not specified. |
| `matlab::data::WrongNumberOfEnumsSuppliedException` | Wrong number of enumerations provided. |
| `matlab::data::NumberOfElementsExceedsMaximumException` | Number of elements is greater than `size_t`. |

#### Examples

```
#include "MatlabDataArray.hpp"
#include <vector>

int main()
{
    using namespace matlab::data;
    ArrayFactory f;
    auto blue = f.createEnumArray({ 1,1 }, "TextColor", { "Blue" });

    // Create an argument vector
    std::vector<Array> args({ f.createCharArray("My text"), std::move(blue) });

    return 0;
}
```

### createSparseArray

```
template <typename T>
SparseArray<T> createSparseArray(ArrayDimensions dims,
    size_t nnz,
    buffer_ptr_t<T> data,
    buffer_ptr_t<size_t> rows,
    buffer_ptr_t<size_t> cols)
```

#### Description

Creates a `SparseArray<T>` with `rows`-by-`cols` dimensions. You can only have two dimensions for sparse arrays. The method does not copy the buffer and the array takes ownership of the memory.

#### Template Parameters

| | |
|---|---|
| `T` | Element types, specified as `double`, `bool`, or `std::complex<double>`. |

#### Parameters

| | |
|---|---|
| `ArrayDimensions dims` | Dimensions for the array. |
| `size_t nnz` | Number of nonzero elements. |
| `buffer_ptr_t<T> data` | Buffer containing the nonzero elements. |
| `buffer_ptr_t<size_t> rows` | Buffer containing the row value for each element. |
| `buffer_ptr_t<size_t> cols` | Buffer containing the column value for each element. |

#### Throws

| | |
|---|---|
| `matlab::OutOfMemoryException` | Unable to allocate the array. |
| `matlab::data::InvalidDimensionsInSparseArrayException` | More than two dimensions specified. |
| `matlab::data::NumberOfElementsExceedsMaximumException` | Number of elements is greater than `size_t`. |

#### Examples

```
#include "MatlabDataArray.hpp"

int main() {
    std::vector<double> data = { 3.5, 12.98, 21.76 };
    std::vector<size_t> rows = { 0,0,1 };
    std::vector<size_t> cols = { 0,4,8 };
    size_t nnz = 3;

    matlab::data::ArrayFactory factory;
    auto data_p = factory.createBuffer<double>(nnz);
    auto rows_p = factory.createBuffer<size_t>(nnz);
    auto cols_p = factory.createBuffer<size_t>(nnz);

    double* dataPtr = data_p.get();
    size_t* rowsPtr = rows_p.get();
    size_t* colsPtr = cols_p.get();
    std::for_each(data.begin(), data.end(), [&](const double& e) { *(dataPtr++) = e; });
    std::for_each(rows.begin(), rows.end(), [&](const size_t& e) { *(rowsPtr++) = e; });
    std::for_each(cols.begin(), cols.end(), [&](const size_t& e) { *(colsPtr++) = e; });

    matlab::data::SparseArray<double> arr =
        factory.createSparseArray<double>({ 2,9 }, nnz, std::move(data_p),
            std::move(rows_p), std::move(cols_p));
    return 0;
}
```

### `createEmptyArray`

`Array createEmptyArray()`

#### Descriptions

Creates an empty `Array` containing no elements.

#### Returns

| | |
|---|---|
| `Array` | Empty array. |

**Throws**

| | |
|---|---|
| `matlab::OutOfMemoryException` | Unable to allocate the array. |

### createBuffer

```
template <typename T>
buffer_ptr_t<T> createBuffer(size_t numberOfElements)
```

**Description**

Creates an uninitialized buffer to pass to the `createArrayFromBuffer` method.

**Template Parameters**

| | |
|---|---|
| `T` | Primitive types. |

**Parameters**

| | |
|---|---|
| `size_t numberOfElements` | Number of elements, not the actual buffer size. |

**Returns**

| | |
|---|---|
| `buffer_ptr_t<T>` | Unique_ptr containing the buffer. |

**Throws**

| | |
|---|---|
| `matlab::OutOfMemoryException` | Unable to allocate the array. |

### createArrayFromBuffer

```
template <typename T>
TypedArray<T> createArrayFromBuffer(ArrayDimensions dims,
    buffer_ptr_t<T> buffer,
    MemoryLayout memoryLayout = MemoryLayout::COLUMN_MAJOR)
```

**Description**

Creates a `TypedArray<T>` using the given buffer.

**Template Parameters**

| | |
|---|---|
| `T` | Primitive types. |

**Parameters**

| | |
|---|---|
| `ArrayDimensions dims` | Dimensions for the array. |
| `buffer_ptr_t<T> buffer` | Buffer containing the data. The buffer is not copied. The `TypedArray<T>` object takes ownership of the buffer. |
| `MemoryLayout memoryLayout` | Memory layout for input buffer, specified as `MemoryLayout::COLUMN_MAJOR` or as `MemoryLayout::ROW_MAJOR`. The default layout is `COLUMN_MAJOR`. This parameter is optional. |

**Throws**

| | |
|---|---|
| `matlab::OutOfMemoryException` | Unable to allocate the array. |
| `matlab::data::InvalidArrayTypeException` | Buffer type not valid. |
| `matlab::data::InvalidMemoryLayoutException` | Invalid memory layout. |
| `matlab::data::InvalidDimensionsInRowMajorArrayException` | Dimensions not valid. This exception occurs for arrays created with MATLAB R2019a and R2019b if a row-major array is not 2-D. |
| `matlab::data::NumberOfElementsExceedsMaximumException` | Number of elements is greater than `size_t`. |

## See Also

**Introduced in R2017b**

# matlab::data::Reference<Array>

C++ class to get reference to `Array`

## Description

Use the `Reference<Array>` class to get a reference to an `Array` element of a container object, such as a MATLAB structure or cell array. The class is a base class for all reference types that refer to arrays and provides basic array information. `ArrayRef` is defined as:

```
using ArrayRef = Reference<Array>;
```

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `ArrayReferenceExt.hpp` |

## Member Functions


- "getType" on page 1-12
- "getDimensions" on page 1-12
- "getNumberOfElements" on page 1-13
- "isEmpty" on page 1-13


### getType

```
ArrayType getType() const
```

#### Returns

| | |
|---|---|
| `ArrayType` | Type of the array |

#### Throws

| | |
|---|---|
| `matlab::data::NotEnoughIndicesProvidedException` | Not enough indices provided. |
| `matlab::data::InvalidArrayIndexException` | Index provided is not valid for this `Array` or one of the indices is out of range. |
| `matlab::data::InvalidArrayTypeException` | `Array` type not recognized. |

### getDimensions

```
ArrayDimensions getDimensions() const
```

#### Returns

| | |
|---|---|
| `ArrayDimensions` | `Array` dimensions vector. |

**Throws**

| | |
|---|---|
| matlab::data::NotEnoughIndicesProvidedException | Not enough indices provided. |
| matlab::data::InvalidArrayIndexException | Index provided is not valid for this Array or one of the indices is out of range. |

### getNumberOfElements

```
size_t getNumberOfElements() const
```

**Returns**

| | |
|---|---|
| size_t | Number of elements in array. |

**Throws**

| | |
|---|---|
| matlab::data::NotEnoughIndicesProvidedException | Not enough indices provided. |
| matlab::data::InvalidArrayIndexException | Index provided is not valid for this Array or one of the indices is out of range. |

### isEmpty

```
bool isEmpty() const
```

**Returns**

| | |
|---|---|
| bool | Returns true if array is empty, otherwise returns false. |

**Throws**

| | |
|---|---|
| matlab::data::NotEnoughIndicesProvidedException | Not enough indices provided. |
| matlab::data::InvalidArrayIndexException | Index provided is not valid for this Array or one of the indices is out of range. |

## Free Functions

- “getReadOnlyElements” on page 1-13
- “getWritableElements” on page 1-14

### getReadOnlyElements

```
template <typename T>
Range<TypedIterator, T const> getReadOnlyElements(const Reference<Array>& ref)
```

**Description**

Get a range containing the elements of the Array or Reference<Array>. Iterators contained in the range are const.

**Parameters**

| | |
|---|---|
| const Reference<Array>& ref | Reference<Array>. |

**Returns**

| | |
|---|---|
| `Range<TypedIterator, T const>` | Range containing `begin` and `end` iterators for the elements of the input `Reference<Array>`. |

**Throws**

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | `Array` does not contain type `T`. |

### `getWritableElements`

```
template <typename T>
Range<TypedIterator, T> getWritableElements(Reference<Array>& ref)
```

**Description**

Get a range containing the elements of the `Array` or `Reference<Array>`. Iterators contained in the range are non-`const`.

**Parameters**

| | |
|---|---|
| `Reference<Array>& ref` | `Reference<Array>`. |

**Returns**

| | |
|---|---|
| `Range<TypedIterator, T>` | Range containing `begin` and `end` iterators for the elements of the input `Reference<Array>`. |

**Throws**

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | `Array` does not contain type `T`. |

## See Also

`ArrayType`

**Introduced in R2017b**

# matlab::data::ArrayType

C++ array type enumeration class

## Description

Use `ArrayType` objects to identify the data type and other attributes of a MATLAB array.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `ArrayType.hpp` |

## Enumeration

```
enum class ArrayType {
    UNKNOWN,
    LOGICAL,
    CHAR,
    DOUBLE,
    SINGLE,
    INT8,
    UINT8,
    INT16,
    UINT16,
    INT32,
    UINT32,
    INT64,
    UINT64,
    COMPLEX_DOUBLE,
    COMPLEX_SINGLE,
    COMPLEX_INT8,
    COMPLEX_UINT8,
    COMPLEX_INT16,
    COMPLEX_UINT16,
    COMPLEX_INT32,
    COMPLEX_UINT32,
    COMPLEX_INT64,
    COMPLEX_UINT64,
    CELL,
    STRUCT,
    VALUE_OBJECT,
    HANDLE_OBJECT_REF,
    ENUM,
    SPARSE_LOGICAL,
    SPARSE_DOUBLE,
    SPARSE_COMPLEX_DOUBLE,
    MATLAB_STRING
};
```

## C++ Data Type Conversion

| MATLAB ArrayType Value | C++ Type | Description |
|---|---|---|
| DOUBLE | double | double-precision (64-bit), floating-point number |
| SINGLE | float | single-precision (32-bit), floating-point number |
| INT8 | int8_t | signed 8-bit integer |
| UINT8 | uint8_t | unsigned 8-bit integer |
| INT16 | int16_t | signed 16-bit integer |
| UINT16 | uint16_t | unsigned 16-bit integer |
| INT32 | int32_t | signed 32-bit integer |
| UINT32 | uint32_t | unsigned 32-bit integer |
| INT64 | int64_t | signed 64-bit integer |
| UINT64 | uint64_t | unsigned 64-bit integer |
| CHAR | char16_t | 16-bit character |
| LOGICAL | bool | logical |
| COMPLEX_DOUBLE | std::complex<double> | complex, double-precision (64-bit), floating-point number |
| COMPLEX_SINGLE | std::complex<float> | complex, single precision (32-bit), floating-point number |
| COMPLEX_INT8 | std::complex<int8_t> | complex, signed 8-bit integer |
| COMPLEX_UINT8 | std::complex<uint8_t> | complex, unsigned 8-bit integer |
| COMPLEX_INT16 | std::complex<int16_t> | complex, signed 16-bit integer |
| COMPLEX_UINT16 | std::complex<uint16_t> | complex, unsigned 16-bit integer |
| COMPLEX_INT32 | std::complex<int32_t> | complex, signed 32-bit integer |
| COMPLEX_UINT32 | std::complex<uint32_t> | complex, unsigned 32-bit integer |
| COMPLEX_INT64 | std::complex<int64_t> | complex, signed 64-bit integer |
| COMPLEX_UINT64 | std::complex<uint64_t> | complex, unsigned 64-bit integer |
| CELL | matlab::data::Array | Array containing other Arrays |
| STRUCT | matlab::data::Struct | Array with named fields that can contain data of varying types and sizes |
| VALUE_OBJECT | matlab::data::Object | MATLAB value object |
| HANDLE_OBJECT_REF | matlab::data::Object | Reference to an existing handle object in MATLAB |
| ENUM | matlab::data::Enumeration | Array of enumeration values |

| MATLAB ArrayType Value | C++ Type | Description |
|---|---|---|
| SPARSE_LOGICAL | bool | Sparse array of logical |
| SPARSE_DOUBLE | double | Sparse array of double |
| SPARSE_COMPLEX_DOUBLE | std::complex<double> | Sparse array of complex double |
| MATLAB_STRING | matlab::data::MATLABString | MATLAB string |

## Examples

### Test Array for COMPLEX_DOUBLE Type

Suppose that you have an array declared as follows.

```
matlab::data::TypedArray<double> const argArray
```

After you set values for argArray, call the sqrt function.

```
matlab::data::Array const tresults = matlabPtr->feval(u"sqrt", argArray);
```

These statements test the result for type COMPLEX_DOUBLE and then set the array type.

```
matlab::data::TypedArray<std::complex<double>> results = factory.createEmptyArray();
matlab::data::ArrayType type = tresults.getType();
if (type == matlab::data::ArrayType::COMPLEX_DOUBLE)
    results = (matlab::data::TypedArray<std::complex<double>>) tresults;
else
    std::cout << "ERROR: complex double array expected." << std::endl;
```

## See Also

matlab::data::apply_visitor | matlab::data::apply_visitor_ref

**Topics**

"Handling Inputs and Outputs"
"Data Access in Typed, Cell, and Structure Arrays"

**Introduced in R2017b**

# matlab::data::CellArray

C++ class to access MATLAB cell arrays

## Description

A `CellArray` is a `TypedArray` with `Array` as the element type. Use `CellArray` objects to access MATLAB cell arrays. To create a `CellArray`, call `createCellArray` in the `ArrayFactory` class.

`CellArray` is defined as:

```
using CellArray = TypedArray<Array>;
```

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `TypedArray.hpp` |

## See Also

`createCellArray`

**Introduced in R2017b**

# matlab::data::CharArray

C++ class to access MATLAB character arrays

## Description

Use `CharArray` objects to work with MATLAB character arrays. To create a `CharArray`, call `createCharArray` in the `ArrayFactory` class.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Base class: | `TypedArray<char16_t>` |
| Include: | `CharArray.hpp` |

## Constructors

- "Copy Constructors" on page 1-19
- "Copy Assignment Operators" on page 1-20
- "Move Constructors" on page 1-20
- "Move Assignment Operators" on page 1-21

### Copy Constructors

`CharArray(const CharArray& rhs)`

`CharArray(const Array& rhs)`

#### Description

Creates a shared data copy of a `CharArray` object.

#### Parameters

| | |
|---|---|
| `const CharArray& rhs` | Value to copy. |
| `const Array& rhs` | Value specified as `ArrayType::CHAR` object. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::CHAR`. |

#### Examples

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    CharArray A = factory.createCharArray("This is a char array");
    CharArray B(A);
```

```
    return 0;
}
```

#### Related Topics

createCharArray

### Copy Assignment Operators

`CharArray& operator=(const CharArray& rhs)`

`CharArray& operator=(const Array& rhs)`

#### Description

Assigns a shared data copy to a `CharArray` object.

#### Parameters

| | |
|---|---|
| `const CharArray& rhs` | Value to copy. |
| `const Array& rhs` | Value specified as `ArrayType::CHAR` object. |

#### Returns

| | |
|---|---|
| `CharArray&` | Updated instance. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::CHAR`. |

#### Examples

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    CharArray A = factory.createCharArray("This is a char array");
    CharArray C = factory.createCharArray("");

    // Arrays A and C refer to the same data.
    C = A;

    return 0;
}
```

### Move Constructors

`CharArray(CharArray&& rhs)`

`CharArray(Array&& rhs)`

#### Description

Moves contents of a `CharArray` object to a new instance.

**Parameters**

| | |
|---|---|
| `CharArray&& rhs` | Value to move. |
| `Array&& rhs` | Value specified as `ArrayType::CHAR` object. |

**Throws**

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::CHAR`. |

**Examples**

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    CharArray A = factory.createCharArray("This is a char array");

    // Move constructor - Creates B, copies data from A. A not valid.
    CharArray B(std::move(A));

    return 0;
}
```

### Move Assignment Operators

`CharArray& operator=(CharArray&& rhs)`

`CharArray& operator=(Array&& rhs)`

**Description**

Assigns the input to this `CharArray` object.

**Parameters**

| | |
|---|---|
| `CharArray&& rhs` | Value to move. |
| `Array&& rhs` | Value specified as `ArrayType::CHAR` object. |

**Returns**

| | |
|---|---|
| `CharArray&` | Updated instance. |

**Throws**

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::CHAR`. |

**Examples**

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
    CharArray A = factory.createCharArray("This is a char array");
```

```
    // Move assignment - Data from A moved to C. A no longer valid.
    CharArray C = factory.createCharArray("");
    C = std::move(A);

    return 0;
}
```

## Member Functions

- “toUTF16” on page 1-22
- “toAscii” on page 1-22

### toUTF16

```
String toUTF16() const
```

#### Returns

| | |
|---|---|
| `matlab::data::String` | Contents of `CharArray` as `matlab::data::String`. |

#### Throws

None

### toAscii

```
std::string toAscii() const
```

#### Returns

| | |
|---|---|
| `std::string` | Contents of `CharArray` as ASCII string. |

#### Throws

| | |
|---|---|
| `matlab::data::NonAsciiCharInRequestedAsciiOutputException` | Data contains non-ASCII characters. |

#### Examples

```
#include "MatlabDataArray.hpp"

int main()
{
    using namespace matlab::data;
    ArrayFactory f;
    auto arr = f.createCharArray("helloworld");
    std::string s = arr.toAscii();

    return 0;
}
```

#### Related Topics

“Evaluate Mathematical Function in MATLAB”

## See Also

"createCharArray" on page 1-6 | `TypedArray` | `matlab::data::String`

**Introduced in R2017b**

# matlab::data::Reference<CharArray>

C++ class to get reference to `CharArray`

## Description

The `CharArrayExt` class extends the APIs available to a reference to a `CharArray`.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Base class: | `Reference<Array>` |
| Include: | `TypedArrayRef.hpp` |

## Member Functions

- "toUTF16" on page 1-24
- "toAscii" on page 1-24

### toUTF16

```
String toUTF16() const
```

#### Returns

| | |
|---|---|
| `matlab::data::String` | Contents of reference to `CharArray` as `matlab::data::String` string. |

#### Throws

None

### toAscii

```
std::string toAscii() const
```

#### Returns

| | |
|---|---|
| `std::string` | Contents of reference to `CharArray` as ASCII string. |

#### Throws

| | |
|---|---|
| `matlab::data::NonAsciiCharInRequestedAsciiOutputException` | Data contains non-ASCII characters. |

## See Also

`CharArray` | `Reference<TypedArray<T>>`

**Introduced in R2017b**

# matlab::data::EnumArray

C++ class to access MATLAB enumeration arrays

## Description

Use `EnumArray` objects to access enumeration arrays. To create an `EnumArray`, call `createEnumArray` in the `ArrayFactory` class.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Base class: | `TypedArray<Enumeration>` |
| Include: | `EnumArray.hpp` |

## Constructors


- "Copy Constructors" on page 1-25
- "Copy Assignment Operators" on page 1-25
- "Move Constructors" on page 1-26
- "Move Assignment Operators" on page 1-26


### Copy Constructors

```
EnumArray(const EnumArray& rhs)
```

```
EnumArray(const Array& rhs)
```

#### Description

Creates a shared data copy of an `EnumArray` object.

#### Parameters

| | |
|---|---|
| `const EnumArray& rhs` | Value to copy. |
| `const Array& rhs` | Value specified as `EnumArray` object. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::ENUM`. |

### Copy Assignment Operators

```
EnumArray& operator=(const EnumArray& rhs)
```

```
EnumArray& operator=(const Array& rhs)
```

#### Description

Assigns a shared data copy to an `EnumArray` object.

**Parameters**

| | |
|---|---|
| `const EnumArray& rhs` | Value to copy. |
| `const Array& rhs` | Value specified as `ArrayType::ENUM` object. |

**Returns**

| | |
|---|---|
| `EnumArray&` | Updated instance. |

**Throws**

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::ENUM`. |

### Move Constructors

`EnumArray(EnumArray&& rhs)`

`EnumArray(Array&& rhs)`

**Description**

Moves contents of an `EnumArray` object to a new instance.

**Parameters**

| | |
|---|---|
| `EnumArray&& rhs` | Value to move. |
| `Array&& rhs` | Value specified as `ArrayType::ENUM` object. |

**Throws**

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::ENUM`. |

### Move Assignment Operators

`EnumArray& operator=(EnumArray&& rhs)`

`EnumArray& operator=(Array&& rhs)`

**Description**

Assigns the input to this `EnumArray` object.

**Parameters**

| | |
|---|---|
| `EnumArray&& rhs` | Value to move. |
| `Array&& rhs` | Value specified as `ArrayType::ENUM` object. |

**Returns**

| | |
|---|---|
| `EnumArray&` | Updated instance. |

**Throws**

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::ENUM`. |

## Member Functions

### `getClassName`

```
std::string getClassName() const
```

**Description**

Return class name for this `EnumArray`.

**Returns**

| | |
|---|---|
| `std::string` | Class name. |

**Throws**

None

## See Also

“createEnumArray” on page 1-8 | `Enumeration` | `TypedArray`

**Introduced in R2017b**

# matlab::data::Reference<EnumArray>

C++ class to get reference to `EnumArray`

## Description

The `EnumArrayExt` class extends the APIs available to a reference to an `EnumArray`.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Base class: | `Reference<Array>` |
| Include: | `TypedArrayRef.hpp` |

## Member Functions

### getClassName

```
std::string getClassName() const
```

#### Description

Return class name for this reference to an `EnumArray` object.

#### Returns

| | |
|---|---|
| `std::string` | Class name. |

#### Throws

None

## See Also

`EnumArray` | `Reference<TypedArray<T>>`

**Introduced in R2017b**

# matlab::data::Enumeration

Element type for MATLAB enumeration arrays

## Description

`Enumeration` is the element type for an `EnumArray` object.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `Enumeration.hpp` |

## See Also

`EnumArray`

### Topics

"MATLAB Data API Types"

**Introduced in R2017b**

# matlab::Exception

C++ base class for exceptions

## Description

All MATLAB C++ exceptions can be caught as `matlab::Exception`.

### Class Details

| | |
|---|---|
| Namespace: | `matlab` |
| Include: | `Exception.hpp` |

## See Also

**Topics**
"MATLAB Data API Exceptions"

**Introduced in R2017b**

# matlab::data::ForwardIterator<T>

Templated C++ class to provide forward iterator support for `StructArray` field names

## Description

Use `ForwardIterator` objects to access a range of field name elements in a `StructArray`.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `ForwardIterator.hpp` |

### Template Parameters

| | |
|---|---|
| `T` | `matlab::data::MATLABFieldIdentifier` |

## Constructors


- "Copy Constructors" on page 1-31
- "Copy Assignment Operators" on page 1-31


### Copy Constructors

```
ForwardIterator(const ForwardIterator<T>& rhs)
```

#### Description

Creates a shared data copy of a `ForwardIterator<T>` object.

#### Parameters

| | |
|---|---|
| `const ForwardIterator<T>& rhs` | Object to copy. |

#### Returns

| | |
|---|---|
| `ForwardIterator` | New instance. |

#### Throws

None

### Copy Assignment Operators

```
ForwardIterator<T>& operator=(const ForwardIterator<T>& rhs)
```

#### Description

Assigns a shared data copy to a `ForwardIterator<T>` object.

**Parameters**

| | |
|---|---|
| const ForwardIterator<T>& rhs | Object to assign. |

**Returns**

| | |
|---|---|
| ForwardIterator<T> | Updated instance. |

**Throws**

None

## Other Operators


- "operator++" on page 1-32
- "operator--" on page 1-32
- "operator=" on page 1-33
- "operator!=" on page 1-33
- "operator*" on page 1-33
- "operator->" on page 1-33
- "operator[]" on page 1-34


### operator++

```
ForwardIterator<T>& operator++()
```

**Description**

Pre-increment operator.

**Returns**

| | |
|---|---|
| ForwardIterator<T>& | Reference to updated value. |

**Throws**

None

### operator--

```
ForwardIterator<T> operator--(int)
```

**Description**

Post-increment operator.

**Returns**

| | |
|---|---|
| ForwardIterator<T> | New object. |

**Throws**

None

### operator=

```
bool operator==(const ForwardIterator<T>& rhs) const
```

#### Parameters

| | |
|---|---|
| `const ForwardIterator<T>& rhs` | Iterator to compare. |

#### Returns

| | |
|---|---|
| `bool` | Returns `true` if the iterators point to the same element. Otherwise, returns `false`. |

#### Throws

None

### operator!=

```
bool operator!=(const ForwardIterator<T>& rhs) const
```

#### Parameters

| | |
|---|---|
| `const ForwardIterator<T>& rhs` | Iterator to compare. |

#### Returns

| | |
|---|---|
| `bool` | Returns `true` if this iterator points to a different element. Otherwise, returns `false`. |

#### Throws

None

### operator*

```
reference operator*() const
```

#### Returns

| | |
|---|---|
| `reference` | Shared copy of element that iterator points to, specified as:<br>• `T&` for arithmetic types.<br>• `Reference<T>` for non-arithmetic types. |

#### Throws

None

### operator->

```
pointer operator->()
```

**Returns**

| | |
|---|---|
| `pointer` | Pointer to element pointed to by this iterator, specified as:<br>• `T*` for arithmetic types.<br>• `Reference<T>*` for non-arithmetic types. |

**Throws**

None

### `operator[]`

```
reference operator[](const size_t& rhs) const
```

**Description**

Get a reference using a linear index.

**Returns**

| | |
|---|---|
| `reference` | Element pointed to by this iterator, specified as `typename iterator::reference`. |

**Throws**

None

## See Also

`MATLABFieldIdentifier` | `StructArray`

**Introduced in R2017b**

# matlab::data::MATLABFieldIdentifier

C++ class used to identify field names in MATLAB struct array

## Description

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `MATLABFieldIdentifier.hpp` |

## Constructors


- "Default Constructor" on page 1-35
- "Constructor" on page 1-35
- "Destructor" on page 1-35
- "Copy Constructors" on page 1-36
- "Copy Assignment Operators" on page 1-36
- "Move Constructors" on page 1-36
- "Move Assignment Operators" on page 1-37


### Default Constructor

```
MATLABFieldIdentifier()
```

#### Description

Construct an empty `MATLABFieldIdentifier`.

#### Throws

None

### Constructor

```
MATLABFieldIdentifier(std::string str)
```

#### Description

Construct a `MATLABFieldIdentifier` from `std::string`.

#### Parameters

| | |
|---|---|
| `std::string str` | String that contains the field name. |

### Destructor

```
~MATLABFieldIdentifier()
```

#### Description

Destroy a `MATLABFieldIdentifier`.

#### Throws

None

### Copy Constructors

`MATLABFieldIdentifier(const MATLABFieldIdentifier& rhs)`

#### Description

Creates a shared data copy of a `MATLABFieldIdentifier` object.

#### Parameters

| | |
|---|---|
| `const MATLABFieldIdentifier& rhs` | Value to copy. |

#### Throws

None

### Copy Assignment Operators

`MATLABFieldIdentifier& operator=(MATLABFieldIdentifier const& rhs)`

#### Description

Assigns a shared data copy to a `MATLABFieldIdentifier` object.

#### Parameters

| | |
|---|---|
| `MATLABFieldIdentifier const& rhs` | Value to move. |

#### Returns

| | |
|---|---|
| `MATLABFieldIdentifier&` | Updated instance. |

#### Throws

None

### Move Constructors

`MATLABFieldIdentifier(MATLABFieldIdentifier&& rhs)`

#### Description

Moves contents a `MATLABFieldIdentifier` object to a new instance.

#### Parameters

| | |
|---|---|
| `MATLABFieldIdentifier&& rhs` | Value to move. |

#### Throws

None

#### Move Assignment Operators

`MATLABFieldIdentifier& operator=(MATLABFieldIdentifier&& rhs)`

##### Parameters

| | |
|---|---|
| `MATLABFieldIdentifier&& rhs` | Value to move. |

##### Returns

| | |
|---|---|
| `MATLABFieldIdentifier&` | Updated instance. |

##### Throws

None

## Destructor

`~MATLABFieldIdentifier()`

##### Description

Destroy a `MATLABFieldIdentifier`.

## Other Operators

### `operator std::string`

`operator std::string() const`

##### Returns

| | |
|---|---|
| `std::string` | Representation of the `MATLABFieldIdentifier` object. |

##### Throws

None

## Free Functions

### `operator==`

`bool operator==(const MATLABFieldIdentifier& rhs) const`

##### Description

Check if two `MATLABFieldIdentifier` objects are identical.

##### Parameters

| | |
|---|---|
| `const MATLABFieldIdentifier& rhs` | Value to be compared. |

#### Returns

| | |
|---|---|
| `bool` | Returns true if the objects are identical. Otherwise, returns false. |

#### Throws

None

## Examples

### Get Contents of Structure

Access the data in MATLAB structures that are passed to C++ MEX functions or C++ Engine programs using the structure field name.

Here is a structure passed to a MEX function. The `Date` field contains the date when the structure is created, as returned by the `date` function. The `Data` field contains a numeric value.

```
s = struct('Date',date,'Data',100);
```

In a MEX function, store the input as a `StructArray`. Use the `getFieldNames` member function to get a range of `MATLABFieldIdentifier` elements representing the structure field names. Use the second element to get the numeric data from the `Data` field. Store numeric data in a `TypedArray` with elements of type `double`.

```
matlab::data::StructArray inStruct(inputs[0]);
matlab::data::Range<matlab::data::ForwardIterator, matlab::data::MATLABFieldIdentifier const>
    fields = inStruct.getFieldNames();
const matlab::data::TypedArray<double> data = inStruct[0][fields.begin()[1]];
double cppData = data[0];
```

## See Also

`ForwardIterator` | `StructArray` | `TypedArray<T>`

#### Topics

“Data Access in Typed, Cell, and Structure Arrays”
“Create Structure Arrays from C++”

#### Introduced in R2017b

# matlab::data::MATLABString

Element type for MATLAB string arrays

## Description

Use `MATLABString` to represent MATLAB string arrays in C++. To be able to represent missing string array elements, `MATLABString` is defined as:

```
using MATLABString = optional<String>;
```

For more information on string arrays in MATLAB, see “Create String Arrays”.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `String.hpp` |

## Examples

### Pass String Array from MATLAB to MEX function

Create a string array in MATLAB and pass it to a C++ MEX function:

```
str(1) = "";
str(2) = "Gemini";
str(3) = string(missing)
result = myMexFcn(str);
```

In the MEX function, assign the input to an array of type `matlab::data::MATLABString`.

```
matlab::data::TypedArray<matlab::data::MATLABString> stringArray = inputs[0];
```

### Pass String Array from MEX function to MATLAB

Create a string array in the MEX function and pass this array to MATLAB as output. The array defines text elements, an empty string, and a missing string element.

```
matlab::data::ArrayFactory factory;
outputs[0] = factory.createArray<MATLABString>({ 1,3 },
    { matlab::data::MATLABString(u""),
       matlab::data::MATLABString(u"Gemini"),
       matlab::data::MATLABString() });
```

The result returned to MATLAB is a string array.

```
result =

  1×3 string array

    ""    "Gemini"    <missing>
```

## See Also

`matlab::data::String` | `matlab::data::optional<T>`

**Topics**
"C++ MEX Applications"
"MATLAB Engine API for C++"

**Introduced in R2017b**

# matlab::data::Reference<MATLABString>

C++ class to get reference to element of `StringArray`

## Description

A `Reference<MATLABString>` object is created when using `operator[]` into a `StringArray` or dereferencing a String array iterator.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `MATLABStringReferenceExt.hpp` |

## Cast

### `String()`

```
operator String() const
```

#### Returns

| | |
|---|---|
| `matlab::data::String` | Element of a `MATLABString` array converted to `String`. |

#### Throws

| | |
|---|---|
| `matlab::data::NotEnoughIndicesProvidedException` | Not enough indices provided. |
| `matlab::data::InvalidArrayIndexException` | Index provided is not valid for this Array or one of the indices is out of range. |
| `matlab::data::TooManyIndicesProvidedException` | Too many indices provided. |
| `std::runtime_error` | Array element does not have a value. |

## Member Functions


- "bool" on page 1-41
- "has_value" on page 1-42


### `bool`

```
operator bool() const
```

#### Description

Check whether string contains a value.

#### Returns

| | |
|---|---|
| `operator` | True, if string contains a value. |

#### Throws

| | |
|---|---|
| `matlab::data::NotEnoughIndicesProvidedException` | Not enough indices provided. |
| `matlab::data::InvalidArrayIndexException` | Index provided is not valid for this Array or one of the indices is out of range. |
| `matlab::data::TooManyIndicesProvidedException` | Too many indices provided. |

### has_value

```
bool has_value() const
```

#### Description

Check whether string contains a value.

#### Returns

| | |
|---|---|
| `bool` | True, if string contains a value. |

#### Throws

| | |
|---|---|
| `matlab::data::NotEnoughIndicesProvidedException` | Not enough indices provided. |
| `matlab::data::InvalidArrayIndexException` | Index provided is not valid for this Array or one of the indices is out of range. |
| `matlab::data::TooManyIndicesProvidedException` | Too many indices provided. |

## See Also

**Introduced in R2017b**

# matlab::data::Array

C++ base class for all array types

## Description

Use `Array` objects to represent single and multi-dimensional arrays. The `Array` class provides methods to get generic information about all arrays, such as dimensions and type. The class has methods to create both deep (cloned) copies and shared data copies and supports copy-on-write semantics.

To construct `Array` objects, use `ArrayFactory` methods.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `MDArray.hpp` |

## Constructors


- "Default Constructor" on page 1-43
- "Copy Constructors" on page 1-43
- "Copy Assignment Operators" on page 1-44
- "Move Constructors" on page 1-44
- "Move Assignment Operators" on page 1-44


### Default Constructor

```
Array()
```

#### Throws

None

### Copy Constructors

```
Array(const Array& rhs)
```

#### Description

Creates a shared data copy of an `Array` object.

#### Parameters

| | |
|---|---|
| `const Array& rhs` | Value to copy. |

#### Throws

None

### Copy Assignment Operators

`Array& operator=(const Array& rhs)`

**Description**

Assigns a shared data copy to an `Array` object.

**Parameters**

| | |
|---|---|
| `const Array& rhs` | Value to copy. |

**Returns**

| | |
|---|---|
| `Array&` | Updated instance. |

**Throws**

None

### Move Constructors

`Array(Array&& rhs)`

**Description**

Moves contents of an `Array` object to a new instance.

**Parameters**

| | |
|---|---|
| `Array&& rhs` | Value to move. |

**Throws**

None

### Move Assignment Operators

`Array& operator=(Array&& rhs)`

**Description**

Assigns the input to this `Array` object.

**Parameters**

| | |
|---|---|
| `Array&& rhs` | Value to move. |

**Returns**

| | |
|---|---|
| `Array&` | Updated instance. |

**Throws**

None

## Destructor

`virtual ~Array()`

## Indexing Operators

### `operator[]`

`ArrayElementRef<false> operator[](size_t idx)`

`ArrayElementRef<true> operator[](size_t idx) const`

#### Description

Enables `[]` indexing on `const` and non-`const` arrays. Indexing is 0-based.

#### Parameters

| | |
|---|---|
| `size_t idx` | First array index |

#### Returns

| | |
|---|---|
| `ArrayElementRef<false>` | Temporary object containing the index specified. The return value allows the element of the array to be modified or retrieved. |
| `ArrayElementRef<true>` | Temporary object containing the index specified. The return value allows the element of the array to be retrieved, but not modified. |

#### Throws

None

## Member Functions

- "getType" on page 1-45
- "getMemoryLayout" on page 1-46
- "getDimensions" on page 1-46
- "getNumberOfElements" on page 1-46
- "isEmpty" on page 1-46

### `getType`

`ArrayType getType() const`

#### Returns

| | |
|---|---|
| `ArrayType` | Array type. |

#### Throws

None

### getMemoryLayout

```
MemoryLayout getMemoryLayout() const
```

**Returns**

| | |
|---|---|
| `MemoryLayout` | Memory layout for array, specified as `MemoryLayout::COLUMN_MAJOR` or `MemoryLayout::ROW_MAJOR`. |

**Throws**

| | |
|---|---|
| `matlab::data::InvalidMemoryLayout` | Invalid memory layout. |

### getDimensions

```
ArrayDimensions getDimensions() const
```

**Returns**

| | |
|---|---|
| `ArrayDimensions` | Vector of each dimension in array. |

**Throws**

None

### getNumberOfElements

```
size_t getNumberOfElements() const
```

**Returns**

| | |
|---|---|
| `size_t` | The number of elements in array. |

**Throws**

None

### isEmpty

```
bool isEmpty() const
```

**Returns**

| | |
|---|---|
| `bool` | True if array is empty. False if array is not empty. |

**Throws**

None

## Free Functions

- “getReadOnlyElements” on page 1-47
- “getWritableElements” on page 1-47

### getReadOnlyElements

```
template <typename T>
Range<TypedIterator, T const> getReadOnlyElements(const Array& arr)
```

**Description**

Get a range containing the elements of the Array. Iterators contained in the range are const.

**Parameters**

| | |
|---|---|
| const Array& arr | Array |

**Returns**

| | |
|---|---|
| Range<TypedIterator, T const> | Range containing begin and end iterators for input Array. |

**Throws**

| | |
|---|---|
| matlab::data::InvalidArrayTypeException | Array does not contain type T. |

### getWritableElements

```
template <typename T>
Range<TypedIterator, T> getWritableElements(Array& arr)
```

**Description**

Get a range containing the elements of the Array. Iterators contained in the range are non-const.

**Parameters**

| | |
|---|---|
| Array& arr | Array |

**Returns**

| | |
|---|---|
| Range<TypedIterator, T> | Range containing begin and end iterators for input Array. |

**Throws**

| | |
|---|---|
| matlab::data::InvalidArrayTypeException | Array does not contain type T. |

## See Also

ArrayFactory

**Introduced in R2017b**

# matlab::data::Object

Element type for MATLAB object arrays

## Description

`Object` is the element type for an `ObjectArray`.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `Object.hpp` |

## See Also

`ObjectArray`

**Introduced in R2017b**

# matlab::data::ObjectArray

C++ class to access MATLAB object arrays

## Description

Use `ObjectArray` objects to access MATLAB object arrays. To create an `ObjectArray`, call `createArray` in the `ArrayFactory` class using this syntax:

```
template <typename ItType, typename T>
TypedArray<T> createArray(ArrayDimensions dims,
    ItType begin,
    ItType end)
```

To create a scalar object, call `createScalar` using this syntax:

```
ObjectArray createScalar(const Object& val);
```

`ObjectArray` is defined as:

```
using ObjectArray = TypedArray<Object>;
```

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `ObjectArray.hpp` |

You cannot combine elements of an `ObjectArray` into a heterogeneous array.

If the class defining the `Object` overrides `subsref` or `subsasgn`, then you cannot access the elements of the `ObjectArray`.

## Examples

### Create `ObjectArray`

Create an `ObjectArray` from `myObject` class objects. The iterators are pointers to the beginning and the end of the array.

```
class myObject {
  public:
    const std::vector<matlab::data::Object>& getObjs() const {
       return fObjs;
    }
  private:
    std::vector<matlab::data::Object> fObjs;
};

const myObject& a1;
const myObject& a2;
matlab::data::ArrayFactory factory;
const auto& objs = a1.getObjs();
matlab::data::ObjectArray arr1 = factory.createArray({1,2}, objs.begin(), objs.end());
```

### Iterate Through `ObjectArray`

Iterate using a range-based for loop through an `ObjectArray` and retrieve the objects in the array.

```
std::vector<matlab::data::Object> fObjs;

// Use a range-based for loop to iterate over the objects.
for (const auto& o : objs) {
    fObjs.push_back(o);
}
```

## Objects in MEX and Engine Applications

C++ MEX and C++ Engine applications can get and set property values on MATLAB objects. For information on how to access MATLAB objects in these applications, see these topics:

- "MATLAB Objects in MEX Functions" - for C++ MEX applications
- "Get MATLAB Objects and Access Properties" - for C++ Engine applications

## See Also

`Object` | `createArray` | `createScalar`

**Introduced in R2017b**

# matlab::data::optional<T>

Templated C++ class representing optional values

## Description

Use `optional` objects to represent values that might or might not exist.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `Optional.hpp` |

### Template Parameters

| | |
|---|---|
| `T` | Array type, specified as `matlab::data::String`. |

## Constructors


- "Default Constructors" on page 1-51
- "Copy Constructors" on page 1-51
- "Copy Assignment Operators" on page 1-51
- "Move Constructors" on page 1-52
- "Move Assignment Operators" on page 1-52


### Default Constructors

```
optional()
```

### Copy Constructors

```
optional(const optional& other)
```

#### Description

Creates a shared data copy.

#### Parameters

| | |
|---|---|
| `const optional& other` | Value to copy. |

#### Throws

None

### Copy Assignment Operators

```
optional<T>& operator=(const optional<T>& other)
```

#### Description

Assigns a shared data copy.

**Parameters**

| | |
|---|---|
| `const optional<T>& other` | Value to copy. |

**Returns**

| | |
|---|---|
| `optional<T>&` | Updated instance. |

**Throws**

None

### Move Constructors

`optional(optional&& other)`

`optional(T&& value)`

**Description**

Moves contents of an `optional` object to a new instance.

**Parameters**

| | |
|---|---|
| `optional&& other` | Value to move. |
| `T&& value` | Value of type `T` to move. |

**Throws**

None

### Move Assignment Operators

`optional<T>& operator=(optional<T>&& other)`

`optional<T>& operator=(T&& value)`

**Description**

Assigns the input to this instance.

**Parameters**

| | |
|---|---|
| `optional<T>&& other` | Value to move. |
| `T&& value` | |

**Returns**

| | |
|---|---|
| `optional<T>&` | Updated instance. |

**Throws**

None

## Other Operators



### operator=

```
optional<T>& operator=(nullopt_t)
```

```
optional<T>& operator=(const optional<T>& other)
```

```
optional<T>& operator=(optional<T>&& other)
```

```
optional<T>& operator=(T&& value)
```

```
optional<T>& operator=(const T& value)
```

**Description**

Assignment operators.

**Returns**

| | |
|---|---|
| `optional<T>&` | Updated instance. |

**Throws**

None

### operator->

```
const T* operator->() const
```

```
T* operator->()
```

**Returns**

| | |
|---|---|
| `const T*`<br>`T*` | Pointer to the element. |

**Throws**

| | |
|---|---|
| `std::runtime_error` | `optional` object does not contain a value. |

### operator*

```
const T& operator*() const
```

```
T& operator*()
```

**Returns**

| | |
|---|---|
| `const T&`<br>`T&` | Reference to the element. |

**Throws**

| | |
|---|---|
| `std::runtime_error` | `optional` object does not contain a value. |

### operator T

`operator T() const`

**Description**

Cast `optional<T>` value to `T`.

**Returns**

| | |
|---|---|
| `operator` | Value contained in `optional<T>`, if it exists. |

**Throws**

| | |
|---|---|
| `std::runtime_error` | There is no value. |

## Member Functions


- "bool" on page 1-54
- "has_value" on page 1-54
- "swap" on page 1-55
- "reset" on page 1-55


### bool

`explicit operator bool() const`

**Description**

Check whether object contains a value.

**Returns**

| | |
|---|---|
| `operator` | True, if object contains a value. |

**Throws**

None

### has_value

`bool has_value() const`

**Description**

Check whether object contains a value.

#### Returns

| | |
|---|---|
| `bool` | True, if object contains a value. |

#### Throws

None

### `swap`

`void swap(optional &other)`

#### Description

Swap value of this optional instance with value contained in the parameter.

#### Parameters

| | |
|---|---|
| `optional &other` | Value to swap. |

#### Throws

None

### `reset`

`void reset()`

#### Description

Reset optional value to missing

#### Throws

None

## See Also

#### Introduced in R2017b

# matlab::data::Range<ItType,ElemType>

Templated C++ class to provide range-based operation support

## Description

`Range` objects wrap `begin` and `end` functions to enable range-based operations.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `Range.hpp` |

### Template Parameters

| | |
|---|---|
| `IteratorType` | Iterator type |
| `ElementType` | Element type |

## Constructors


- "Constructor" on page 1-56
- "Move Constructors" on page 1-56
- "Move Assignment Operators" on page 1-57


### Constructor

```
Range(IteratorType<ElementType> begin, IteratorType<ElementType> end)
```

#### Description

Creates a `Range` object.

#### Parameters

| | |
|---|---|
| `IteratorType<ElementType> begin` | First and last elements of range. |
| `IteratorType<ElementType> end` | |

#### Returns

| | |
|---|---|
| `Range` | New instance. |

#### Throws

None

### Move Constructors

```
Range(Range&& rhs)
```

#### Description

Moves contents of a Range object to a new instance.

#### Parameters

| | |
|---|---|
| `Range&& rhs` | Range to move. |

#### Returns

| | |
|---|---|
| `Range` | New instance. |

#### Throws

None

### Move Assignment Operators

`Range& operator=(Range&& rhs)`

#### Description

Assigns the input to this Range object.

#### Parameters

| | |
|---|---|
| `Range&& rhs` | Range to move. |

#### Returns

| | |
|---|---|
| `Range&` | Updated instance. |

#### Throws

None

## begin

`IteratorType<ElementType>& begin()`

### Returns

| | |
|---|---|
| `IteratorType<ElementType>&` | First element in range. |

#### Throws

None

## end

`IteratorType<ElementType>& end()`

### Returns

| | |
|---|---|
| `IteratorType<ElementType>&` | End of range. |

#### Throws

None

## See Also

#### Introduced in R2017b

# matlab::data::Reference<T>

Templated C++ class to get references to `Array` elements

## Description

A `Reference` object is a reference to an element of an `Array` without making a copy. A `Reference` is:

- Not a shared copy
- Valid as long as the array that contains the reference is valid
- Not thread-safe

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `Reference.hpp` |

### Template Parameters

| | |
|---|---|
| `T` | Type of element referred to, specified as:<br>• `Array`<br>• `Struct`<br>• `Enumeration`<br>• `MATLABString`<br>• All `std::complex` types |

## Constructors


- “Copy Constructor” on page 1-59
- “Copy Assignment Operators” on page 1-59
- “Move Assignment Operators” on page 1-60
- “Move Constructors” on page 1-60


### Copy Constructor

```
Reference(const Reference<T>& rhs)
```

#### Parameters

| | |
|---|---|
| `const Reference<T>& rhs` | Value to copy. |

### Copy Assignment Operators

```
Reference<T>& operator=(const Reference<T>& rhs)
```

**Parameters**

| | |
|---|---|
| const Reference<T>& rhs | Value to copy. |

**Returns**

| | |
|---|---|
| Reference<T>& | Updated instance. |

#### Move Assignment Operators

```
Reference<T>& operator=(Reference<T>&& rhs)
```

**Parameters**

| | |
|---|---|
| Reference<T>&& rhs | Value to move. |

**Returns**

| | |
|---|---|
| Reference<T>& | Updated instance. |

**Throws**

None

#### Move Constructors

```
Reference(Reference<T>&& rhs)
```

**Description**

Moves contents of a Reference object to a new instance.

**Parameters**

| | |
|---|---|
| Reference<T>&& rhs | Value to move. |

**Throws**

None

## Other Operators

- “operator=” on page 1-60
- “operator<<” on page 1-61
- “operator T()” on page 1-61
- “operator std::string()” on page 1-61

### operator=

```
Reference<T>& operator=(T rhs)
```

```
Reference<T>& operator=(std::string rhs)
```

```
Reference<T>& operator=(String rhs)
```

#### Parameters

| | |
|---|---|
| `T rhs` | Value to assign. The indexed array must be non-`const`. |
| `std::string rhs` | String to assign. The array must be non-`const` and allow strings to be assigned. |
| `String rhs` | String to assign to `StringArray`. The indexed array must be non-`const`. |

#### Returns

| | |
|---|---|
| `Reference<T>&` | Updated instance. |

#### Throws

None

### operator<<

```
std::ostream& operator <<(std::ostream& os, Reference<T> const& rhs)
```

#### Parameters

| |
|---|
| `std::ostream& os` |
| `Reference<T> const& rhs` |

#### Returns

| |
|---|
| `std::ostream&` |

### operator T()

```
operator T() const
```

#### Description

Cast to element from the array.

#### Returns

| | |
|---|---|
| `T` | Shared copy of element from array. |

#### Throws

None

### operator std::string()

```
operator std::string() const
```

#### Description

Casts array to `std::string`, making a copy of the `std::string`. This operator is valid only for types that can be cast to a `std::string`.

#### Returns

| | |
|---|---|
| `std::string` | String. |

#### Throws

| | |
|---|---|
| `matlab::data::NonAsciiCharInInputDataException` | Input is `std::string` and contains non-ASCII characters. |
| `std::runtime_error` | `MATLABString` is missing. |

## Free Functions

#### operator==

```
inline bool operator ==(Reference<MATLABString> const& lhs, std::string
const& rhs)
```

```
inline bool operator ==(std::string const& lhs, Reference<MATLABString>
const& rhs)
```

```
inline bool operator ==(Reference<MATLABString> const& lhs, String const&
rhs)
```

```
inline bool operator ==(String const& lhs, Reference<MATLABString> const&
rhs)
```

```
inline bool operator ==(Reference<MATLABString> const& lhs, MATLABString
const& rhs)
```

```
inline bool operator ==(MATLABString const& lhs, Reference<MATLABString>
const& rhs)
```

```
inline bool operator ==(Reference<MATLABString> const& lhs,
Reference<MATLABString> const& rhs)
```

```
template<typename T> bool operator ==(Reference<T> const& lhs, T const& rhs)
```

```
template<typename T> bool operator ==(T const& lhs, Reference<T> const& rhs)
```

```
template<typename T> bool operator ==(Reference<T> const& lhs, Reference<T>
const& rhs)
```

#### Parameters

| | | |
|---|---|---|
| `Reference<MATLABString> const& lhs` | `std::string const& rhs` | Values to compare. |
| `std::string const& lhs` | `Reference<MATLABString> const& rhs` | |
| `Reference<MATLABString> const& lhs` | `String const& rhs` | |
| `String const& lhs` | `Reference<MATLABString> const& rhs` | |

| | |
|---|---|
| `Reference<MATLABString> const& lhs` | `MATLABString const& rhs` |
| `MATLABString const& lhs` | `Reference<MATLABString> const& rhs` |
| `Reference<MATLABString> const& lhs` | `Reference<MATLABString> const& rhs` |
| `Reference<T> const& lhs` | `T const& rhs` |
| `T const& lhs` | `Reference<T> const& rhs` |
| `Reference<T> const& lhs` | `Reference<T> const& rhs` |

**Returns**

| | |
|---|---|
| `bool` | Returns `true` if values are equal. |

**Throws**

| | |
|---|---|
| `std::runtime_error` | Cannot compare argument to `MATLABString`. |

## See Also

**Topics**

"Access C++ Data Array Container Elements"

**Introduced in R2017b**

# matlab::data::SparseArray<T>

Templated C++ class to access data in MATLAB sparse arrays

## Description

Use SparseArray objects to work with sparse MATLAB arrays. To create a SparseArray, call createSparseArray in the ArrayFactory class.

### Class Details

| | |
|---|---|
| Namespace: | matlab::data |
| Base class: | matlab::data::Array |
| Include: | SparseArray.hpp |

### Template Parameters

| | |
|---|---|
| T | Type of element referred to, specified as:<br>• bool<br>• double<br>• std::complex<double> |

## Constructors

- “Copy Constructors” on page 1-64
- “Copy Assignment Operators” on page 1-65
- “Move Constructors” on page 1-65
- “Move Assignment Operators” on page 1-66

### Copy Constructors

```
SparseArray(const SparseArray<T>& rhs)
```

```
SparseArray(const Array& rhs)
```

#### Description

Creates a shared data copy of a SparseArray object.

#### Parameters

| | |
|---|---|
| const SparseArray<T>& rhs | Value to copy. |
| const Array& rhs | Value specified as Array of ArrayType::SPARSE_LOGICAL, ArrayType::SPARSE_DOUBLE, or ArrayType::SPARSE_COMPLEX_DOUBLE. |

#### Throws

| | |
|---|---|
| matlab::data::InvalidArrayTypeException | Type of input Array is not sparse. |

### Copy Assignment Operators

```
SparseArray& operator=(const SparseArray<T>& rhs)
```

```
SparseArray& operator=(const Array& rhs)
```

#### Description

Assigns a shared data copy to a SparseArray object.

#### Parameters

| | |
|---|---|
| const SparseArray<T>& rhs | Value to copy. |
| const Array& rhs | Value specified as Array of type ArrayType::SPARSE_LOGICAL, ArrayType::SPARSE_DOUBLE, or ArrayType::SPARSE_COMPLEX_DOUBLE. |

#### Returns

| | |
|---|---|
| SparseArray& | Updated instance. |

#### Throws

| | |
|---|---|
| matlab::data::InvalidArrayTypeException | Type of input Array is not sparse. |

### Move Constructors

```
SparseArray(SparseArray&& rhs)
```

```
SparseArray(Array&& rhs)
```

#### Description

Moves contents of a SparseArray object to a new instance.

#### Parameters

| | |
|---|---|
| const SparseArray<T>& rhs | Value to move. |
| const Array& rhs | Value specified as Array of type ArrayType::SPARSE_LOGICAL, ArrayType::SPARSE_DOUBLE, or ArrayType::SPARSE_COMPLEX_DOUBLE. |

#### Throws

| | |
|---|---|
| matlab::data::InvalidArrayTypeException | Type of input Array is not sparse. |

### Move Assignment Operators

`SparseArray& operator=(SparseArray<T>&& rhs)`

`SparseArray& operator=(Array&& rhs)`

#### Description

Assigns the input to this `SparseArray` object.

#### Parameters

| | |
|---|---|
| `const SparseArray<T>& rhs` | Value to move. |
| `const Array& rhs` | Value specified as `Array` of type `ArrayType::SPARSE_LOGICAL`, `ArrayType::SPARSE_DOUBLE`, or `ArrayType::SPARSE_COMPLEX_DOUBLE`. |

#### Returns

| | |
|---|---|
| `SparseArray&` | Updated instance. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not sparse. |

## Iterators



### Begin Iterators

`iterator begin()`

`const_iterator begin() const`

`const_iterator cbegin() const`

#### Returns

| | |
|---|---|
| `iterator` | Iterator to beginning of array, specified as `TypedIterator<T>`. |
| `const_iterator` | Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`. |

#### Throws

None

### End Iterators

`iterator end()`

`const_iterator end() const`

```
const_iterator cend() const
```

**Returns**

| | |
|---|---|
| `iterator` | Iterator to end of array, specified as `TypedIterator<T>`. |
| `const_iterator` | Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`. |

**Throws**

None

## Member Functions

- “getNumberOfNonZeroElements” on page 1-67
- “getIndex” on page 1-67

### getNumberOfNonZeroElements

```
size_t getNumberOfNonZeroElements() const
```

**Description**

Returns the number of nonzero elements in the array.

**Returns**

| | |
|---|---|
| `size_t` | Number of nonzero elements in array. |

**Throws**

None

### getIndex

```
SparseIndex getIndex(const TypedIterator<T>& it)
```

```
SparseIndex getIndex(const TypedIterator<T const>& it)
```

**Description**

Returns the row-column coordinates of the nonzero entry that the iterator is pointing to.

**Parameters**

| | |
|---|---|
| `const TypedIterator<T>& it` | Iterator pointing to current entry in sparse matrix. |
| `const TypedIterator<T const>& it` | |

**Returns**

| | |
|---|---|
| `SparseIndex` | Row-column coordinates of nonzero entry that iterator points to. `SparseIndex` is defined as `std::pair<size_t, size_t>`. |

**Throws**

None

## See Also
`Array` | `createSparseArray`

**Introduced in R2017b**

# matlab::data::Reference<SparseArray<T>>

Templated C++ class to get reference to `SparseArray`

## Description

Use the `Reference<SparseArray>` class to get a reference to a `SparseArray` element of a container object, such as a MATLAB structure or cell array.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `SparseArrayRef.hpp` |

### Template Parameters

| | |
|---|---|
| `T` | Type of elements in `SparseArray`, specified as `bool`, `double`, or `std::complex<double>`. |

## Iterators

- "Begin Iterators" on page 1-69
- "End Iterators" on page 1-69

### Begin Iterators

```
iterator begin()
```

```
const_iterator begin() const
```

```
const_iterator cbegin() const
```

#### Returns

| | |
|---|---|
| `iterator` | Iterator to beginning of array, specified as `TypedIterator<T>`. |
| `const_iterator` | Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`. |

#### Throws

None

### End Iterators

```
iterator end()
```

```
const_iterator end() const
```

```
const_iterator cend() const
```

**Returns**

| | |
|---|---|
| `iterator` | Iterator to beginning of array, specified as `TypedIterator<T>`. |
| `const_iterator` | Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`. |

**Throws**

None

## Member Functions

### `getNumberOfNonZeroElements`

```
size_t getNumberOfNonZeroElements() const
```

**Description**

Returns the number of nonzero elements in the array. Since sparse arrays only store nonzero elements, this method returns the actual array size. It is different from array dimensions that specify the full array size.

**Returns**

| | |
|---|---|
| `size_t` | Number of nonzero elements in the array. |

**Throws**

None

## See Also

**Introduced in R2017b**

# matlab::data::String

Type representing strings as `std::basic_string<char16_t>`

## Description

The `String` class defines the element type of a `StringArray`. `String` is defined as:

```
using String = std::basic_string<char16_t>;
```

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `String.hpp` |

## See Also

`matlab::data::MATLABString`

**Introduced in R2017b**

# matlab::data::StringArray

C++ class to access MATLAB string arrays

## Description

Use `StringArray` objects to access MATLAB string arrays. To create a `StringArray`, call `createArray` or `createScalar` in the `ArrayFactory` class with a `MATLABString` template.

`StringArray` is defined as:

```
using StringArray = TypedArray<MATLABString>;
```

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `TypedArray.hpp` |

## See Also

`MATLABString`

**Introduced in R2017b**

# matlab::data::StructArray

C++ class to access MATLAB struct arrays

## Description

Use `StructArray` objects to work with MATLAB struct arrays. To access a field for a single element in the array, use the field name. To create a `StructArray` object, call `createStructArray` in the `ArrayFactory` class.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Base class: | `TypedArray<Struct>` |
| Include: | `StructArray.hpp` |

## Constructors


- "Copy Constructors" on page 1-73
- "Copy Assignment Operators" on page 1-73
- "Move Constructors" on page 1-74
- "Move Assignment Operators" on page 1-74


### Copy Constructors

```
StructArray(const StructArray& rhs)
```

```
StructArray(const Array& rhs)
```

#### Description

Creates a shared data copy of a `StructArray` object.

#### Parameters

| | |
|---|---|
| `const StructArray& rhs` | Value to copy. |
| `const Array& rhs` | Value specified as `ArrayType::STRUCT` object. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::STRUCT`. |

### Copy Assignment Operators

```
StructArray& operator=(const StructArray& rhs)
```

```
StructArray& operator=(const Array& rhs)
```

**Description**

Assigns a shared data copy to a `StructArray` object.

**Parameters**

| | |
|---|---|
| `const StructArray& rhs` | Value to copy. |
| `const Array& rhs` | Value specified as `ArrayType::STRUCT` object. |

**Returns**

| | |
|---|---|
| `StructArray&` | Updated instance. |

**Throws**

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::STRUCT`. |

### Move Constructors

`StructArray(StructArray&& rhs)`

`StructArray(Array&& rhs)`

**Description**

Moves contents of a `StructArray` object to a new instance.

**Parameters**

| | |
|---|---|
| `StructArray&& rhs` | Value to move. |
| `Array&& rhs` | Value specified as `ArrayType::STRUCT` object. |

**Throws**

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` is not `ArrayType::STRUCT`. |

### Move Assignment Operators

`StructArray& operator=(StructArray&& rhs)`

**Description**

Assigns the input to this `StructArray` object.

**Parameters**

| | |
|---|---|
| `StructArray&& rhs` | Value to move. |

**Returns**

| | |
|---|---|
| `StructArray&` | Updated instance. |

**Throws**

None

## Destructor

`~StructArray()`

### Description

Free memory for `StructArray` object.

## Member Functions

- “getFieldNames” on page 1-75
- “getNumberOfFields” on page 1-75

### getFieldNames

```
Range<ForwardIterator, MatlabFieldIdentifier const> getFieldNames() const
```

#### Returns

| | |
|---|---|
| `Range<ForwardIterator, MatlabFieldIdentifier const>` | Contains `begin` and `end` iterators that enable access to all fields in `StructArray` object. |

#### Throws

None

### getNumberOfFields

```
size_t getNumberOfFields() const
```

#### Returns

| | |
|---|---|
| `size_t` | Number of fields. |

#### Throws

None

## Examples

### Create StructArray

Assume that you have the following MATLAB structure.

```
s = struct('loc', {'east', 'west'}, 'data', {[1, 2, 3], [4., 5., 6., 7., 8.]})
```

Create a variable containing the data for `loc` `east`.

```
val = s(1).data
```

The following C++ code creates these variables.

```
#include "MatlabDataArray.hpp"

int main() {
    using namespace matlab::data;
    ArrayFactory factory;
```

```
    StructArray S = factory.createStructArray({ 1,2 }, { "loc", "data" });
    S[0]["loc"] = factory.createCharArray("east");
    S[0]["data"] = factory.createArray<uint8_t>({ 1, 3 }, { 1, 2, 3 });
    S[1]["loc"] = factory.createCharArray("west");
    S[1]["data"] = factory.createArray<double>({ 1, 5 }, { 4., 5., 6., 7., 8. });

    Reference<Array> val = S[0]["data"];
    return 0;
}
```

## See Also

`MATLABFieldIdentifier` | `Range` | `Struct` | `createStructArray`

### Topics

"Create Structure Array and Send to MATLAB"

**Introduced in R2017b**

# matlab::data::Reference<StructArray>

C++ class to get reference to `StructArray`

## Description

The `StructArrayExt` class extends the APIs available to a reference to a `StructArray`.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Base class: | `Reference<Array>` |
| Include: | `TypedArrayRef.hpp` |

## Member Functions

- “getFieldNames” on page 1-77
- “getNumberOfFields” on page 1-77

### getFieldNames

```
Range<ForwardIterator, MATLABFieldIdentifier const> getFieldNames() const
```

#### Returns

| | |
|---|---|
| `Range<ForwardIterator, MatlabFieldIdentifier const>` | Contains `begin` and `end` methods that enable access to all fields in `StructArray` object. |

#### Throws

None

### getNumberOfFields

```
size_t getNumberOfFields() const
```

#### Returns

| | |
|---|---|
| `size_t` | Number of fields. |

#### Throws

None

## See Also

`Reference<TypedArray<T>>` | `StructArray`

**Introduced in R2017b**

# matlab::data::Struct

Element type for MATLAB struct arrays

## Description

`Struct` is the element type for a `StructArray` object.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `Struct.hpp` |

## Iterators

- “Begin Iterators” on page 1-78
- “End Iterators” on page 1-78

### Begin Iterators

`const_iterator begin() const`

`const_iterator cbegin() const`

#### Returns

| | |
|---|---|
| `const_iterator` | Iterator to beginning of array, specified as `TypedIterator<typename std::add_const<T>::type>` |

#### Throws

None

### End Iterators

`const_iterator end() const`

`const_iterator cend() const`

#### Returns

| | |
|---|---|
| `const_iterator` | Iterator to end of array, specified as `TypedIterator<typename std::add_const<T>::type>` |

#### Throws

None

## Indexing Operators

### `operator[]`

`Array operator[](std::string idx) const`

#### Description

Enables `[]` indexing on a `StructArray` object. Indexing is 0-based.

#### Parameters

| | |
|---|---|
| `std::string idx` | Field name. |

#### Returns

| | |
|---|---|
| `Array` | Shared copy of `Array` found at specified field. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidFieldNameException` | Field does not exist in this `StructArray`. |

## See Also

"createStructArray" on page 1-7 | `StructArray`

**Introduced in R2017b**

# matlab::data::Reference<Struct>

C++ class to get reference to element of `StructArray`

## Description

Use the `Reference<Struct>` class to access an element of a `StructArray`.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `StructRef.hpp` |

## Indexing Operators

### `operator[]`

```
Reference<Array> operator[](std::string idx)
```

```
Array operator[](std::string idx) const
```

#### Description

Index into the Struct with a field name.

#### Parameters

| | |
|---|---|
| `std::string idx` | Field name. |

#### Returns

| | |
|---|---|
| `Reference<Array>` | Reference to `Array` found at specified field. |
| `Array` | Shared copy of `Array` found at specified field. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidFieldNameException` | Field does not exist in the struct. |

## Iterators


- "Begin Iterators" on page 1-80
- "End Iterators" on page 1-81


### Begin Iterators

```
iterator begin()
```

```
const_iterator begin() const
```

```
const_iterator cbegin() const
```

#### Returns

| | |
|---|---|
| `iterator` | Iterator to beginning of list of fields, specified as `TypedIterator<T>`. |
| `const_iterator` | Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`. |

#### Throws

None

### End Iterators

`iterator end()`

`const_iterator end() const`

`const_iterator cend() const`

#### Returns

| | |
|---|---|
| `iterator` | Iterator to end of list of fields, specified as `TypedIterator<T>`. |
| `const_iterator` | Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`. |

#### Throws

None

## Cast

### `Struct()`

`operator Struct() const`

#### Returns

| | |
|---|---|
| `Struct` | Shared copy of `Struct`. |

#### Throws

None

## See Also

**Introduced in R2017b**

# matlab::data::TypedArray<T>

Templated C++ class to access array data

## Description

The templated `TypedArray` class provides type-safe APIs to handle all MATLAB array types (except sparse arrays). To create a `TypedArray`, call `createArray` or `createScalar` in the `ArrayFactory` class with one of the templates listed in "Template Instantiations" on page 1-82.

This class defines the following iterator types:

```
using iterator = TypedIterator<T>;
using const_iterator = TypedIterator<T const>;
```

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Base class: | `matlab::data::Array` |
| Include: | `TypedArray.hpp` |

### Template Parameters

| | |
|---|---|
| `T` | Type of element referred to. |

### Template Instantiations

| |
|---|
| `double` |
| `float` |
| `int8_t` |
| `uint8_t` |
| `int16_t` |
| `uint16_t` |
| `int32_t` |
| `uint32_t` |
| `int64_t` |
| `uint64_t` |
| `char16_t` |
| `bool` |
| `std::complex<double>` |
| `std::complex<float>` |
| `std::complex<int8_t>` |
| `std::complex<uint8_t>` |
| `std::complex<int16_t>` |

```
std::complex<uint16_t>
std::complex<int32_t>
std::complex<uint32_t>
std::complex<int64_t>
std::complex<uint64_t>
matlab::data::Array
matlab::data::Struct
matlab::data::Enumeration
matlab::data::MATLABString
```

## Constructors


- "Copy Constructor" on page 1-83
- "Copy Assignment Operator" on page 1-83
- "Move Constructor" on page 1-84
- "Move Assignment Operator" on page 1-84


### Copy Constructor

```
TypedArray(const TypedArray<T>& rhs)
```

```
TypedArray(const Array& rhs)
```

#### Description

Creates a shared data copy of the input.

#### Parameters

| | |
|---|---|
| `const TypedArray<T>& rhs` | Value to be copied. |
| `const Array& rhs` | Value specified as `matlab::data::Array` object. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` does not match the type for `TypedArray<T>`. |

### Copy Assignment Operator

```
TypedArray<T>& operator=(const TypedArray<T>& rhs)
```

```
TypedArray<T>& operator=(const Array& rhs)
```

#### Description

Assigns a shared data copy of the input to this `TypedArray<T>`.

#### Parameters

| | |
|---|---|
| `const TypedArray<T>& rhs` | Value to be copied. |

| | |
|---|---|
| `const Array& rhs` | Value specified as `matlab::data::Array` object. |

#### Returns

| | |
|---|---|
| `TypedArray<T>&` | Updated instance. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` does not match the type for `TypedArray<T>`. |

### Move Constructor

```
TypedArray(TypedArray<T>&& rhs)
```

```
TypedArray(Array&& rhs)
```

#### Description

Moves contents of the input to a new instance.

#### Parameters

| | |
|---|---|
| `TypedArray<T>&& rhs` | Value to be moved. |
| `Array&& rhs` | Value specified as `matlab::data::Array` object. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input does not match. |

### Move Assignment Operator

```
TypedArray<T>& operator=(TypedArray<T>&& rhs)
```

```
TypedArray<T>& operator=(Array&& rhs)
```

#### Description

Moves the input to this `TypedArray<T>` object.

#### Parameters

| | |
|---|---|
| `TypedArray<T>&& rhs` | Value to move. |

#### Returns

| | |
|---|---|
| `TypedArray<T>&` | Updated instance. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidArrayTypeException` | Type of input `Array` does not match the type for `TypedArray<T>`. |

## Destructor

```
virtual ~TypedArray()
```

## Iterators


- "Begin Iterators" on page 1-85
- "End Iterators" on page 1-85


### Begin Iterators

```
iterator begin()
```

```
const_iterator begin() const
```

```
const_iterator cbegin() const
```

#### Returns

| | |
|---|---|
| `iterator` | Iterator to beginning of array, specified as `TypedIterator<T>`. |
| `const_iterator` | Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`. |

#### Throws

None

### End Iterators

```
iterator end()
```

```
const_iterator end() const
```

```
const_iterator cend() const
```

#### Returns

| | |
|---|---|
| `iterator` | Iterator to end of array, specified as `TypedIterator<T>`. |
| `const_iterator` | Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`. |

#### Throws

None

## Indexing Operators

### operator[]

```
ArrayElementTypedRef<T, std::is_const<T>::value> operator[](size_t idx)
```

```
ArrayElementTypedRef<T, true> operator[](size_t idx) const
```

#### Description

Enables [] indexing on a TypedArray. Indexing is 0-based.

#### Parameters

| | |
|---|---|
| size_t idx | First array index. |

#### Returns

| | |
|---|---|
| ArrayElementTypedRef<T, std::is_const<T>::value> | Temporary object containing index specified. If type T is const, then the return value allows the element of the array to be retrieved, but not modified. Otherwise, the element can be modified or retrieved. |
| ArrayElementTypedRef<T, true> | Temporary object containing index specified. The return value allows the element of the array to be retrieved, but not modified. |

#### Throws

None

## Member Functions

### release

`buffer_ptr_t<T> release()`

#### Description

Release the underlying buffer from the Array. If the Array is shared, a copy of the buffer is made; otherwise, no copy is made. After the buffer is released, the array contains no elements.

#### Returns

| | |
|---|---|
| buffer_ptr_t<T> | unique_ptr containing data pointer. |

#### Throws

| | |
|---|---|
| matlab::data::InvalidArrayTypeException | TypedArray does not support releasing the buffer. |

## Examples

### Assign Values to Array Elements

Create an array equivalent to the MATLAB array [1 2; 3 4], then replace each element of the array with a single value.

```
#include "MatlabDataArray.hpp"

int main() {
    matlab::data::ArrayFactory factory;
    // Create an array equivalent to the MATLAB array [1 2; 3 4].
    matlab::data::TypedArray<double> D = factory.createArray<double>({ 2,2 }, { 1,3,2,4 });
    // Change the values.
    for (auto& elem : D) {
```

```
        elem = 5.5;
    }
    return 0;
}
```

## See Also

Array | ArrayType

### Topics

“Bring Result of MATLAB Calculation Into C++”

**Introduced in R2017b**

# matlab::data::Reference<TypedArray<T>>

Templated C++ class to get reference to `TypedArray`

## Description

The `Reference<TypedArray<T>>` class extends the APIs available to a reference to an `Array`. It derives from the `Reference<Array>` class and provides iterators and type-safe indexing. `Reference<TypedArray<T>>` is not thread-safe - do not pass references to `TypedArray` objects between threads.

`TypedArrayRef` is defined in `TypedArrayRef.hpp` as:

```
template <typename T>
using TypedArrayRef = Reference<TypedArray<T>>;
```

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Base class: | `Reference<Array>` |
| Include: | `TypedArrayRef.hpp` |

## Constructor

`Reference(const Reference<Array>& rhs)`

### Description

Create a `Reference<TypedArray<T>>` object from a `Reference<Array>` object.

### Parameters

| | |
|---|---|
| `const Reference<Array>& rhs` | Value to copy. |

### Throws

| | |
|---|---|
| `matlab::data::TypeMismatchException` | Element of `Array` does not match <T>. |

## Iterators

- "Begin Iterators" on page 1-88
- "End Iterators" on page 1-89

### Begin Iterators

`iterator begin()`

`const_iterator begin() const`

`const_iterator cbegin() const`

#### Returns

| | |
|---|---|
| `iterator` | Iterator to beginning of array, specified as `TypedIterator<T>`. |
| `const_iterator` | Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`. |

#### Throws

None

### End Iterators

`iterator end()`

`const_iterator end() const`

`const_iterator cend() const`

#### Returns

| | |
|---|---|
| `iterator` | Iterator to end of array, specified as `TypedIterator<T>`. |
| `const_iterator` | Iterator, specified as `TypedIterator<typename std::add_const<T>::type>`. |

#### Throws

None

## Indexing Operators

### operator[]

`ArrayElementTypedRef<arr_elem_type, std::is_const<T>::value> operator[](size_t idx)`

`ArrayElementTypedRef<arr_elem_type, true> operator[](size_t idx) const`

#### Description

Enables `[]` indexing on a reference to an `Array`. Indexing is 0-based.

#### Parameters

| | |
|---|---|
| `size_t idx` | First array index. |

#### Returns

| | |
|---|---|
| `ArrayElementTypedRef<arr_elem_type, std::is_const<T>::value>` | Temporary object containing index specified. If type `T` is `const`, then the return value allows the element of the array to be retrieved, but not modified. Otherwise, the element can be modified or retrieved. |
| `ArrayElementTypedRef<arr_elem_type, true>` | Temporary object containing index specified. The return value allows the element of the array to be retrieved, but not modified. |

#### Throws

| | |
|---|---|
| `matlab::data::InvalidFieldNameException` | Field name is invalid for a struct. |

## Other Operators

### `operator=`

`Reference<TypedArray<T>>& operator= (TypedArray<T> rhs)`

#### Description

Assign a `TypedArray` to an element of the referenced `Array`. The `Array` being indexed must be non-`const`.

#### Parameters

| | |
|---|---|
| `TypedArray<T> rhs` | Value to assign. |

#### Returns

| | |
|---|---|
| `Reference<TypedArray<T>>&` | Updated instance. |

#### Throws

None

## See Also

**Introduced in R2017b**

# matlab::data::TypedIterator<T>

Templated C++ class to provide random access iterator

## Description

`TypedIterator` is the return type of all `begin` and `end` functions that support random access.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include: | `TypedIterator.hpp` |

### Template Parameters

| | |
|---|---|
| `T` | Type of element referred to. |

### Template Instantiations

```
double
float
int8_t
uint8_t
int16_t
uint16_t
int32_t
uint32_t
int64_t
uint64_t
char16_t
bool
std::complex<double>
std::complex<float>
std::complex<int8_t>
std::complex<uint8_t>
std::complex<int16_t>
std::complex<uint16_t>
std::complex<int32_t>
std::complex<uint32_t>
std::complex<int64_t>
std::complex<uint64_t>
matlab::data::Array
```

```
matlab::data::Struct
matlab::data::Enumeration
matlab::data::MATLABString
```

## Constructors


- "Copy Constructors" on page 1-92
- "Copy Assignment Operators" on page 1-92
- "Move Constructors" on page 1-92
- "Move Assignment Operators" on page 1-93


### Copy Constructors

`TypedIterator(const TypedIterator<T>& rhs)`

#### Description

Creates a shared data copy of a `TypedIterator` object.

#### Parameters

| | |
|---|---|
| `const TypedIterator<T>& rhs` | Value to copy. |

#### Throws

None

### Copy Assignment Operators

`TypedIterator<T>& operator=(const TypedIterator<T>& rhs)`

#### Description

Assigns a shared data copy to a `TypedIterator` object.

#### Parameters

| | |
|---|---|
| `const TypedIterator<T>& rhs` | Value to copy. |

#### Returns

| | |
|---|---|
| `TypedIterator<T>&` | Updated instance. |

#### Throws

None

### Move Constructors

`TypedIterator(TypedIterator<T> &&rhs)`

#### Description

Moves contents of a `TypedIterator` object to a new instance.

#### Parameters

| | |
|---|---|
| TypedIterator<T>&& rhs | Value to move. |

#### Throws

None

### Move Assignment Operators

TypedIterator<T>& operator=(TypedIterator<T>&& rhs)

#### Description

Assigns the input to this TypedIterator object.

#### Parameters

| | |
|---|---|
| TypedIterator<T>&& rhs | Value to move. |

#### Returns

| | |
|---|---|
| TypedIterator<T>& | Updated instance. |

#### Throws

None

## Other Operators

### operator++

`TypedIterator<T>& operator++()`

**Description**

Pre-increment operator.

**Returns**

| | |
|---|---|
| `TypedIterator<T>&` | Original iterator. |

**Throws**

None

### operator--

`TypedIterator<T>& operator--()`

**Description**

Pre-decrement operator.

**Returns**

| | |
|---|---|
| `TypedIterator<T>&` | Original iterator. |

**Throws**

None

### operator++

`TypedIterator<T> operator++(int)`

**Description**

Post-increment operator.

**Returns**

| | |
|---|---|
| `TypedIterator<T>` | Copy of original iterator. |

**Throws**

None

### operator--

`TypedIterator<T> operator--(int)`

**Description**

Post-decrement operator.

**Returns**

| | |
|---|---|
| `TypedIterator<T>` | Copy of original iterator. |

**Throws**

None

### `operator+=`

`TypedIterator<T>& operator+=(difference_type d)`

**Description**

Addition assignment operator.

**Parameters**

| | |
|---|---|
| `difference_type d` | Amount to add, specified as `std::ptrdiff_t`. |

**Returns**

| | |
|---|---|
| `TypedIterator<T>&` | Updated instance. |

**Throws**

None

### `operator-=`

`TypedIterator<T>& operator-=(difference_type d)`

**Description**

Subtraction assignment operator.

**Parameters**

| | |
|---|---|
| `difference_type d` | Amount to subtract, specified as `std::ptrdiff_t`. |

**Returns**

| | |
|---|---|
| `TypedIterator<T>&` | Updated instance. |

**Throws**

None

### `operator!=`

`bool operator!=(const TypedIterator<T>& rhs) const`

**Parameters**

| | |
|---|---|
| `const TypedIterator<T>& rhs` | Iterator to compare. |

**Returns**

| | |
|---|---|
| `bool` | Returns `true` if iterators do not point to same element. |

**Throws**

None

### operator<

`bool operator<(const TypedIterator<T>& rhs) const`

**Parameters**

| | |
|---|---|
| `const TypedIterator<T>& rhs` | Iterator to compare. |

**Returns**

| | |
|---|---|
| `bool` | Returns `true` if left-side iterator is less than right-side iterator. |

### operator>

`bool operator>(const TypedIterator<T>& rhs) const`

**Parameters**

| | |
|---|---|
| `const TypedIterator<T>& rhs` | Iterator to compare. |

**Returns**

| | |
|---|---|
| `bool` | Returns `true` if left-side iterator is greater than right-side iterator. |

### operator<=

`bool operator<=(const TypedIterator<T>& rhs) const`

**Parameters**

| | |
|---|---|
| `const TypedIterator<T>& rhs` | Iterator to compare. |

**Returns**

| | |
|---|---|
| `bool` | Returns `true` if left-side iterator is less than or equal to right-side iterator. |

**Throws**

None

### operator>=

`bool operator>=(const TypedIterator<T>& rhs) const`

**Parameters**

| | |
|---|---|
| `const TypedIterator<T>& rhs` | Iterator to compare. |

**Returns**

| | |
|---|---|
| `bool` | Returns `true` if left-side iterator is greater than or equal to right-side iterator. |

**Throws**

None

### operator+

`TypedIterator<T> operator+(difference_type d) const`

**Description**

Creates an iterator that is added to this one by the amount passed in.

**Parameters**

| | |
|---|---|
| `difference_type d` | Amount to add, specified as `std::ptrdiff_t`. |

**Returns**

| | |
|---|---|
| `TypedIterator<T>` | Updated instance. |

**Throws**

None

### operator-

`TypedIterator<T> operator-(difference_type d) const`

**Description**

Creates an iterator that is decremented from this one by the amount passed in.

**Parameters**

| | |
|---|---|
| `difference_type d` | Amount to subtract, specified as `std::ptrdiff_t`. |

**Returns**

| | |
|---|---|
| `TypedIterator<T>` | Updated instance. |

**Throws**

None

### operator-

`difference_type operator-(const TypedIterator<T>& rhs) const`

**Parameters**

| | |
|---|---|
| `const TypedIterator<T>& rhs` | Iterator to compare. |

**Returns**

| | |
|---|---|
| `difference_type` | Difference between iterators, specified as `std::ptrdiff_t`. |

**Throws**

None

### operator*

`reference operator*() const`

#### Returns

| | |
|---|---|
| `reference` | Element pointed to by this iterator, specified as:<br>• `T&` for arithmetic types.<br>• `Reference<T>` for non-arithmetic types. |

#### Throws

None

### operator->

```
pointer operator->()
```

#### Returns

| | |
|---|---|
| `pointer` | Pointer to element pointed to by this iterator, specified as:<br>• `T*` for arithmetic types.<br>• `Reference<T>*` for non-arithmetic types. |

#### Throws

None

### operator[]

```
reference operator[](const size_t& rhs) const
```

#### Description

Get a reference using a linear index.

#### Returns

| | |
|---|---|
| `reference` | Element pointed to by this iterator, specified as:<br>• `T&` for arithmetic types.<br>• `Reference<T>` for non-arithmetic types. |

#### Throws

None

## Free Function

### operator==

```
bool operator==(const TypedIterator<T>& rhs) const
```

#### Parameters

| | |
|---|---|
| `const TypedIterator<T>& rhs` | Iterator to compare. |

#### Returns

| | |
|---|---|
| `bool` | Returns `true` if both iterators point to same element. |

#### Throws

None

## See Also

#### Introduced in R2017b

# matlab::data::apply_visitor

Call Visitor class on arrays

## Description

`auto apply_visitor(Array a, V visitor)` dispatch to visitor class operations based on array type.

Use `apply_visitor` to pass in an instance of Array or one of its subclasses and a visitor functor, and invoke the operator() method, which must be defined in the user-defined functor, with the appropriate concrete array type.

### Include

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include | `ArrayVisitors.hpp` |

## Parameters

| | |
|---|---|
| `matlab::data::Array a&&` | The `matlab::data::Array` to operate on with the visitor class, passed:<br>• by value<br>• by const lvalue ref<br>• by rvalue ref<br>• by nonconst lvalue ref<br><br>To modify the original array, pass it by rvalue ref into the operator() method and return the modified array. Then the calling code should move the returned array into the old array. Due to copy-on-write behavior, passing by nonconst lvalue ref does not modify the original array. |
| visitor class V&& | The user-supplied visitor object, passed:<br>• by value<br>• by const lvalue ref<br>• by rvalue ref<br>• by nonconst lvalue ref |

## Return Value

| | |
|---|---|
| `auto` | Outputs returned by the visitor. |

## See Also

**Topics**
"Operate on C++ Arrays Using Visitor Pattern"

**Introduced in R2017b**

# matlab::data::apply_visitor_ref

Call Visitor class on array references

## Description

`auto apply_visitor_ref(const ArrayRef& a, V visitor)` dispatch to visitor class operations based on array reference type.

### Include

| | |
|---|---|
| Namespace: | `matlab::data` |
| Include | `ArrayVisitors.hpp` |

## Parameters

| | |
|---|---|
| `const matlab::data::ArrayRef& a` | A `matlab::data::ArrayRef` reference to the array to operate on with the visitor class. |
| visitor class V | The user-supplied visitor class. |

## Return Value

| | |
|---|---|
| `auto` | Outputs returned by the visitor. |

## See Also

**Topics**
"Operate on C++ Arrays Using Visitor Pattern"

**Introduced in R2017b**

# matlab::mex::Function

Base class for C++ MEX functions

## Description

The `MexFunction` class that you implement in C++ MEX functions must inherit from the `matlab.mex.Function` class. The `matlab.mex.Function` class enables access to the C++ Engine API and defines a virtual `operator()` function that your `MexFunction` class must override.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::mex` |
| Include: | `mexAdapter.hpp` — Include this file only once for the implementation of `MexFunction` class |

## Member Functions


- “operator()” on page 1-103
- “getEngine” on page 1-103
- “mexLock” on page 1-104
- “mexUnlock” on page 1-104
- “getFunctionName” on page 1-104


### operator()

`virtual void operator()(ArgumentList outputs, ArgumentList inputs)`

Function call operator that you must override in the `MexFunction` class. This operator enables instances of your `MexFunction` class to be called like a function.

#### Parameters

| | |
|---|---|
| `matlab::mex::ArgumentList outputs` | Collection of `matlab::data::Array` objects that are returned to MATLAB |
| `matlab::mex::ArgumentList inputs` | Collection of `matlab::data::Array` objects that are passed to the MEX function from MATLAB |

#### Examples

```
class MexFunction : public matlab::mex::Function {
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
        ...
    }
}
```

### getEngine

`std::shared_ptr<matlab::engine::MATLABEngine> getEngine()`

Returns a pointer to the `MATLABEngine` object, which enables access to the C++ Engine API.

#### Returns

| | |
|---|---|
| `std::shared_ptr<matlab::engine::MATLABEngine>` | Pointer to `MATLABEngine` object |

#### Examples

Call the MATLAB `clear` function.

```
std::shared_ptr<MATLABEngine> matlabPtr = getEngine();
matlabPtr->eval(matlab::engine::convertUTF8StringToUTF16String("clear"));
```

### mexLock

```
void mexLock()
```

Prevents clearing MEX file from memory. Do not call `mexLock` or `mexUnlock` from a user thread.

#### Examples

Lock the MEX file.

```
mexLock();
```

### mexUnlock

Unlocks MEX file and allows clearing of the file from memory. Do not call `mexLock` or `mexUnlock` from a user thread.

```
void mexLock()
```

#### Examples

Unlock the MEX file.

```
mexUnlock();
```

### getFunctionName

```
std::u16string getFunctionName() const
```

Returns the name of the MEX function, which is the name of the source file.

#### Examples

Get the file name of the currently executing MEX function.

```
std::u16string fileName = getFunctionName();
```

## See Also

`matlab::mex::ArgumentList`

#### Topics

"C++ MEX API"
"Structure of C++ MEX Function"

**Introduced in R2018a**

# matlab::mex::ArgumentList

Container for inputs and outputs from C++ MEX functions

## Description

C++ MEX functions pass inputs and outputs as `matlab::data::Array` objects contained in `matlab::mex::ArgumentList` objects. The `MexFunction::operator()` accepts two arguments, one for inputs and one for outputs, defined as `matlab::mex::ArgumentList`.

`ArgumentList` is a wrapper enabling iteration over the underlying collections holding the input and output data.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::mex` |
| Include: | `mex.hpp` |

## Member Functions


- “operator[ ]” on page 1-105
- “begin” on page 1-106
- “end” on page 1-106
- “size” on page 1-106
- “empty” on page 1-107


### operator[ ]

```
matlab::data::Array operator[](size_t idx)
```

Enables `[]` indexing into the elements of an `ArgumentList`.

#### Parameters

| | |
|---|---|
| `size_t idx` | Index into the elements of the input array, which are the input arguments to the MEX function |

#### Returns

| | |
|---|---|
| `matlab::data::Array` | Iterator pointing to the first element in the `ArgumentList` array |

#### Examples

Call a MEX function from MATLAB with an array, a scalar, and a character vector as inputs and a single output:

```
a = myMEXFunction(array, scalar, 'character vector')
```

Assign the first input argument to a `TypedArray`, the second input to a scalar `const double` (assume both are of type `double` in MATLAB), and the third input as a `matlab::data::CharArray`.

```
void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
    matlab::data::TypedArray<double> inArray = inputs[0];
    const double inScalar = inputs[1][0];
    matlab::data::CharArray inChar = inputs[2];
    result = ...
    outputs[0] = result;
}
```

### begin

`iterator_type begin()`

Returns an iterator pointing to the first element in the `ArgumentList` array.

#### Returns

| | |
|---|---|
| `iterator_type` | Iterator pointing to the first element in the `ArgumentList` array |

#### Examples

Build a vector from the input arguments.

```
void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
    std::vector<matlab::data::TypedArray<double>> vectorDoubles(inputs.begin(), inputs.end());
    ...
}
```

### end

`iterator_type end()`

Returns an iterator pointing past the last element in the `ArgumentList` array.

#### Returns

| | |
|---|---|
| `iterator_type` | Iterator pointing past the last element in the `ArgumentList` array |

### size

`size_t numArgs size()`

Returns the number of elements in the argument list. Use to check the number of inputs and outputs specified at the call site.

#### Returns

| | |
|---|---|
| `size_t` | Size of the `ArgumentList` array |

#### Examples

Determine if the MEX function is called with three input arguments.

```
class MexFunction : public matlab::mex::Function {
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
    if (inputs.size() == 3) {
        // MEX function called with three input arguments
        ...
    }
}
```

### empty

`bool empty()`

Returns logical value indicating if argument list is empty.

#### Returns

| | |
|---|---|
| `bool` | Returns logical `true` if the argument list is empty (`size() == 0`) |

#### Examples

Determine if the MEX function is called with no input arguments.

```
class MexFunction : public matlab::mex::Function {
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs) {
    if (inputs.empty()) {
        // MEX function called with no input arguments
        ...
    }
}
```

## See Also

`matlab::mex::Function`

#### Topics

"C++ MEX API"
"Structure of C++ MEX Function"

#### Introduced in R2018a

# matlab::engine::MATLABEngine

Evaluate MATLAB functions from C++ program

## Description

The `matlab::engine::MATLABEngine` class uses a MATLAB process as a computational engine for C++. This class provides an interface between the C++ language and MATLAB, enabling you to evaluate MATLAB functions and expressions from C++ programs.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include: | `MatlabEngine.hpp` |

## Factory Methods

The `matlab::engine::MATLABEngine` class provides methods to start MATLAB and to connect to a shared MATLAB session synchronously or asynchronously.

- `matlab::engine::startMATLAB` — Start MATLAB synchronously
- `matlab::engine::startMATLABAsync` — Start MATLAB asynchronously
- `matlab::engine::connectMATLAB` — Connect to shared MATLAB session synchronously
- `matlab::engine::connectMATLABAsync` — Connect to shared MATLAB session asynchronously

## Unsupported Startup Options

The engine does not support these MATLAB startup options:

- `-h`
- `-help`
- `-?`
- `-n`
- `-e`
- `-softwareopengl`
- `-logfile`

For information on MATLAB startup options, see “Commonly Used Startup Options”. For an example of how to use MATLAB startup options when starting engine applications, see “Start MATLAB with Startup Options”.

## Method Summary

### Member Functions



## Member Function Details

### feval

```
std::vector<matlab::data::Array> feval(const matlab::engine::String &function,
   const size_t numReturned,
   const std::vector<matlab::data::Array> &args,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>())

matlab::data::Array feval(const matlab::engine::String &function,
   const std::vector<matlab::data::Array> &args,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>())

matlab::data::Array feval(const matlab::engine::String &function,
   const matlab::data::Array &arg,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>())
```

```
ResultType feval(const matlab::engine::String &function,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output,
   const std::shared_ptr<matlab::engine::StreamBuffer> &error,
   RhsArgs&&... rhsArgs )

ResultType feval(const matlab::engine::String &function,
   RhsArgs&&... rhsArgs)
```

#### Description

Evaluate MATLAB functions with input arguments synchronously. Use feval when you want to pass arguments from C++ to MATLAB and when you want to return a result from MATLAB to C++.

Inputs and outputs can be types defined by the MATLAB Data Array API or can be native C++ types.

#### Parameters

| | |
|---|---|
| const matlab::engine::String &function | Name of the MATLAB function or script to evaluate. Specify the name as an std::u16string. Also, you can specify this parameter as an std::string. |
| const size_t numReturned | Number of returned values |
| const std::vector<matlab::data::Array> &args | Multiple input arguments to pass to the MATLAB function in an std::vector. The vector is converted to a column array in MATLAB. |
| const matlab::data::Array arg | Single input argument to pass to the MATLAB function. |
| const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>() | Stream buffer used to store the standard output from the MATLAB function. |
| const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>() | Stream buffer used to store the error message from the MATLAB function. |
| RhsArgs&&... rhsArgs | Native C++ data types used for function inputs. feval accepts scalar inputs of these C++ data types: bool, int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t, float, double. |

#### Return Value

| | |
|---|---|
| std::vector<matlab::data::Array> | Outputs returned from MATLAB function. |
| matlab::data::Array | Single output returned from MATLAB function. |
| ResultType | Output returned from MATLAB function as a user-specified type. Can be an std::tuple if returning multiple arguments. |

**Exceptions**

| | |
|---|---|
| `matlab::engine::MATLABNotAvailableException` | The MATLAB session is not available. |
| `matlab::engine::MATLABExecutionException` | There is a MATLAB runtime error in the function. |
| `matlab::engine::TypeConversionException` | The result of a MATLAB function cannot be converted to the specified type. |
| `matlab::engine::MATLABSyntaxException` | There is a syntax error in the MATLAB function. |

**Examples**

This example passes an array of numeric values to a MATLAB function. The code performs these steps:

- Creates a `matlab::data::Array` with the dimensions 2-by-3 from a vector of numeric values of type double.
- Starts a shared MATLAB session.
- Passes the data array to the MATLAB `sqrt` function and returns the result to C++.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
using namespace matlab::engine;

    std::vector<double> cppData{ 4, 8, 12, 16, 20, 24 };

    // Create a 2-by-3 matlab data array
    matlab::data::ArrayFactory factory;
    auto inputArray = factory.createArray({ 2, 3 }, cppData.cbegin(), cppData.cend());

    // Start MATLAB engine
    std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();

    // Pass data array to MATLAB sqrt function
    // And return results.
    auto result = matlabPtr->feval(u"sqrt", inputArray);
```

When calling `feval` using native C++ types, the input arguments are restricted to scalar values. For example, this code returns the square root of a scalar value.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

    // Start MATLAB engine synchronously
    std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();

    // Call sqrt function
    double result = matlabPtr->feval<double>(u"sqrt", double(27));
```

For functions that return multiple output arguments, you can use the MATLAB data API or, if using C++ types, an `std::tuple`. For an example, see “Call Function with Native C++ Types”.

**Related Topics**

“Call MATLAB Functions from C++”

“MATLAB Data API”

## fevalAsync

```
FutureResult<std::vector<matlab::data::Array>> fevalAsync(const matlab::engine::String &function,
   const size_t numReturned,
   const std::vector<matlab::data::Array> &args,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>())
```

```
FutureResult<matlab::data::Array> fevalAsync(const matlab::engine::String &function,
   const std::vector<matlab::data::Array> &args,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>())
```

```
FutureResult<matlab::data::Array> fevalAsync(const matlab::engine::String &function,
   const matlab::data::Array &arg,
   const std::shared_ptr<matlab::engine::StreamBuffer> & output = std::shared_ptr<matlab::engine::StreamBuffer>(),
   const std::shared_ptr<matlab::engine::StreamBuffer> & error = std::shared_ptr<matlab::engine::StreamBuffer>())
```

```
FutureResult<ResultType> fevalAsync(const matlab::engine::String &function,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output,
   const std::shared_ptr<matlab::engine::StreamBuffer> &error,
   RhsArgs&&... rhsArgs)
```

```
FutureResult<ResultType> fevalAsync(const matlab::engine::String &function,
   RhsArgs&&... rhsArgs)
```

### Description

Evaluate MATLAB function with input arguments and returned values asynchronously.

### Parameters

| | |
|---|---|
| `const matlab::engine::String &function` | Name of the MATLAB function or script to evaluate. Specify the name as an `std::u16string`. Also, you can specify this parameter as an `std::string`. |
| `const size_t numReturned` | Number of returned values |
| `const std::vector<matlab::data::Array> &args` | Multiple input arguments to pass to the MATLAB function in an `std::vector`. The vector is converted to a column array in MATLAB. |
| `const matlab::data::Array arg` | Single input argument to pass to the MATLAB function. |
| `const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer>()` | Stream buffer used to store the standard output from the MATLAB function. |
| `const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer>()` | Stream buffer used to store the error message from the MATLAB function. |

| | |
|---|---|
| `RhsArgs&&... rhsArgs` | Native C++ data types used for function inputs. `feval` accepts scalar inputs of these C++ data types: `bool`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `float`, `double`. |

#### Return Value

| | |
|---|---|
| `FutureResult` | A `FutureResult` object used to get the result of calling the MATLAB function. |

#### Exceptions

None

#### Examples

This example passes the scalar double 12.7 to the MATLAB `sqrt` function asynchronously. The `FutureResult` is then used to get the result.

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
using namespace matlab::engine;
```

```
std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
    matlab::data::ArrayFactory factory;
    matlab::data::Array argument = factory.createScalar<double>(12.7);
    FutureResult<matlab::data::Array> future = matlabPtr->fevalAsync(u"sqrt", std::move(argument));
       ...
    matlab::data::TypedArray<double> result = future.get();
```

#### Related Topics

"Call Function Asynchronously"

### eval

```
void eval(const matlab::engine::String &statement,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer> (),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer> ())
```

#### Description

Evaluate a MATLAB statement as a string synchronously.

#### Parameters

| | |
|---|---|
| `const matlab::engine::String &statement` | MATLAB statement to evaluate |
| `const std::shared_ptr<matlab::engine::StreamBuffer> &output` | Stream buffer used to store the standard output from the MATLAB statement. |
| `const std::shared_ptr<matlab::engine::StreamBuffer> &error` | Stream buffer used to store the error message from the MATLAB command. |

**Exceptions**

| | |
|---|---|
| `matlab::engine::MATLABNotAvailableException` | The MATLAB session is not available. |
| `matlab::engine::MATLABExecutionException` | There is a runtime error in the MATLAB statement. |
| `matlab::engine::MATLABSyntaxException` | There is a syntax error in the MATLAB statement. |

**Examples**

This example evaluates the following MATLAB statement.

```
a = sqrt(12.7);
```

The statement creates the variable `a` in the MATLAB base workspace.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlabPtr->eval(u"a = sqrt(12.7);");
```

**Related Topics**

"Evaluate MATLAB Statements from C++"

## evalAsync

```
FutureResult<void> evalAsync(const matlab::engine::String &str,
   const std::shared_ptr<matlab::engine::StreamBuffer> &output = std::shared_ptr<matlab::engine::StreamBuffer> (),
   const std::shared_ptr<matlab::engine::StreamBuffer> &error = std::shared_ptr<matlab::engine::StreamBuffer> ())
```

**Description**

Evaluate a MATLAB statement as a string asynchronously.

**Parameters**

| | |
|---|---|
| `const matlab::engine::String& str` | MATLAB statement to evaluate |
| `const std::shared_ptr<matlab::engine::StreamBuffer> & output` | Stream buffer used to store the standard output from the MATLAB statement. |
| `const std::shared_ptr<matlab::engine::StreamBuffer> & error` | Stream buffer used to store the error message from the MATLAB command. |

**Return Value**

| | |
|---|---|
| `FutureResult` | A `FutureResult` object used to wait for the completion of the MATLAB statement. |

#### Exceptions

None

#### Examples

This example evaluates the following MATLAB statement asynchronously.

```
a = sqrt(12.7);
```

The statement creates the variable `a` in the MATLAB base workspace.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
FutureResult<void> future = matlabPtr->evalAsync(u"a = sqrt(12.7);");
```

#### Related Topics

"Evaluate MATLAB Statements from C++"

### getVariable

```
matlab::data::Array getVariable(const matlab::engine::String &varName,
    matlab::engine::WorkspaceType workspaceType = matlab::engine::WorkspaceType::BASE)
```

#### Description

Get a variable from the MATLAB base or global workspace.

#### Parameters

| | |
|---|---|
| `const matlab::engine::String& varName` | Name of a variable in the MATLAB workspace. Specify the name as an `std::u16string`. Also, you can specify this parameter as an `std::string`. |
| `matlab::engine::WorkspaceType workspaceType = matlab::engine::WorkspaceType::BASE` | MATLAB workspace (BASE or GLOBAL) to get the variable from. For more information, see `global`. |

#### Return Value

| | |
|---|---|
| `matlab::data::Array` | Variable obtained from the MATLAB base or global workspace |

#### Exceptions

| | |
|---|---|
| `matlab::engine::MATLABNotAvailableException` | The MATLAB session is not available. |
| `matlab::engine::MATLABExecutionException` | The requested variable does not exist in the specified MATLAB base or global workspace. |

#### Examples

This example gets a variable named `varName` from the MATLAB base workspace.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;
```

```
std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlab::data::Array varName = matlabPtr->getVariable(u"varName");
```

#### Related Topics

"Pass Variables from MATLAB to C++"

### getVariableAsync

```
FutureResult<matlab::data::Array> getVariableAsync(const matlab::engine::String &varName,
    matlab::engine::WorkspaceType workspaceType = matlab::engine::WorkspaceType::BASE)
```

#### Description

Get a variable from the MATLAB base or global workspace asynchronously.

#### Parameters

| | |
|---|---|
| `const matlab::engine::String& varName` | Name of the variable in MATLAB workspace. Specify the name as an `std::u16string`. Also, you can specify this parameter as an `std::string`. |
| `matlab::engine::WorkspaceType workspaceType = matlab::engine::WorkspaceType::BASE` | MATLAB workspace (BASE or GLOBAL) to get the variable from. For more information, see `global`. |

#### Return Value

| | |
|---|---|
| `FutureResult` | A `FutureResult` object that you can use to get the variable obtained from the MATLAB workspace as a `matlab.data.Array`. |

#### Exceptions

None

#### Examples

This example gets a variable named `varName` from the MATLAB base workspace asynchronously.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;
```

```
std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
FutureResult<matlab::data::Array> future = matlabPtr->getVariableAsync(u"varName");
...
matlab::data::Array varName = future.get();
```

#### Related Topics

"Pass Variables from MATLAB to C++"

### setVariable

```
void setVariable(const matlab::engine::String &varName,
   const matlab::data::Array &var,
   matlab::engine::WorkspaceType workspaceType = matlab::engine::WorkspaceType::BASE)
```

#### Description

Put a variable into the MATLAB base or global workspace. If a variable with the same name exists in the MATLAB workspace, `setVariable` overwrites it.

#### Parameters

| | |
|---|---|
| `const matlab::engine::String& varName` | Name of the variable to create in the MATLAB workspace. Specify the name as an `std::u16string`. Also, you can specify this parameter as an `std::string`. |
| `const matlab::data::Array var` | Value of the variable to create in the MATLAB workspace |
| `matlab::engine::WorkspaceType workspaceType = matlab::engine::WorkspaceType::BASE` | Put the variable in the MATLAB BASE or GLOBAL workspace. For more information, see `global`. |

#### Exceptions

| | |
|---|---|
| `matlab::engine::MATLABNotAvailableException` | The MATLAB session is not available. |

#### Examples

This example puts the variable named `data` in the MATLAB base workspace.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlab::data::Array data = factory.createArray<double>({ 1, 3 }, { 4, 8, 6 });
matlabPtr->setVariable(u"data", data);
```

#### Related Topics

"Pass Variables from C++ to MATLAB"

### setVariableAsync

```
FutureResult<void> setVariableAsync(const matlab::engine::String &varName,
   const matlab::data::Array var,
   matlab::engine::WorkspaceType workspaceType = matlab::engine::WorkspaceType::BASE)
```

#### Description

Put a variable into the MATLAB base or global workspace asynchronously. If a variable with the same name exists in the MATLAB base workspace, `setVariableAsync` overwrites it.

#### Parameters

| | |
|---|---|
| `const matlab::engine::String& varName` | Name of the variable to create in the MATLAB workspace. Specify the name as an `std::u16string`. Also, you can specify this parameter as an `std::string`. |
| `const matlab::data::Array var` | Value of the variable to create in the MATLAB workspace |

| | |
|---|---|
| `matlab::engine::WorkspaceType workspaceType = matlab::engine::WorkspaceType::BASE` | Put the variable in the MATLAB BASE or GLOBAL workspace. For more information, see `global`. |

**Exceptions**

None

**Example**

This example puts the variable named `data` in the MATLAB base workspace.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlab::data::Array data = factory.createArray<double>({ 1, 3 }, { 4., 8., 6. });
FutureResult<void> future = matlabPtr->setVariableAsync(u"data", data);
```

**Related Topics**

"Pass Variables from MATLAB to C++"

### getProperty

```
matlab::data::Array getProperty(const matlab::data::Array &objectArray,
    size_t index,
    const matlab::engine::String &propertyName)

matlab::data::Array getProperty(const matlab::data::Array &object,
   const matlab::engine::String &propertyName)
```

**Description**

Get the value of an object property. If the object input argument is an array of objects, specify the index of the array element that corresponds to the object whose property value you want to get.

**Parameters**

| | |
|---|---|
| `const matlab::data::Array &objectArray` | Array of MATLAB objects |
| `const matlab::data::Array &object` | Scalar MATLAB object |
| `size_t index` | Zero-based index into the object array, specifying the object in that array whose property value is returned |
| `const String &propertyName` | Name of the property. Specify the name as an `std::u16string`. Also, you can specify this parameter as an `std::string`. |

**Return Value**

| | |
|---|---|
| `matlab::data::Array` | Value of the named property |

**Exceptions**

| | |
|---|---|
| `matlab::engine::MATLABNotAvailableException` | The MATLAB session is not available. |
| `matlab::engine::MATLABExecutionException` | The property does not exist. |

**Examples**

This example evaluates a MATLAB statement in a try/catch block using `MATLABEngine::eval`. The `MATLABEngine::getVariable` member function returns the exception object. `MATLABEngine::getProperty` returns the exception `message` property value as a `matlab::data::CharArray`.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlabPtr->eval(u"try; surf(4); catch me; end");
matlab::data::Array mException = matlabPtr->getVariable(u"me");
matlab::data::CharArray message = matlabPtr->getProperty(mException, u"message");
std::cout << "messages is: " << message.toAscii() << std::endl;
```

**Related Topics**

"Get MATLAB Objects and Access Properties"

## getPropertyAsync

```
FutureResult<matlab::data::Array> getPropertyAsync(const matlab::data::Array &objectArray,
    size_t index,
    const matlab::engine::String &propertyName)

FutureResult<matlab::data::Array> getPropertyAsync(const matlab::data::Array &object,
   const matlab::engine::String &propertyName)
```

**Description**

Get the value of an object property asynchronously. If the object input argument is an array of objects, specify the index of the array element that corresponds to the object whose property value you want to get.

**Parameters**

| | |
|---|---|
| `const matlab::data::Array &objectArray` | Array of MATLAB objects |
| `const matlab::data::Array &object` | Scalar MATLAB object |
| `size_t index` | Zero-based index into the object array, specifying the object in that array whose property value is returned |
| `const matlab::engine::String &propertyName` | Name of the property. Specify the name as an `std::u16string`. Also, you can specify this parameter as an `std::string`. |

#### Return Value

| | |
|---|---|
| `FutureResult` | `FutureResult` object that is used to synchronize the operation. |

#### Exceptions

None

#### Examples

This example evaluates a MATLAB statement in a try/catch block using `MATLABEngine::eval`. The `MATLABEngine::getVariable` member function returns the exception object. `MATLABEngine::getPropertyAsync` returns a `FutureResult` that you use to get the exception `message` property value as a `matlab::data::CharArray`.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlabPtr->eval(u"try;surf(4);catch me;end");
matlab::data::Array mException = matlabPtr->getVariable(u"me");
FutureResult<matlab::data::Array> future = matlabPtr->getPropertyAsync(mException, u"message");
matlab::data::CharArray message = future.get();
std::cout << "messages is: " << message.toAscii() << std::endl;
```

#### Related Topics

"Get MATLAB Objects and Access Properties"

### setProperty

```
void setProperty(matlab::data::Array &objectArray,
    size_t index,
    const matlab::engine::String &propertyName,
    const matlab::data::Array &propertyValue)

void setProperty(matlab::data::Array &object,
    const matlab::engine::String &propertyName,
    const matlab::data::Array &propertyValue)
```

#### Description

Set the value of an object property. If the object input argument is an array of objects, specify the index of the array element that corresponds to the object whose property value you want to set.

#### Parameters

| | |
|---|---|
| `matlab::data::Array &objectArray` | Array of MATLAB objects |
| `matlab::data::Array &object` | Scalar MATLAB object |
| `size_t index` | Zero-based index into the object array, specifying the object in that array whose property value is set |
| `const matlab::engine::String &propertyName` | Name of the property to set. Specify the name as an `std::u16string`. Also, you can specify this parameter as an `std::string`. |
| `const matlab::data::Array &propertyValue` | Value assigned to the property |

**Exceptions**

| | |
|---|---|
| `matlab::engine::MATLABNotAvailableException` | The MATLAB session is not available. |
| `matlab::engine::MATLABExecutionException` | The property does not exist. |

**Examples**

This example shows how to set a MATLAB object property. It creates a MATLAB graph and returns the line handle object. Setting the value of the line `LineStyle` property to the character `:` changes the property value of the line object in MATLAB and updates the line style of the graph.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlab::data::ArrayFactory factory;
matlab::data::Array yData = factory.createArray<double>({ 1, 5 }, { 4.0, 11.0, 4.7, 36.2, 72.3 });
matlab::data::Array lineHandle = matlabPtr->feval(u"plot", yData);
matlab::data::CharArray lineStyle = factory.createCharArray(":");
matlabPtr->setProperty(lineHandle, u"LineStyle", lineStyle);
```

**Related Topics**

"Set Property on MATLAB Object"

### setPropertyAsync

```
FutureResult<void> setPropertyAsync(matlab::data::Array &objectArray,
    size_t index,
    const matlab::engine::String &propertyName,
    const matlab::data::Array &propertyValue)

FutureResult<void> setPropertyAsync(matlab::data::Array &object,
 const matlab::engine::String &propertyName,
 const matlab::data::Array &propertyValue)
```

**Description**

Set the value of an object property asynchronously. If the object input argument is an array of objects, specify the index of the array element that corresponds to the object whose property value you want to set.

**Parameters**

| | |
|---|---|
| `matlab::data::Array &objectArray` | Array of MATLAB objects |
| `matlab::data::Array &object` | Scalar MATLAB object |
| `size_t index` | Zero-based index into the object array, specifying the object in that array whose property value is set |
| `const matlab::engine::String &propertyName` | Name of the property to set. Specify the name as an `std::u16string`. Also, you can specify this parameter as an `std::string`. |
| `const matlab::data::Array &propertyValue` | Value assigned to the property. |

#### Exceptions

None

#### Examples

This example shows how to set a MATLAB object property asynchronously. It creates a MATLAB graph and returns the line handle object. Setting the line `LineStyle` property to the character `:` changes the property value of the object in MATLAB and updates the line style of the graph.

```
#include "MatlabEngine.hpp"
using namespace matlab::engine;

std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
matlab::data::ArrayFactory factory;
matlab::data::Array yData = factory.createArray<double>({ 1, 5 }, { 4.0, 11.0, 4.7, 36.2, 72.3 });
matlab::data::Array lineHandle = matlabPtr->feval(u"plot", yData);
matlab::data::CharArray lineStyle = factory.createCharArray(":");
FutureResult<void> future = matlabPtr->setPropertyAsync(lineHandle, u"LineStyle", lineStyle);
```

#### Related Topics

"Set Property on MATLAB Object"

## See Also

### Introduced in R2017b

# matlab::engine::connectMATLAB

Connect to shared MATLAB session synchronously

## Description

```
std::unique_ptr<MATLABEngine> connectMATLAB()
```

```
std::unique_ptr<MATLABEngine> connectMATLAB(const matlab::engine::String&
name)
```

Connect synchronously to a shared MATLAB session on the local machine.

- If you specify the name of a shared MATLAB session, but the engine cannot find a session with that name, the engine throws an exception.
- If you do not specify a name and there is no shared MATLAB session available, the engine starts a new shared MATLAB session. The MATLAB desktop is not started.
- If you do not specify a name and there are shared MATLAB sessions available, the engine connects to the first available session.

### Include

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include | `MatlabEngine.hpp` |

## Parameters

| | |
|---|---|
| `const matlab::engine::String& name` | Name of the shared MATLAB session |

## Return Value

| | |
|---|---|
| `std::unique_ptr<MATLABEngine>` | Pointer to a `MATLABEngine` object |

## Exceptions

| | |
|---|---|
| `matlab::engine::EngineException` | Throws exception if function fails to connect to the specified MATLAB session. |

## Examples

### Connect to Shared MATLAB Session

Connect to a shared MATLAB session named `my_matlab`.

```
std::unique_ptr<MATLABEngine> matlabPrt = connectMATLAB(u"my_matlab");
```

## See Also

`matlab::engine::connectMATLABAsync`

**Topics**

"Start MATLAB Sessions from C++"

**Introduced in R2017b**

# matlab::engine::connectMATLABAsync

Connect to shared MATLAB session asynchronously

## Description

`FutureResult<std::unique_ptr<MATLABEngine>> connectMATLABAsync()`

`FutureResult<std::unique_ptr<MATLABEngine>> connectMATLABAsync(const matlab::engine::String& name)`

Connect asynchronously to a shared MATLAB session on the local machine.

- If you specify the name of a shared MATLAB session, but the engine cannot find a session with that name, the engine throws an exception.
- If you do not specify a name and there is no shared MATLAB session available, the engine starts a new shared MATLAB session. The MATLAB desktop is not started.
- If you do not specify a name and there are shared MATLAB sessions available, the engine connects to the first available session.

### Include

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include | `MatlabEngine.hpp` |

## Parameters

| | |
|---|---|
| `const matlab::engine::String& name` | Name of the shared MATLAB session |

## Return Value

| | |
|---|---|
| `FutureResult<std::unique_ptr<MATLABEngine>>` | A `FutureResult` object that you can use to get the pointer to the `MATLABEngine` |

## Examples

### Connect to Shared MATLAB Session Asynchronously

Connect to a shared MATLAB session named `my_matlab` asynchronously. Use the `FutureResult` `get` method to retrieve the pointer to the `MATLABEngine` object.

```
#include "MatlabEngine.hpp"
void asyncConnect() {
    using namespace matlab::engine;
```

```
    // Find and connect to shared MATLAB session
    FutureResult<std::unique_ptr<MATLABEngine>> future = connectMATLABAsync(u"my_matlab");
    ...
    std::unique_ptr<MATLABEngine> matlabPtr = future.get();
}
```

## See Also

`matlab::engine::connectMATLAB`

### Topics

"Connect C++ to Running MATLAB Session"

**Introduced in R2017b**

# matlab::engine::convertUTF8StringToUTF16String

Convert UTF-8 string to UTF-16 string

## Description

`std::basic_string<char16_t> convertUTF8StringToUTF16String(const std::string& utf8string)`

Convert a UTF-8 string to a UTF-16 string. Use this function to convert ASCII strings to `matlab::engine::String` strings, which are used by MATLAB C++ Engine functions.

### Include

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include | `MatlabEngine.hpp` |

## Parameters

| | |
|---|---|
| `const std::string& utf8string` | A UTF-8 string |

## Return Value

| | |
|---|---|
| `std::basic_string<char16_t>` | A UTF-16 string |

## Exceptions

| | |
|---|---|
| `matlab::engine::OutofMemoryException` | The function failed to allocate memory. |
| `matlab::engine::TypeConversionException` | The input type cannot be converted to `std::basic_string<char16_t>`. |

## Examples

### Convert String

Convert a UTF-8 string to a `matlab::engine::String` (UTF-16 string).

```
std::u16string matlabStatement = convertUTF8StringToUTF16String("sRoot = sqrt(12.7);");
```

## Alternative Conversion

If you are using a C++ compiler that supports the use of the "u" prefix to create UTF-16 encoded string literals, you can use this approach to create inputs for engine functions. For example, this code defines a variable that contains a MATLAB statement as a UTF-16 string.

```
std::u16string matlabStatement = u"sRoot = sqrt(12.7);";
```

For an up-to-date list of supported compilers, see the Supported and Compatible Compilers website.

## See Also

`matlab::engine::String` | `matlab::engine::convertUTF16StringToUTF8String`

**Introduced in R2017b**

# matlab::engine::convertUTF16StringToUTF8String

Convert UTF-16 string to UTF-8 string

## Description

`std::string convertUTF16StringToUTF8String(const std::basic_string<char16_t>& utf16string)`

Convert a UTF-16 string to a UTF-8 string.

### Include

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include | `MatlabEngine.hpp` |

## Parameters

| | |
|---|---|
| `const std::basic_string<char16_t>& utf16string` | A UTF-16 string |

## Return Value

| | |
|---|---|
| `std::string` | A UTF-8 string |

## Exceptions

| | |
|---|---|
| `matlab::engine::OutofMemoryException` | The function failed to allocate memory. |
| `matlab::engine::TypeConversionException` | The input type cannot be converted to `std::string`. |

## Examples

### Convert String

Convert a `matlab::engine::String` (UTF-16 string) to a `std::string` (UTF-8 string).

```
matlab::engine::String matlabStatement = (u"sqrt(12.7);");
std::string cmdString = convertUTF16StringToUTF8String(matlabStatement);
```

## See Also

`matlab::engine::String` | `matlab::engine::convertUTF8StringToUTF16String`

**Introduced in R2017b**

# matlab::engine::findMATLAB

Find shared MATLAB sessions synchronously

## Description

`std::vector<String> findMATLAB()`

Find all shared MATLAB sessions on the local machine.

### Include

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include | `MatlabEngine.hpp` |

## Parameters

None

## Return Value

| | |
|---|---|
| `std::vector<String>` | A vector of the names of all shared MATLAB sessions on the local machine, or an empty vector if no shared MATLAB sessions are available |

## Exceptions

| | |
|---|---|
| `matlab::engine::EngineException` | Throws exception if the call fails while searching for shared MATLAB sessions. |

## Examples

### Find Shared MATLAB Session Synchronously

```
std::vector<String> names = findMATLAB();
```

## See Also

`matlab::engine::findMATLABAsync`

**Introduced in R2017b**

# matlab::engine::findMATLABAsync

Find shared MATLAB sessions asynchronously

## Description

`FutureResult<std::vector<String>> findMATLABAsync()`

Find all shared MATLAB sessions on the local machine asynchronously.

### Include

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include | `MatlabEngine.hpp` |

## Parameters

None

## Return Value

| | |
|---|---|
| `FutureResult<std::vector<String>>` | A `FutureResult` object that you can use to get the names of shared MATLAB sessions on the local machine. |

## Examples

### Find Shared MATLAB Session Asynchronously

Find the names of all shared MATLAB sessions on the local machine asynchronously. Use the `FutureResult` `get` method to retrieve the names.

```
FutureResult<std::vector<String>> futureNames = findMATLABAsync();
...
std::vector<String> matlabSessions = futureNames.get();
```

## See Also

`matlab::engine::findMATLAB`

**Introduced in R2017b**

# matlab::engine::FutureResult

Retrieve result from asynchronous operation

## Description

A future result is an object that you can use to retrieve the result of MATLAB functions or statements. The `FutureResult` class provides all member functions of the C++ `std::future` class.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include | `MatlabEngine.hpp` |

## Constructor Summary

Create a `FutureResult` object using these asynchronous functions:

- Asynchronous member functions defined by `matlab::engine::MATLABEngine`.
- `matlab::engine::startMATLABAsync`, `matlab::engine::connectMATLABAsync`, and `matlab::engine::findMATLABAsync`.

## Method Summary

### Member Functions

| | |
|---|---|
| "cancel" on page 1-134 | Cancel the operation held by the `FutureResult` object. |

| Member Functions Delegated to std::future |
|---|
| `operator=`, `share`, `get`, `wait`, `wait_for`, `wait_until` |

#### Exceptions Thrown by `get` Method

| | |
|---|---|
| `matlab::engine::EngineException` | Cannot start or connect to MATLAB session. |
| `matlab::engine::CancelException` | Execution of command is canceled. |
| `matlab::engine::InterruptedException` | Evaluation of command is interrupted. |
| `matlab::engine::MATLABNotAvailableException` | The MATLAB session is not available. |
| `matlab::engine::MATLABSyntaxException` | There is a syntax error in the MATLAB function. |
| `matlab::engine::MATLABExecutionException` | MATLAB runtime error in the function. |

| | |
|---|---|
| `matlab::engine::TypeConversionException` | The result from a MATLAB function cannot be converted to the specified type. |

## Method Details

### `cancel`

```
bool FutureResult::cancel(bool allowInterrupt = true);
```

#### Description

Cancel the evaluation of the MATLAB function or statement. You cannot cancel asynchronous operations that use: `matlab::engine::startMATLABAsync`, `matlab::engine::connectMATLABAsync`, or `matlab::engine::findMATLABAsync`.

#### Parameters

| | |
|---|---|
| `bool allowInterrupt` | If false, do not interrupt if execution had already begun. |

#### Returns

| | |
|---|---|
| `bool` | Was command canceled if execution had already begun. |

#### Example

```
bool flag = future.cancel();
```

#### Exception Safety

No exceptions thrown

## See Also

**Topics**
"Call Function Asynchronously"

**Introduced in R2017b**

# matlab::engine::startMATLAB

Start MATLAB synchronously

## Description

```
std::unique_ptr<MATLABEngine> startMATLAB(const std::vector<String>& options
= std::vector<String>())
```

Start MATLAB synchronously in a separate process with optional MATLAB startup options.

### Include

| | |
|---|---|
| Namespace: | matlab::engine |
| Include | MatlabEngine.hpp |

## Parameters

| | |
|---|---|
| const std::vector<String>& options | Options used to start MATLAB. You can specify multiple startup options. The engine supports all MATLAB startup options, except for the options listed in “Unsupported Startup Options” on page 1-108. For a list of options, see the platform-specific command matlab (Windows), matlab (macOS), or matlab (Linux). |

## Return Value

| | |
|---|---|
| std::unique_ptr<MATLABEngine> | Pointer to the MATLABEngine object |

## Exceptions

| | |
|---|---|
| matlab::engine::EngineException | MATLAB failed to start. |

## Examples

### Start MATLAB Synchronously

Start MATLAB synchronously and return a unique pointer to the MATLABEngine object.

```
std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
```

#### Start MATLAB with Desktop

```
auto matlabApplication = matlab::engine::startMATLAB({u"-desktop"});
```

#### Start MATLAB with Options

Start MATLAB with the `-nojvm` option and return a unique pointer to the `MATLABEngine` object.

```
std::vector<String> optionVec;
optionVec.push_back(u"-nojvm");
std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB(optionVec);
```

## See Also

`matlab::engine::MATLABEngine` | `matlab::engine::startMATLABAsync`

**Topics**

"Start MATLAB Sessions from C++"

**Introduced in R2017b**

# matlab::engine::startMATLABAsync

Start MATLAB asynchronously

## Description

```
FutureResult<std::unique_ptr<MATLABEngine>> startMATLABAsync(const
std::vector<String>& options = std::vector<String>())
```

Start MATLAB asynchronously in a separate process with optional MATLAB startup options.

### Include

| | |
|---|---|
| Namespace: | matlab::engine |
| Include | MatlabEngine.hpp |

## Parameters

| | |
|---|---|
| const std::vector<String>& options | Startup options used to launch MATLAB. You can specify multiple startup options. The engine supports all MATLAB startup options, except for the options listed in “Unsupported Startup Options” on page 1-108. For a list of options, see the platform-specific command matlab (Windows), matlab (macOS), or matlab (Linux). |

## Return Value

| | |
|---|---|
| FutureResult<std::unique_ptr<MATLABEngine>> | A FutureResult object used to get the pointer to the MATLABEngine |

## Examples

### Start MATLAB Asynchronously

Start MATLAB asynchronously and return a FutureResult object. Use the FutureResult to get a pointer to the MATLABEngine object.

```
FutureResult<std::unique_ptr<MATLABEngine>> matlabFuture = startMATLABAsync();
...
std::unique_ptr<MATLABEngine> matlabPtr = matlabFuture.get();
```

## See Also

matlab::engine::startMATLAB

### Topics

“Specify Startup Options”

**Introduced in R2017b**

# matlab::engine::terminateEngineClient

Free engine resources during runtime

## Description

`void matlab::engine::terminateEngineClient()`

Release all MATLAB engine resources during runtime when you no longer need the MATLAB engine in your application program.

**Note** Programs cannot start a new MATLAB engine or connect to a shared MATLAB session after calling `terminateEngineClient`.

### Include

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include | `MatlabEngine.hpp` |

## Examples

Terminate the engine session to free resources.

```
// Start MATLAB session
std::unique_ptr<MATLABEngine> matlabPtr = startMATLAB();
...
// Terminate MATLAB session
matlab::engine::terminateEngineClient();
```

## See Also

`matlab::engine::startMATLAB`

**Introduced in R2017b**

# matlab::engine::WorkspaceType

Type of MATLAB workspace

## Description

The `matlab::engine::WorkspaceType` enum class specifies the MATLAB workspace to pass variables to or get variables from.

| | |
|---|---|
| `BASE` | Variables scoped to the MATLAB base workspace (command line and nonfunction scripts) |
| `GLOBAL` | Variables scoped to the MATLAB global workspace (command line, functions, and scripts) |

MATLAB scopes variables by workspace. Variables that are scoped to the base workspace must be passed to functions as arguments. Variables scoped to the global workspace can be accessed by any function that defines the specific variable name as global.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include | `MatlabEngine.hpp` |

## Examples

This example:

- Connects to a shared MATLAB session
- Creates a `matlab::data::Array` containing numeric values of type double
- Puts the array in the MATLAB global workspace

```
#include "MatlabDataArray.hpp"
#include "MatlabEngine.hpp"
#include <iostream>

static void putGlobalVar() {

    using namespace matlab::engine;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(u"myMatlabEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    // Create matlab data array factory
    matlab::data::ArrayFactory factory;

    // Create data variable
    matlab::data::Array data = factory.createArray<double>
        ({ 1, 5 }, { 4.0, 11.0, 4.7, 36.2, 72.3 });

    // Put data variable in MATLAB global workspace
    matlabPtr->setVariable(u"data", data, WorkspaceType::GLOBAL);
}
```

## See Also

`matlab::data::ArrayFactory` | `matlab::engine::MATLABEngine`

**Topics**
"Pass Variables from C++ to MATLAB"
"Pass Variables from MATLAB to C++"

**Introduced in R2017b**

# matlab::engine::String

Define UTF16 string

## Description

Type definition for `std::basic_string<char16_t>`.

## Examples

This example defines a variable containing the name of a shared MATLAB session. Pass this string to the `matlab::engine::connectMATLAB` function to connect to the named session.

```
matlab::engine::String session(u"myMatlabEngine");
std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);
```

## See Also

`matlab::engine::convertUTF16StringToUTF8String` | `matlab::engine::convertUTF8StringToUTF16String`

### Topics

"MATLAB Engine API for C++"
"Connect C++ to Running MATLAB Session"

**Introduced in R2017b**

# matlab::engine::StreamBuffer

Define stream buffer

## Description

Type definition for `std::basic_streambuf<char16_t>`.

## Examples

This example defines string buffers to return output from the evaluation of a MATLAB function by the `MATLABEngine::eval` member function. This function uses a buffer derived from `matlab::engine::StreamBuffer` to return output from MATLAB to C++.

```
#include "MatlabEngine.hpp"
#include "MatlabDataArray.hpp"
#include <iostream>

using namespace matlab::engine;
using SBuf = std::basic_stringbuf<char16_t>;

void printFromBuf(const std::shared_ptr<SBuf> buf)
{
    //Get text from buf
    auto text_ = buf->str();
    std::cout << "*" << convertUTF16StringToUTF8String(text_)
        << "*" << std::endl;
}

int main() {

    //Create Array factory
    matlab::data::ArrayFactory factory;

    // Connect to named shared MATLAB session started as:
    // matlab -r "matlab.engine.shareEngine('myMatlabEngine')"
    String session(u"myMatlabEngine");
    std::unique_ptr<MATLABEngine> matlabPtr = connectMATLAB(session);

    auto outBuf = std::make_shared<SBuf>();
    auto errBuf = std::make_shared<SBuf>();

    matlabPtr->eval(u"matlab.engine.engineName", outBuf, errBuf);
    printFromBuf(outBuf);
    printFromBuf(errBuf);
    return 0;
}
```

## See Also

`matlab::engine::connectMATLAB` | `matlab::engine::convertUTF16StringToUTF8String`

### Topics

"Redirect MATLAB Command Window Output to C++"

**Introduced in R2017b**

# matlab::engine::SharedFutureResult

Retrieve result from asynchronous operation as shared future

## Description

A shared future result is an object that you use to retrieve the result of MATLAB functions or statements any number of times.

### Class Details

| | |
|---|---|
| Namespace: | `matlab::engine` |
| Include | `MatlabEngine.hpp` |

## Constructor Summary

Create a `FutureResult` object using one of these asynchronous functions:

- Asynchronous member functions defined by `matlab::engine::MATLABEngine`.
- `matlab::engine::startMATLABAsync`, `matlab::engine::connectMATLABAsync`, and `matlab::engine::findMATLABAsync`.

## Method Summary

### Member Functions

| | |
|---|---|
| "cancel" on page 1-134 | Cancel the operation held by the `FutureResult` object. |

| Member Function Delegated to std::shared_future |
|---|
| `operator=`, `get`, `valid`, `wait`, `wait_for`, `wait_until` |

#### Exceptions Thrown by get Method

| | |
|---|---|
| `matlab::engine::EngineException` | Cannot start or connect to MATLAB session. |
| `matlab::engine::CancelException` | Execution of command is canceled. |
| `matlab::engine::InterruptedException` | Evaluation of command is interrupted. |
| `matlab::engine::MATLABNotAvailableException` | The MATLAB session is not available. |
| `matlab::engine::MATLABSyntaxException` | There is a syntax error in the MATLAB function. |
| `matlab::engine::MATLABExecutionException` | MATLAB runtime error in the function. |

| | |
|---|---|
| `matlab::engine::TypeConversionException` | The result from a MATLAB function cannot be converted to the specified type. |

## Method Details

### `cancel`

```
bool FutureResult::cancel(bool allowInterrupt = true);
```

#### Description

Cancel the evaluation of the MATLAB function or statement.

Note that you cannot cancel asynchronous start, connection, or find operations, which are initiated using these functions: `matlab::engine::startMATLABAsync`, `matlab::engine::connectMATLABAsync`, or `matlab::engine::findMATLABAsync`.

#### Parameters

| | |
|---|---|
| `bool allowInterrupt` | If false, do not interrupt if execution has already begun. |

#### Return Value

| | |
|---|---|
| `bool` | True if the MATLAB command can be canceled |

#### Examples

```
bool flag = future.cancel();
```

#### Exceptions

None

## See Also

`matlab::engine::FutureResult`

**Topics**

"Call Function Asynchronously"

**Introduced in R2017b**

# com.mathworks.engine.MatlabEngine

Java class using MATLAB as a computational engine

## Description

The `com.mathworks.engine.MatlabEngine` class uses a MATLAB process as a computational engine for Java®. This class provides an interface between the Java language and MATLAB, enabling you to evaluate MATLAB functions and expressions from Java.

## Creation

The `MatlabEngine` class provides static methods to start MATLAB and to connect to a shared MATLAB session synchronously or asynchronously. Only these static methods can instantiate this class:

- Start MATLAB synchronously — “startMatlab” on page 1-147
- Connect to shared MATLAB session synchronously — “connectMatlab” on page 1-149
- Start MATLAB asynchronously — “startMatlabAsync” on page 1-148
- Connect to shared MATLAB session asynchronously — “connectMatlabAsync” on page 1-150

### Unsupported Startup Options

The engine does not support these MATLAB startup options:

- `-h`
- `-help`
- `-?`
- `-n`
- `-e`
- `-softwareopengl`
- `-logfile`

For information on MATLAB startup options, see “Commonly Used Startup Options”.

## Method Summary

### Static Methods

| | |
|---|---|
| “startMatlab” on page 1-147 | Start MATLAB synchronously. |
| “startMatlabAsync” on page 1-148 | Start MATLAB asynchronously. |
| “findMatlab” on page 1-148 | Find all available shared MATLAB sessions from a local machine synchronously. |

| | |
|---|---|
| "findMatlabAsync" on page 1-149 | Find all available shared MATLAB sessions from a local machine asynchronously. |
| "connectMatlab" on page 1-149 | Connect to a shared MATLAB session on a local machine synchronously. |
| "connectMatlabAsync" on page 1-150 | Connect to a shared MATLAB session on a local machine asynchronously. |

#### Member Variable

| | |
|---|---|
| NULL_WRITER | Use a writer that ignores the contents from the MATLAB command window. |

#### Member Functions

| | |
|---|---|
| "feval" on page 1-150 | Evaluate a MATLAB function with arguments synchronously. |
| "fevalAsync" on page 1-152 | Evaluate a MATLAB function with arguments asynchronously. |
| "eval" on page 1-153 | Evaluate a MATLAB expression as a string synchronously. |
| "evalAsync" on page 1-153 | Evaluate a MATLAB expression as a string asynchronously. |
| "getVariable" on page 1-154 | Get a variable from the MATLAB base workspace synchronously. |
| "getVariableAsync" on page 1-155 | Get a variable from the MATLAB base workspace asynchronously. |
| "putVariable" on page 1-155 | Put a variable into the MATLAB base workspace synchronously. |
| "putVariableAsync" on page 1-156 | Put a variable into the MATLAB base workspace asynchronously. |
| "disconnect" on page 1-156 | Disconnect from the current MATLAB session synchronously. |
| "disconnectAsync" on page 1-157 | Disconnect from the current MATLAB session asynchronously. |
| "quit" on page 1-157 | Force the shutdown of the current MATLAB session synchronously. |
| "quitAsync" on page 1-157 | Force the shutdown of the current MATLAB session asynchronously. |
| "close" on page 1-158 | Disconnect or terminate the current MATLAB session. |

## Method Details

### startMatlab

```
static MatlabEngine startMatlab(String[] options)
```

```
static MatlabEngine startMatlab()
```

#### Description

Start MATLAB synchronously.

**Parameters**

| | |
|---|---|
| String[] options | Startup options used to start MATLAB engine. You can specify multiple startup options. The engine supports all MATLAB startup options, except for the options listed in "Unsupported Startup Options" on page 1-146. For a list of options, see the platform-specific command matlab (Windows), matlab (macOS), or matlab (Linux). |

**Returns**

Instance of MatlabEngine

**Throws**

| | |
|---|---|
| com.mathworks.engine .EngineException | MATLAB fails to start. |

**Example**

```
String[] options = {"-noFigureWindows", "-r", "cd H:"};
MatlabEngine eng = MatlabEngine.startMatlab(options);
```

**See Also**

"Start and Close MATLAB Session from Java"

### startMatlabAsync

```
static Future<MatlabEngine> startMatlabAsync(String[] options)
```

```
static Future<MatlabEngine> startMatlabAsync()
```

**Description**

Start MATLAB asynchronously. Once MATLAB has started, then cancel is a no-op.

**Parameters**

| | |
|---|---|
| String[] options | Startup options used to start MATLAB engine. You can specify multiple startup options. The engine supports all MATLAB startup options, except for the options listed in "Unsupported Startup Options" on page 1-146. For a list of options, see the platform-specific command matlab (Windows), matlab (macOS), or matlab (Linux). |

**Returns**

Instance of Future<MatlabEngine>

**Example**

```
Future<MatlabEngine> future = MatlabEngine.startMatlabAsync();
```

**See Also**

"Start and Close MATLAB Session from Java"

### findMatlab

```
static String[] findMatlab()
```

#### Description

Find all shared MATLAB sessions on the local machine synchronously.

#### Returns

An array of the names of all shared MATLAB sessions on the local machine, or an empty vector if there are no shared MATLAB sessions available on the local machine.

#### Throws

| | |
|---|---|
| `com.mathworks.engine.EngineException` | If there is a failure during the search for MATLAB sessions. |

#### Example

```
String[] engines = MatlabEngine.findMatlab();
```

#### See Also

"Connect Java to Running MATLAB Session"

### findMatlabAsync

```
static Future<String[]> findMatlabAsync()
```

#### Description

Find all shared MATLAB sessions on local machine asynchronously.

#### Returns

An instance of `Future<String[]>`

#### Example

```
Future<String[]> future = MatlabEngine.findMatlabAsync();
```

#### See Also

"Connect Java to Running MATLAB Session"

### connectMatlab

```
static MatlabEngine connectMatlab(String name)
```

```
static MatlabEngine connectMatlab()
```

#### Description

Connect to a shared MATLAB session on local machine synchronously.

- If you specify the name of a shared MATLAB session, but the engine cannot find a session with that name, the engine throws an exception.
- If you do not specify a name and there is no shared MATLAB session available, the engine starts a new shared MATLAB session with default options.
- If you do not specify a name and there are shared MATLAB sessions available, the engine connects to the first available session.

**Parameters**

| | |
|---|---|
| `String name` | Name of the shared MATLAB session. Use “findMatlab” on page 1-148 to get the names of shared MATLAB sessions. |

**Returns**

An instance of `MatlabEngine`

**Throws**

| | |
|---|---|
| `com.mathworks.engine.EngineException` | MATLAB fails to start or connect. |

**Example**

```
MatlabEngine engine = MatlabEngine.connectMatlab();
```

**See Also**

“Connect Java to Running MATLAB Session”

### `connectMatlabAsync`

`static Future<MatlabEngine> connectMatlabAsync(String name)`

`static Future<MatlabEngine> connectMatlabAsync`

**Description**

Connect to a shared MATLAB session on local machine asynchronously. The behavior is the same as that of `connectMatlab` except the mechanism is asynchronous. Once a connection has been made to MATLAB, then cancel is a no-op.

**Parameters**

| | |
|---|---|
| `String name` | Name of the shared MATLAB session. |

**Returns**

An instance of `Future<MatlabEngine>`

**Example**

```
Future<MatlabEngine> future = MatlabEngine.connectMatlabAsync();
```

**See Also**

“Connect Java to Running MATLAB Session”

### `feval`

`<T> T feval(int nlhs, String func, Writer output, Writer error, Object… args)`

`<T> T feval(int nlhs, String func, Object… args)`

`<T> T feval(String func, Writer output, Writer error, Object… args)`

`<T> T feval(String func, Object… args)`

**Description**

Evaluate MATLAB functions with input arguments synchronously.

**Parameters**

| | |
|---|---|
| String func | Name of the MATLAB function or script to evaluate. |
| int nlhs | Number of expected outputs. Default is 1.<br><br>If nlhs is greater than 1, the returned type T must be <Object[]>.<br><br>If nlhs is 0, the returned type T must be <Void> or <?>.<br><br>If nlhs is 1, the returned type T can be the expected type or <Object> if the type is not known. |
| Writer output | Stream used to store the standard output from the MATLAB function. If you do not specify a writer, the output is written to the command window or terminal. Use NULL_WRITER to ignore the output from the MATLAB command window. |
| Writer error | Stream used to store the standard error from the MATLAB function. If you do not specify a writer, the output is written to the command window or terminal. Use NULL_WRITER to ignore the error message from the MATLAB command window. |
| Object... args | Arguments to pass to the MATLAB function. |

**Returns**

Result of executing the MATLAB function

**Throws**

| | |
|---|---|
| java.util.concurrent.CancellationException | Evaluation of a MATLAB function was canceled. |
| java.lang.InterruptedException | Evaluation of a MATLAB function was interrupted. |
| java.lang.IllegalStateException | The MATLAB session is not available. |
| com.mathworks.engine.MatlabExcecutionException | There is a MATLAB runtime error in the function. |
| com.mathworks.engine.UnsupportedTypeExeption | There is an unsupported data type. |
| com.mathworks.engine.MatlabSyntaxException | There is a syntax error in the MATLAB function. |

**Example**

```
double result = engine.feval("sqrt", 4);
```

**See Also**

"Execute MATLAB Functions from Java"

### fevalAsync

```
<T> Future<T> fevalAsync(int nlhs, String func, Writer output, Writer error,
Object… args)

<T> Future<T> fevalAsync(int nlhs, String func, Object… args)

<T> Future<T> fevalAsync(String func, Writer output, Writer error, Object…
args)

<T> Future<T> fevalAsync(String func, Object… args)
```

#### Description

Evaluate MATLAB functions with input arguments asynchronously.

#### Parameters

| | |
|---|---|
| String func | Name of the MATLAB function or script to evaluate. |
| int nlhs | Number of expected outputs. Default is 1.<br><br>If nlhs is greater than 1, the returned type T must be <Object[]>.<br><br>If nlhs is 0, the returned type T must be <Void> or <?>.<br><br>If nlhs is 1, the returned type T can be the expected type or <Object> if the type is not known. |
| Writer output | Stream used to store the standard output from the MATLAB function. If you do not specify a writer, the output is written to the command window or terminal. Use NULL_WRITER to ignore the output from the MATLAB command window. |
| Writer error | Stream used to store the standard error from the MATLAB function. If you do not specify a writer, the output is written to the command window or terminal. Use NULL_WRITER to ignore the error message from the MATLAB command window. |
| Object... args | Arguments to pass to the MATLAB function. |

#### Returns

An instance of Future<T>

#### Throws

| | |
|---|---|
| java.lang.IllegalStateException | The MATLAB session is not available. |

#### Example

```
Future<Double> future = engine.fevalAsync("sqrt", 4);
```

#### See Also

"Execute MATLAB Functions from Java"

### eval

`void eval(String command, Writer output, Writer error)`

`void eval(String command)`

#### Description

Evaluate a MATLAB statement as a string synchronously.

#### Parameters

| | |
|---|---|
| `String command` | MATLAB statement to evaluate. |
| `Writer output` | Stream used to store the standard output from the MATLAB statement. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the output from the MATLAB command window. |
| `Writer error` | Stream used to store the standard error from the MATLAB statement. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the error message from the MATLAB command window. |

#### Throws

| | |
|---|---|
| `java.util.concurrent.CancellationException` | Evaluation of a MATLAB function was canceled. |
| `java.lang.InterruptedException` | Evaluation of a MATLAB function was interrupted. |
| `java.lang.IllegalStateException` | The MATLAB session is not available. |
| `com.mathworks.engine.MatlabExcecutionException` | There is an error in the MATLAB statement during runtime. |
| `com.mathworks.engine.MatlabSyntaxException` | There is a syntax error in the MATLAB statement. |

#### Example

```
engine.eval("result = sqrt(4)");
```

#### See Also

“Evaluate MATLAB Statements from Java”

### evalAsync

`Future<Void> evalAsync(String command, Writer output, Writer error)`

`Future<Void> evalAsync(String command)`

#### Description

Evaluate a MATLAB statement as a string asynchronously.

**Parameters**

| | |
|---|---|
| `String command` | MATLAB statement to evaluate. |
| `Writer output` | Stream used to store the standard output from the MATLAB statement. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the output from the MATLAB command window. |
| `Writer error` | Stream used to store the standard error from the MATLAB statement. If you do not specify a writer, the output is written to the command window or terminal. Use `NULL_WRITER` to ignore the error message from the MATLAB command window. |

**Returns**

An instance of `Future<Void>`

**Throws**

| | |
|---|---|
| `java.lang.IllegalStateException` | The MATLAB session is not available. |

**Example**

```
Future<Void> future = engine.evalAsync("sqrt(4)");
```

**See Also**

"Evaluate MATLAB Statements from Java"

### getVariable

`<T> T getVariable(String varName)`

**Description**

Get a variable from the MATLAB base workspace.

**Parameters**

| | |
|---|---|
| `String varName` | Name of a variable in the MATLAB base workspace. |

**Returns**

Variable passed from the MATLAB base workspace

**Throws**

| | |
|---|---|
| `java.util.concurrent.CancellationException` | Evaluation of this function is canceled. |
| `java.lang.InterruptedException` | Evaluation of this function is interrupted. |
| `java.lang.IllegalStateException` | The MATLAB session is not available. |

**Example**

```
double myVar = engine.getVariable("myVar");
```

**See Also**

"Pass Variables from MATLAB to Java"

### getVariableAsync

```
<T> Future<T> getVariableAsync(String varName)
```

**Description**

Get a variable from the MATLAB base workspace asynchronously.

**Parameters**

| | |
|---|---|
| `String varName` | Name of a variable in MATLAB base workspace. |

**Returns**

An instance of `Future<T>`

**Throws**

| | |
|---|---|
| `java.lang.IllegalStateException` | The MATLAB session is not available. |

**Example**

```
Future<Double> future = engine.getVariableAsync("myVar");
```

**See Also**

"Pass Variables from MATLAB to Java"

### putVariable

```
void putVariable(String varName, T varData)
```

**Description**

Put a variable into the MATLAB base workspace.

**Parameters**

| | |
|---|---|
| `String varName` | Name of a variable to create in the MATLAB base workspace. |
| `T varData` | Value of the variable to create in the MATLAB base workspace. |

**Throws**

| | |
|---|---|
| `java.util.concurrent.CancellationException` | Evaluation of this function is canceled. |
| `java.lang.InterruptedException` | Evaluation of this function is interrupted. |

| | |
|---|---|
| `java.lang.IllegalStateException` | The MATLAB session is not available. |

#### Example

```
engine.putVariable("myVar", 100);
```

#### See Also

"Pass Variables from Java to MATLAB"

### putVariableAsync

`Future<Void> putVariableAsync(String varName, T varData)`

#### Description

Put a variable into the MATLAB base workspace asynchronously.

#### Parameters

| | |
|---|---|
| `String varName` | Name of a variable to create in the MATLAB base workspace. |
| `T varData` | Value of the variable to create in the MATLAB base workspace. |

#### Returns

An instance of `Future<Void>`

#### Throws

| | |
|---|---|
| `java.lang.IllegalStateException` | The MATLAB session is not available. |

#### Example

```
Future<Void> future = engine.putVariableAsync("myVar", 100);
```

#### See Also

"Pass Variables from Java to MATLAB"

### disconnect

`void disconnect()`

#### Description

Disconnect from the current MATLAB session.

#### Throws

| | |
|---|---|
| `com.mathworks.engine.EngineException` | The current MATLAB session cannot be disconnected. |

#### Example

```
engine.disconnect();
```

#### See Also

"Close MATLAB Engine Session"

### disconnectAsync

```
Future<Void> disconnectAsync()
```

#### Description

Disconnect from the current MATLAB session.

#### Example

```
Future<Void> future = engine.disconnectAsync();
```

#### See Also

"Close MATLAB Engine Session"

### quit

```
void quit()
```

#### Description

Force the shutdown of the current MATLAB session.

#### Throws

| | |
|---|---|
| com.mathworks.engine.EngineException | The current MATLAB session cannot be shut down. |

#### Example

```
engine.quit();
```

#### See Also

"Close MATLAB Engine Session"

### quitAsync

```
Future<Void> quitAsync()
```

#### Description

Force the shutdown of the current MATLAB session asynchronously without waiting for termination.

#### Returns

An instance of Future<Void>

#### Example

```
Future<Void> future = engine.quitAsync();
```

#### See Also

"Close MATLAB Engine Session"

### close

`void close()`

#### Description

`MatlabEngine` provides the `close()` method to implement the `java.lang.AutoCloseable` interface for `MatlabEngine` objects. This `close()` method enables you to use a `try`-with-resources statement to automatically disconnect or terminate the MATLAB session at the end of the statement.

The `MatlabEngine` `close()` method disconnects or terminates the current MATLAB session, depending on the context.

- If a Java process starts the MATLAB session as a default non-shared session, `close()` terminates MATLAB.
- If the MATLAB session is a shared session, `close()` disconnects MATLAB from this Java process. MATLAB terminates when there are no other connections.

To force the shutdown or disconnection of the current MATLAB session, explicitly call `MatlabEngine.quit()`, `MatlabEngine.disconnect()`, or their asynchronous counterparts.

#### Example

```
engine.close();
```

#### See Also

"Close MATLAB Engine Session"

## See Also

`matlab.engine.engineName` | `matlab.engine.isEngineShared` | `matlab.engine.shareEngine`

#### Topics

"Build Java Engine Programs"
"Start and Close MATLAB Session from Java"
"Specify Startup Options"

**Introduced in R2016b**

# com.mathworks.matlab.types.Complex

Java class to pass complex data to and from MATLAB

## Description

The `Complex` class provides Java support for MATLAB complex arrays. Use this class to pass complex data to MATLAB. The MATLAB engine passes complex data to Java as an instance of `Complex`.

All MATLAB numeric types are converted to `double` in Java.

### Field Summary

| | |
|---|---|
| `double real` | The real part of the complex data |
| `double imag` | The imaginary part of the complex data |

## Creation

`Complex(double real, double imag)` constructs an instance of `Complex` with the specified real and imaginary values.

## Examples

### Pass Complex Variable to MATLAB Function

```
import com.mathworks.engine.MatlabEngine

MatlabEngine engine = MatlabEngine.startMatlab();
Complex c = new Complex(2,3);
Complex cj = engine.feval("conj",c);
```

## See Also

`com.mathworks.matlab.types.CellStr` | `com.mathworks.matlab.types.HandleObject` | `com.mathworks.matlab.types.Struct`

### Topics

"Using Complex Variables in Java"

**Introduced in R2016b**

# com.mathworks.matlab.types.HandleObject

Abstract Java class to represent MATLAB handle objects

## Description

Java represents handle objects that are passed from MATLAB as instances of the `HandleObject` class. When passing a handle object back to MATLAB, Java passes a reference to the `HandleObject` instance. This reference can be either an array or a scalar, depending on the original handle object passed to Java from MATLAB.

## Creation

You cannot construct a `HandleObject` in Java. You only can pass a handle object to the MATLAB session in which it was originally created.

## Examples

### Get Handle Object from MATLAB

This example starts a shared MATLAB session and creates a `containers.Map` object in the MATLAB workspace. The statement evaluated in the MATLAB workspace returns a handle variable that refers to the `Map` object.

The engine `getVariable` function returns the MATLAB handle variable as a `HandleObject` instance. This instance is used to call the MATLAB `keys` function to obtain the `Map` keys.

```
import com.mathworks.engine.MatlabEngine;
import com.mathworks.matlab.types.*;

MatlabEngine engine = MatlabEngine.startMatlab();
engine.eval("cm = containers.Map({'id','name'},{11,'mw'});");
HandleObject handle = engine.getVariable("cm");
String[] cells = engine.feval("keys", handle);
```

## See Also

`com.mathworks.matlab.types.CellStr` | `com.mathworks.matlab.types.Complex` | `com.mathworks.matlab.types.Struct`

**Topics**

"Using MATLAB Handle Objects in Java"

**Introduced in R2016b**

# com.mathworks.matlab.types.Struct

Java class to pass MATLAB `struct` to and from MATLAB

## Description

The `Struct` class provides support for passing data between MATLAB and Java as a MATLAB `struct`. The `Struct` class implements the `java.util.Map` interface.

The `Struct` class is designed as an immutable type. Attempting to change the mappings, keys, or values of the returned `Struct` causes an `UnsupportedOperationException`. Calling these methods can cause the exception: `put()`, `putAll()`, `remove()`, `entrySet()`, `keySet()`, and `values()`.

For an example, see "Using MATLAB Structures in Java".

## Creation

`Struct s = new Struct("field1",value1,"field2",value2, ...)` creates an instance of `Struct` with the specified field names and values.

## Methods

### Public Methods

| | |
|---|---|
| `containsKey(Object key)` | Returns true if this map contains a mapping for the specified key. |
| `containsValue(Object value)` | Returns true if this map maps one or more keys to the specified value. |
| `entrySet()` | Returns a `Set` view of the mappings contained in this map. |
| `equals(Object o)` | Compares the specified object with this map for equality. |
| `get(Object key)` | Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key. |
| `hashCode()` | Returns the hash code value for this map. |
| `isEmpty()` | Returns `true` if this map contains no key-value mappings. |
| `keySet()` | Returns a `Set` view of the keys contained in this map. |
| `size()` | Returns the number of key-value mappings in this map. |
| `values()` | Returns a `Collection` view of the values contained in this map. |

## Examples

### Create `Struct` for MATLAB Function Argument

Create a `Struct` and assign a key and value.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

class StructProperties {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        int[] y = {1,2,3,4,5};
        double[] color = {1.0,0.5,0.7};
        Struct s = new Struct("Color",color,"LineWidth",2);
        eng.feval("plot",y,s);
    }
}
```

## See Also

`com.mathworks.matlab.types.CellStr` | `com.mathworks.matlab.types.Complex` | `com.mathworks.matlab.types.HandleObject`

**Topics**

"Using MATLAB Structures in Java"

**Introduced in R2016b**

# com.mathworks.matlab.types.ValueObject

Abstract Java class to represent MATLAB value objects

## Description

Java represents value objects that are passed from MATLAB as instances of the `ValueObject` class.

## Creation

You cannot construct a `ValueObject` in Java. You only can pass a value object to the MATLAB session in which it was originally created.

## Examples

### Create Polygon Object in MATLAB

Create a polygon and call the `numsides` method.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

public class PassValueObject {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();

        // CREATE VALUE OBJECT pgon = polyshape([0 0 1 3], [0 3 3 0]);
        ValueObject pgon = eng.feval("polyshape", new int[]{0,0,1,3}, new int[]{0,3,3,0});

        // CALL METHOD res = numsides(pgon)
        ns = eng.feval("numsides", pgon);
        System.out.println("Number of sides: " + ns);
        eng.close();
    }
}
```

## See Also

**Introduced in R2021a**

# com.mathworks.matlab.types.CellStr

Java class to represent MATLAB cell array of `char` vectors

## Description

The `CellStr` class provides support for passing data from Java to MATLAB as a MATLAB cell array of `char` vectors (called a `cellstr` in MATLAB, see `cellstr`). There are MATLAB functions that require cell arrays of `char` vectors as inputs. To pass arguments from Java to a MATLAB function requiring `cellst` inputs, use the Java `CellStr` class to create a compatible type.

A MATLAB `cellstr` is mapped to a Java `String` array.

## Creation

`CellStr(Object stringArray)` creates a `CellStr` using a `String` or `String` array. The `String` array can have multiple dimensions.

## Methods

### Public Methods

| | |
|---|---|
| `Object getStringArray()` | Get the `String` or `String` array used to create the `CellStr`. |
| `boolean equals(CellStr1,CellStr2)` | Compare one `CellStr` instance with another. Two `CellStr` instances are equal if the `String` or `String` array they contain are the same. |

## Examples

### Construct `CellStr`

- Construct a `CellStr` named `keySet` and put the variable in the MATLAB base workspace.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

class javaCellstr {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        CellStr keySet = new CellStr(new String[]{"Jan","Feb","Mar","Apr"});
        eng.putVariable("mapKeys",keySet);
        eng.close();
    }
}
```

### Construct `CellStr` Array

- Create a `CellStr` array and pass it to the MATLAB `plot` function to change the appearance of the graph produced by MATLAB. The call to the MATLAB `print` function exports the figure as a `jpeg` file named `myPlot.jpg`.

```
import com.mathworks.engine.*;
import com.mathworks.matlab.types.*;

class CellStrArray {
    public static void main(String[] args) throws Exception {
        MatlabEngine eng = MatlabEngine.startMatlab();
        String[][] strArray = new String[2][2];
        strArray[0][0] = "MarkerFaceColor";
        strArray[0][1] = "MarkerEdgeColor";
        strArray[1][0] = "green";
        strArray[1][1] = "red";
        CellStr markerCellStr = new CellStr(strArray);
        eng.putVariable("M",markerCellStr);
        eng.eval("plot(1:10,'--bs',M{:})");
        eng.eval("print('myPlot','-djpeg')");
        eng.close();
    }
}
```

## See Also

`com.mathworks.matlab.types.Complex` | `com.mathworks.matlab.types.HandleObject` | `com.mathworks.matlab.types.Struct`

### Topics

"Pass Java CellStr to MATLAB"

**Introduced in R2016b**

# engClose (C)

Quit MATLAB engine session

## C Syntax

```
#include "engine.h"
int engClose(Engine *ep);
```

## Description

Send a quit command to the MATLAB engine session and close the connection. Returns `0` on success and `1` on failure. Possible failure includes attempting to terminate an already-terminated MATLAB engine session.

## Input Arguments

**ep — Pointer to engine**
`Engine *`

Pointer to engine, specified as `Engine *`.

## Examples

See these examples in *matlabroot*`/extern/examples/eng_mat`:

- `engdemo.c` for a C example on UNIX® operating systems.
- `engwindemo.c` for a C example on Microsoft® Windows® operating systems.

## See Also
`engOpen`

**Introduced before R2006a**

# engEvalString (C)

Evaluate expression in string

## C Syntax

```
#include "engine.h"
int engEvalString(Engine *ep, const char *string);
```

## Description

`engEvalString` evaluates the expression contained in `string` for the MATLAB engine session, `ep`, previously started by `engOpen`.

### UNIX Operating Systems

On UNIX systems, `engEvalString` sends commands to the MATLAB workspace by writing down a pipe connected to the MATLAB *`stdin`* process. MATLAB reads back from *`stdout`* any output resulting from the command that ordinarily appears on the screen, into the buffer defined by `engOutputBuffer`.

To turn off output buffering in C, use:

```
engOutputBuffer(ep, NULL, 0);
```

### Microsoft Windows Operating Systems

On a Windows system, `engEvalString` communicates with MATLAB using a Component Object Model (COM) interface.

## Input Arguments

**ep — Pointer to engine**
`Engine *`

Pointer to engine, specified as `Engine *`.

**`string` — Expression to evaluate**
`const char *`

Expression to evaluate, specified as `const char *`.

## Output Arguments

**`status` — Status**
`int`

Status, returned as `int`. The function returns `1` if the engine session is no longer running or the engine pointer is invalid or `NULL`. Otherwise, returns `0` even if the MATLAB engine session cannot evaluate the command.

## Examples

See these examples in *matlabroot*/extern/examples/eng_mat:

- engdemo.c for a C example on UNIX operating systems.
- engwindemo.c for a C example on Microsoft Windows operating systems.

## See Also

engOpen | engOutputBuffer

**Introduced before R2006a**

# engGetVariable (C)

Copy variable from MATLAB engine workspace

## C Syntax

```
#include "engine.h"
mxArray *engGetVariable(Engine *ep, const char *name);
```

## Description

`engGetVariable` reads the named `mxArray` from the MATLAB engine session associated with `ep`.

The limit for the size of data transferred is 2 GB.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

## Input Arguments

**ep — Pointer to engine**
`Engine *`

Pointer to engine, specified as `Engine *`.

**name — Name of mxArray**
`const char *`

Name of `mxArray` to get from the MATLAB workspace, specified as `const char *`.

## Output Arguments

**`ptr` — Pointer to mxArray**
`mxArray *` | `NULL`

Pointer to a newly allocated `mxArray` structure, returned as `mxArray *`. Returns `NULL` if the attempt fails. `engGetVariable` fails if the named variable does not exist.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `engdemo.c` for a C example on UNIX operating systems.
- `engwindemo.c` for a C example on Microsoft Windows operating systems.

## See Also

`engPutVariable` | `mxDestroyArray`

**Introduced before R2006a**

# engGetVisible (C)

Determine visibility of MATLAB engine session

## C Syntax

```
#include "engine.h"
int engGetVisible(Engine *ep, bool *value);
```

## Arguments

`ep`

Engine pointer

`value`

Pointer to value returned from `engGetVisible`

## Returns

### Microsoft Windows Operating Systems Only

`0` on success, and `1` otherwise.

## Description

`engGetVisible` returns the current visibility setting for MATLAB engine session, `ep`. A *visible* engine session runs in a window on the Windows desktop, thus making the engine available for user interaction. MATLAB removes an invisible session from the desktop.

## Examples

The following code opens engine session `ep` and disables its visibility.

```
Engine *ep;
bool vis;

ep = engOpen(NULL);
engSetVisible(ep, 0);
```

To determine the current visibility setting, use:

```
engGetVisible(ep, &vis);
```

## See Also

`engSetVisible`

# Engine (C)

Type for MATLAB engine

## Description

A handle to a MATLAB engine object.

`Engine` is a C language opaque type.

You can call MATLAB as a computational engine by writing C programs that use the MATLAB engine library. `Engine` is the link between your program and the separate MATLAB engine process.

The header file containing this type is:

```
#include "engine.h"
```

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `engdemo.c` shows how to call the MATLAB engine functions from a C program.
- `engwindemo.c` show how to call the MATLAB engine functions from a C program for Windows systems.
- `fengdemo.F` shows how to call the MATLAB engine functions from a Fortran program.

## See Also

`engOpen`

# engOpen (C)

Start MATLAB engine session

## C Syntax

```
#include "engine.h"
Engine *engOpen(const char *startcmd);
```

## Description

`engOpen` starts a MATLAB process for using MATLAB as a computational engine.

### Windows Platforms

`engOpen` launches MATLAB without a desktop.

The function opens a COM channel to MATLAB. The MATLAB software you registered during installation starts. If you did not register during installation, then see "Register MATLAB as a COM Server".

### UNIX Platforms

On UNIX systems, `engOpen`:

1. Creates two pipes.
2. Forks a new process. Sets up the pipes to pass *`stdin`* and *`stdout`* from MATLAB (parent) software to two file descriptors in the engine program (child).
3. Executes a command to run MATLAB software (`rsh` for remote execution).

## Input Arguments

**`startcmd` — MATLAB startup command**
`const char *` | `NULL`

MATLAB startup command, specified as `const char *`.

On Windows systems, the `startcmd` string must be `NULL`.

On UNIX systems:

- If `startcmd` is `NULL` or the empty string, then `engOpen` starts a MATLAB process on the current host using the command `matlab`. If `startcmd` is a `hostname`, then `engOpen` starts a MATLAB process on the designated host by embedding the specified `hostname` string into the larger string:

  ```
  "rsh hostname \"/bin/csh -c 'setenv DISPLAY\
      hostname:0; matlab'\""
  ```
- If `startcmd` is any other string (has white space in it, or nonalphanumeric characters), then MATLAB executes the string literally.

## Output Arguments

**`ptr` — Handle to MATLAB engine**
`Engine *` | `NULL`

Handle to MATLAB engine, specified as `Engine *`. Returns `NULL` if the open fails.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `engdemo.c` for a C example on UNIX operating systems.
- `engwindemo.c` for a C example on Microsoft Windows operating systems.

## See Also

**Topics**
"Can't Start MATLAB Engine"

**Introduced before R2006a**

# engOpenSingleUse (C)

Start MATLAB engine session for single, nonshared use

## C Syntax

```
#include "engine.h"
Engine *engOpenSingleUse(const char *startcmd, void *dcom,
  int *retstatus);
```

## Arguments

`startcmd`

String to start MATLAB process. On Microsoft Windows systems, the `startcmd` string must be `NULL`.

`dcom`

Reserved for future use; must be `NULL`.

`retstatus`

Return status; possible cause of failure.

## Returns

### Microsoft Windows Operating Systems Only

Pointer to an engine handle, or `NULL` if the open fails.

### UNIX Operating Systems

Not supported on UNIX systems.

## Description

This routine allows you to start multiple MATLAB processes using MATLAB as a computational engine.

`engOpenSingleUse` starts a MATLAB process, establishes a connection, and returns a unique engine identifier, or `NULL` if the open fails. Each call to `engOpenSingleUse` starts a new MATLAB process.

`engOpenSingleUse` opens a COM channel to MATLAB. This starts the MATLAB software you registered during installation. If you did not register during installation, enter the following command at the MATLAB prompt:

```
!matlab -regserver
```

`engOpenSingleUse` allows single-use instances of an engine server. `engOpenSingleUse` differs from `engOpen`, which allows multiple applications to use the same engine server.

See “MATLAB COM Integration” for additional details.

# engOutputBuffer (C)

Specify buffer for MATLAB output

## C Syntax

```
#include "engine.h"
int engOutputBuffer(Engine *ep, char *p, int n);
```

## Description

`engOutputBuffer` defines a character buffer for `engEvalString` to return any output that ordinarily appears on the screen. Returns `1` if you pass it a `NULL` engine pointer. Otherwise, returns `0`.

The default behavior of `engEvalString` is to discard any standard output caused by the command it is executing. A call to `engOutputBuffer` with a buffer of nonzero length tells any subsequent calls to `engEvalString` to save output in the character buffer pointed to by `p`.

To turn off output buffering in C, use:

```
engOutputBuffer(ep, NULL, 0);
```

---

**Note** The buffer returned by `engEvalString` is not `NULL` terminated.

---

## Input Arguments

**ep — Pointer to engine**
`Engine *`

Pointer to engine, specified as `Engine *`.

**p — Pointer to character buffer**
`char *`

Pointer to character buffer, specified as `char *`.

**n — Length of buffer**
`int`

Length of buffer, specified as `int`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `engdemo.c` for a C example on UNIX operating systems.
- `engwindemo.c` for a C example on Microsoft Windows operating systems.

## See Also
`engEvalString` | `engOpen`

**Introduced before R2006a**

# engPutVariable (C)

Put variable into MATLAB engine workspace

## C Syntax

```
#include "engine.h"
int engPutVariable(Engine *ep, const char *name, const mxArray *pm);
```

## Description

`engPutVariable` writes `mxArray` `pm` to the engine `ep`, giving it the variable name `name`. Returns `0` if successful and `1` if an error occurs.

If the `mxArray` does not exist in the workspace, the function creates it. If an `mxArray` with the same name exists in the workspace, the function replaces the existing `mxArray` with the new `mxArray`.

The limit for the size of data transferred is 2 GB.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function.

The engine application owns the original `mxArray` and is responsible for freeing its memory. Although the `engPutVariable` function sends a copy of the `mxArray` to the MATLAB workspace, the engine application does not need to account for or free memory for the copy.

## Input Arguments

**ep — Pointer to engine**
`Engine *`

Pointer to engine, specified as `Engine *`.

**name — Name of mxArray**
`const char *`

Name of `mxArray` in the MATLAB workspace, specified as `const char *`.

**pm — Pointer to mxArray**
`const mxArray *`

Pointer to mxArray, specified as `const mxArray *`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `engdemo.c` for a C example on UNIX operating systems.
- `engwindemo.c` for a C example on Microsoft Windows operating systems.

## See Also

`engGetVariable`

**Introduced before R2006a**

# engSetVisible (C)

Show or hide MATLAB engine session

## C Syntax

```
#include "engine.h"
int engSetVisible(Engine *ep, bool value);
```

## Arguments

`ep`

Engine pointer

`value`

Value to set the `Visible` property to. Set `value` to `1` to make the engine window visible, or to `0` to make it invisible.

## Returns

### Microsoft Windows Operating Systems Only

`0` on success, and `1` otherwise.

## Description

`engSetVisible` makes the window for the MATLAB engine session, `ep`, either visible or invisible on the Windows desktop. You can use this function to enable or disable user interaction with the MATLAB engine session.

## Examples

The following code opens engine session `ep` and disables its visibility.

```
Engine *ep;
bool vis;

ep = engOpen(NULL);
engSetVisible(ep, 0);
```

To determine the current visibility setting, use:

```
engGetVisible(ep, &vis);
```

## See Also

`engGetVisible`

# engClose (Fortran)

Quit MATLAB engine session

## Fortran Syntax

```
#include "engine.h"
integer*4 engClose(ep)
mwPointer ep
```

## Description

Send a quit command to the MATLAB engine session and close the connection. Returns `0` on success and `1` on failure. Possible failure includes attempting to terminate an already-terminated MATLAB engine session.

## Input Arguments

**ep — Pointer to engine**
`mwPointer`

Pointer to engine, specified as `mwPointer`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `fengdemo.F` for a Fortran example.

## See Also
`engOpen`

**Introduced before R2006a**

# engEvalString (Fortran)

Evaluate expression in string

## Fortran Syntax

```
#include "engine.h"
integer*4 engEvalString(ep, string)
mwPointer ep
character*(*) string
```

## Description

`engEvalString` evaluates the expression contained in `string` for the MATLAB engine session, `ep`, previously started by `engOpen`.

### UNIX Operating Systems

On UNIX systems, `engEvalString` sends commands to the MATLAB workspace by writing down a pipe connected to the MATLAB *`stdin`* process. MATLAB reads back from *`stdout`* any output resulting from the command that ordinarily appears on the screen, into the buffer defined by `engOutputBuffer`.

To turn off output buffering in Fortran, use:

```
engOutputBuffer(ep, '')
```

### Microsoft Windows Operating Systems

On a Windows system, `engEvalString` communicates with MATLAB using a Component Object Model (COM) interface.

## Input Arguments

**ep — Pointer to engine**
`mwPointer`

Pointer to engine, specified as `mwPointer`.

**`string` — Expression to evaluate**
`character*(*)`

Expression to evaluate, specified as `character*(*)`.

## Output Arguments

**`status` — Status**
`integer*4`

Status, returned as `integer*4`. The function returns `1` if the engine session is no longer running or the engine pointer is invalid or `NULL`. Otherwise, returns `0` even if the MATLAB engine session cannot evaluate the command.

## Examples

See these examples in *matlabroot*/extern/examples/eng_mat:

- fengdemo.F for a Fortran example.

## See Also

engOpen | engOutputBuffer

**Introduced before R2006a**

# engGetVariable (Fortran)

Copy variable from MATLAB engine workspace

## Fortran Syntax

```
#include "engine.h"
mwPointer engGetVariable(ep, name)
mwPointer ep
character*(*) name
```

## Description

`engGetVariable` reads the named `mxArray` from the MATLAB engine session associated with `ep`. Returns `0` if successful and `1` if an error occurs.

The limit for the size of data transferred is 2 GB.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

## Input Arguments

**ep — Pointer to engine**
`mwPointer`

Pointer to engine, specified as `mwPointer`.

**name — Name of mxArray**
`character*(*)`

Name of `mxArray` to get from the MATLAB workspace, specified as `character*(*)`.

**pm — Pointer to mxArray**
`mwPointer`

Pointer to mxArray, specified as `mwPointer`.

## Output Arguments

**`ptr` — Pointer to mxArray**
`mwPointer` | `NULL`

Pointer to a newly allocated `mxArray` structure, returned as `mwPointer`. Returns `NULL` if the attempt fails. `engGetVariable` fails if the named variable does not exist.

## See Also

`engPutVariable` | `mxDestroyArray`

**Introduced before R2006a**

# engOpen (Fortran)

Start MATLAB engine session

## Fortran Syntax

```
#include "engine.h"
mwPointer engOpen(startcmd)
character*(*) startcmd
```

## Description

`engOpen` starts a MATLAB process for using MATLAB as a computational engine.

### Windows Platforms

`engOpen` launches MATLAB without a desktop. The function opens a COM channel to MATLAB. The MATLAB software you registered during installation starts. If you did not register during installation, then see “Register MATLAB as a COM Server”.

### UNIX Platforms

On UNIX systems, `engOpen`:

1. Creates two pipes.
2. Forks a new process. Sets up the pipes to pass *`stdin`* and *`stdout`* from MATLAB (parent) software to two file descriptors in the engine program (child).
3. Executes a command to run MATLAB software (`rsh` for remote execution).

## Input Arguments

**`startcmd` — MATLAB startup command**
`character*(*)` | `NULL`

MATLAB startup command, specified as `character*(*)`.

On Windows systems, the `startcmd` string must be `NULL`.

On UNIX systems:

- If `startcmd` is `NULL` or the empty string, then `engOpen` starts a MATLAB process on the current host using the command `matlab`. If `startcmd` is a `hostname`, then `engOpen` starts a MATLAB process on the designated host by embedding the specified `hostname` string into the larger string:

  ```
  "rsh hostname \"/bin/csh -c 'setenv DISPLAY\
      hostname:0; matlab'\""
  ```
- If `startcmd` is any other string (has white space in it, or nonalphanumeric characters), then MATLAB executes the string literally.

## Output Arguments

**`ptr` — Handle to MATLAB engine**
`mwPointer` | `NULL`

Handle to MATLAB engine, specified as `mwPointer`. Returns `NULL` if the open fails.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `fengdemo.F` for a Fortran example.

## See Also

**Introduced before R2006a**

# engOutputBuffer (Fortran)

Specify buffer for MATLAB output

## Fortran Syntax

```
#include "engine.h"
integer*4 engOutputBuffer(ep, p)
mwPointer ep
character*n p
```

## Description

`engOutputBuffer` defines a character buffer for `engEvalString` to return any output that ordinarily appears on the screen. Returns `1` if you pass it a `NULL` engine pointer. Otherwise, returns `0`.

The default behavior of `engEvalString` is to discard any standard output caused by the command it is executing. A call to `engOutputBuffer` with a buffer of nonzero length tells any subsequent calls to `engEvalString` to save output in the character buffer pointed to by `p`.

To turn off output buffering in Fortran, use:

```
engOutputBuffer(ep, '')
```

**Note** The buffer returned by `engEvalString` is not `NULL` terminated.

## Input Arguments

**ep — Pointer to engine**
`mwPointer`

Pointer to engine, specified as `mwPointer`.

**p — Pointer to character buffer**
`character*n`

Pointer to character buffer, specified as `character*n`, where `n` is the length of the buffer.

## See Also

`engEvalString` | `engOpen`

**Introduced before R2006a**

# engPutVariable (Fortran)

Put variable into MATLAB engine workspace

## Fortran Syntax

```
#include "engine.h"
integer*4 engPutVariable(ep, name, pm)
mwPointer ep, pm
character*(*) name
```

## Description

`engPutVariable` writes `mxArray` `pm` to the engine `ep`, giving it the variable name `name`.

If the `mxArray` does not exist in the workspace, the function creates it. If an `mxArray` with the same name exists in the workspace, the function replaces the existing `mxArray` with the new `mxArray`.

The limit for the size of data transferred is 2 GB.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function.

The engine application owns the original `mxArray` and is responsible for freeing its memory. Although the `engPutVariable` function sends a copy of the `mxArray` to the MATLAB workspace, the engine application does not need to account for or free memory for the copy.

## Input Arguments

**ep — Pointer to engine**
`mwPointer`

Pointer to engine, specified as `mwPointer`.

**name — Name of mxArray**
`character*(*)`

Name of `mxArray` in the MATLAB workspace, specified as `character*(*)`.

## See Also

`engGetVariable`

**Introduced before R2006a**

# matClose (C and Fortran)

Close MAT-file

## C Syntax

```
#include "mat.h"
int matClose(MATFile *mfp);
```

## Fortran Syntax

```
#include "mat.h"
integer*4 matClose(mfp)
mwPointer mfp
```

## Arguments

`mfp`

Pointer to MAT-file information

## Returns

`EOF` in C (`-1` in Fortran) for a write error, and `0` if successful.

## Description

`matClose` closes the MAT-file associated with `mfp`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matcreat.c`
- `matdgns.c`
- `matdemo1.F`
- `matdemo2.F`

## See Also

`matOpen`

**Introduced before R2006a**

# matDeleteVariable (C and Fortran)

Delete array from MAT-file

## C Syntax

```
#include "mat.h"
int matDeleteVariable(MATFile *mfp, const char *name);
```

## Fortran Syntax

```
#include "mat.h"
integer*4 matDeleteVariable(mfp, name)
mwPointer mfp
character*(*) name
```

## Arguments

`mfp`

Pointer to MAT-file information

`name`

Name of `mxArray` to delete

## Returns

`0` if successful, and nonzero otherwise.

## Description

`matDeleteVariable` deletes the named `mxArray` from the MAT-file pointed to by `mfp`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matdemo1.F`

**Introduced before R2006a**

# MATFile (C and Fortran)

Type for MAT-file

## Description

A handle to a MAT-file object. A MAT-file is the data file format MATLAB software uses for saving data to your disk.

`MATFile` is a C language opaque type.

The MAT-file interface library contains routines for reading and writing MAT-files. Call these routines from your own C/C++ and Fortran programs, using `MATFile` to access your data file.

The header file containing this type is:

```
#include "mat.h"
```

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matcreat.c`
- `matdgns.c`
- `matdemo1.F`
- `matdemo2.F`

## See Also

`matOpen`, `matClose`, `matPutVariable`, `matGetVariable`, `mxDestroyArray`

# matGetDir (C and Fortran)

List of variables in MAT-file

## C Syntax

```
#include "mat.h"
char **matGetDir(MATFile *mfp, int *num);
```

## Fortran Syntax

```
#include "mat.h"
mwPointer matGetDir(mfp, num)
mwPointer mfp
integer*4 num
```

## Arguments

`mfp`

Pointer to MAT-file information

`num`

Pointer to the variable containing the number of `mxArrays` in the MAT-file

## Returns

Pointer to an internal array containing pointers to the names of the `mxArrays` in the MAT-file pointed to by `mfp`. In C, each name is a `NULL`-terminated string. The `num` output argument is the length of the internal array (number of `mxArrays` in the MAT-file). If `num` is zero, `mfp` contains no arrays.

`matGetDir` returns `NULL` in C (`0` in Fortran). If `matGetDir` fails, sets `num` to a negative number.

## Description

This routine provides you with a list of the names of the `mxArrays` contained within a MAT-file.

`matGetDir` allocates memory for the internal array of strings using a `mxCalloc`. Free the memory using `mxFree` when you are finished with the array.

MATLAB variable names can be up to length `mxMAXNAM`, defined in the C header file `matrix.h`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matcreat.c`
- `matdgns.c`
- `matdemo2.F`

**Introduced before R2006a**

# matGetFp (C)

File pointer to MAT-file

## C Syntax

```
#include "mat.h"
FILE *matGetFp(MATFile *mfp);
```

## Arguments

`mfp`

Pointer to MAT-file information

## Returns

C file handle to the MAT-file with handle `mfp`. Returns `NULL` if `mfp` is a handle to a MAT-file in HDF5-based format.

## Description

Use `matGetFp` to obtain a C file handle to a MAT-file. Standard C library routines, like `ferror` and `feof`, use file handle to investigate errors.

**Introduced before R2006a**

# matGetNextVariable (C and Fortran)

Next array in MAT-file

## C Syntax

```
#include "mat.h"
mxArray *matGetNextVariable(MATFile *mfp, const char **name);
```

## Fortran Syntax

```
#include "mat.h"
mwPointer matGetNextVariable(mfp, name)
mwPointer mfp
character*(*) name
```

## Arguments

`mfp`

Pointer to MAT-file information

`name`

Pointer to the variable containing the `mxArray` name

## Returns

Pointer to a newly allocated `mxArray` structure representing the next `mxArray` from the MAT-file pointed to by `mfp`. The function returns the name of the `mxArray` in `name`.

`matGetNextVariable` returns `NULL` in C (`0` in Fortran) for end of file or if there is an error condition. In C, use `feof` and `ferror` from the Standard C Library to determine status.

## Description

`matGetNextVariable` allows you to step sequentially through a MAT-file and read every `mxArray` in a single pass. The function reads and returns the next `mxArray` from the MAT-file pointed to by `mfp`.

Use `matGetNextVariable` immediately after opening the MAT-file with `matOpen` and not with other MAT-file routines. Otherwise, the concept of the *next* `mxArray` is undefined.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

The order of variables returned from successive calls to `matGetNextVariable` is not guaranteed to be the same order in which the variables were written.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matdgns.c`
- `matdemo2.F`

## See Also

`matGetNextVariableInfo`, `matGetVariable`, `mxDestroyArray`

**Introduced before R2006a**

# matGetNextVariableInfo (C and Fortran)

Array header information only

## C Syntax

```
#include "mat.h"
mxArray *matGetNextVariableInfo(MATFile *mfp, const char **name);
```

## Fortran Syntax

```
#include "mat.h"
mwPointer matGetNextVariableInfo(mfp, name)
mwPointer mfp
character*(*) name
```

## Arguments

`mfp`

Pointer to MAT-file information

`name`

Pointer to the variable containing the `mxArray` name

## Returns

Pointer to a newly allocated `mxArray` structure representing header information for the next `mxArray` from the MAT-file pointed to by `mfp`. The function returns the name of the `mxArray` in `name`.

`matGetNextVariableInfo` returns `NULL` in C (`0` in Fortran) when the end of file is reached or if there is an error condition. In C, use `feof` and `ferror` from the Standard C Library to determine status.

## Description

`matGetNextVariableInfo` loads only the array header information, including everything except `pr`, `pi`, `ir`, and `jc`, from the current file offset.

If `pr`, `pi`, `ir`, and `jc` are nonzero values when loaded with `matGetVariable`, `matGetNextVariableInfo` sets them to `-1` instead. These headers are for informational use only. *Never* pass this data back to the MATLAB workspace or save it to MAT-files.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

The order of variables returned from successive calls to `matGetNextVariableInfo` is not guaranteed to be the same order in which the variables were written.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matdgns.c`
- `matdemo2.F`

## See Also

`matGetNextVariable`, `matGetVariableInfo`

**Introduced before R2006a**

# matGetVariable (C and Fortran)

Array from MAT-file

## C Syntax

```
#include "mat.h"
mxArray *matGetVariable(MATFile *mfp, const char *name);
```

## Fortran Syntax

```
#include "mat.h"
mwPointer matGetVariable(mfp, name)
mwPointer mfp
character*(*) name
```

## Arguments

`mfp`

Pointer to MAT-file information

`name`

Name of `mxArray` to get from MAT-file

## Returns

Pointer to a newly allocated `mxArray` structure representing the `mxArray` named by `name` from the MAT-file pointed to by `mfp`.

`matGetVariable` returns `NULL` in C (`0` in Fortran) if the attempt to return the `mxArray` named by `name` fails.

## Description

This routine allows you to copy an `mxArray` out of a MAT-file.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matcreat.c`
- `matdemo1.F`

## See Also

`matPutVariable`, `mxDestroyArray`

**Introduced before R2006a**

# matGetVariableInfo (C and Fortran)

Array header information only

## C Syntax

```
#include "mat.h"
mxArray *matGetVariableInfo(MATFile *mfp, const char *name);
```

## Fortran Syntax

```
#include "mat.h"
mwPointer matGetVariableInfo(mfp, name)
mwPointer mfp
character*(*) name
```

## Arguments

`mfp`

Pointer to MAT-file information

`name`

Name of `mxArray` to get from MAT-file

## Returns

Pointer to a newly allocated `mxArray` structure representing header information for the `mxArray` named by `name` from the MAT-file pointed to by `mfp`.

`matGetVariableInfo` returns `NULL` in C (`0` in Fortran) if the attempt to return header information for the `mxArray` named by `name` fails.

## Description

`matGetVariableInfo` loads only the array header information, including everything except `pr`, `pi`, `ir`, and `jc`. It recursively creates the cells and structures through their leaf elements, but does not include `pr`, `pi`, `ir`, and `jc`.

If `pr`, `pi`, `ir`, and `jc` are nonzero values when loaded with `matGetVariable`, `matGetVariableInfo` sets them to `-1` instead. These headers are for informational use only. *Never* pass this data back to the MATLAB workspace or save it to MAT-files.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

## Examples

See these examples in *matlabroot*`/extern/examples/eng_mat`:

- `matdemo2.F`

## See Also

`matGetVariable`

**Introduced before R2006a**

# matGetErrno (C and Fortran)

Error codes for MAT-file API

## C Syntax

```
#include "mat.h"
matError matGetErrno(MATFile *mfp)
```

## Fortran Syntax

```
#include "mat.h"
matError matGetErrno(mfp)
mwPointer mfp
```

## Arguments

`mfp`

Pointer to MAT-file

## Returns

`matError` error code enumeration

```
typedef enum {
    mat_NO_ERROR = 0,
    mat_UNKNOWN_ERROR,
    mat_GENERIC_READ_ERROR,
    mat_GENERIC_WRITE_ERROR,

    mat_INDEX_TOO_BIG,
    /* Read-time error indicating that (typically) an index or dimension
     * written on a 64-bit platform exceeds 2^32, and we're trying to
     * read it on a 32-bit platform. */

    mat_FILE_FORMAT_VIOLATION,
    /* Read-time error indicating that some data or structure internal to
     * MAT file is bad - damaged or written improperly. */

    mat_FAIL_TO_IDENTIFY,
    /* Read-time error indicating that the contents of the file do not
     * match any known type of MAT file. */

    mat_BAD_ARGUMENT,
    /* Unsuitable data was passed to the MAT API */

    mat_OUTPUT_BAD_DATA,
    /* Write-time error indicating that something in the mxArray makes it
     * not suitable to write. */

    mat_FULL_OBJECT_OUTPUT_CONVERT,
    /* Write-time error indicating that conversion of an object (opaque or
     * OOPS) to a saveable form, has failed.  In this case the object is the
     * value of a variable, and the variable will not be saved at all. */

    mat_PART_OBJECT_OUTPUT_CONVERT,
    /* Write-time error indicating that conversion of an object (opaque or
     * OOPS) to a saveable form, has failed.  In this case the object is
     * the value in a field or element of a variable, and the variable
     * will be saved with an empty in that field or element. */

    mat_FULL_OBJECT_INPUT_CONVERT,
    /* Read-time error indicating that conversion of saveable data
     * to an object (opaque or OOPS), has failed.  In this case the object
```

```
     * is the value of a variable, and the variable has not been loaded. */

    mat_PART_OBJECT_INPUT_CONVERT,
    /* Read-time error indicating that conversion of saveable data
     * to an object (opaque or OOPS), has failed.  In this case the object is
     * the value in a field or element of a variable, and the variable
     * will be loaded with an empty in that field or element. */

    mat_OPERATION_NOT_SUPPORTED,
    /* Error indicating that the particular MAT API operation is
     * not supported on this kind of MAT file, or this kind of stream. */

    mat_OUT_OF_MEMORY,
    /* Operations internal to the MAT library encountered out-of-memory. */

    mat_BAD_VARIABLE_NAME,
    /* The name for a MATLAB variable contains illegal characters,
     * or exceeds the length allowed for that file format. */

    mat_OPERATION_PROHIBITED_IN_WRITE_MODE,
    /* The operation requested is only available when the file is open
       in Read or Update mode.  For example: matGetDir. */

    mat_OPERATION_PROHIBITED_IN_READ_MODE,
    /* The operation requested is only available when the file is open
       in Write or Update mode.  For example: matPutVariable. */

    mat_WRITE_VARIABLE_DOES_NOT_EXIST,
    /* A write operation that requires a variable already exist did not find the
     * variable in the file.  For example: matDeleteVariable. */

    mat_READ_VARIABLE_DOES_NOT_EXIST,
    /* A read operation that requires a variable already exist did not find the
     * variable in the file.  For example: matGetVariable. */

    mat_FILESYSTEM_COULD_NOT_OPEN,
    /* The MAT module could not open the requested file. */

    mat_FILESYSTEM_COULD_NOT_OPEN_TEMPORARY,
    /* The MAT module could not open a temporary file. */

    mat_FILESYSTEM_COULD_NOT_REOPEN,
    /* The MAT module could not REopen the requested file. */

    mat_BAD_OPEN_MODE,
    /* The mode argument to matOpen did not match any expected value */

    mat_FILESYSTEM_ERROR_ON_CLOSE,
    /* The MAT module got an error while fclose-ing the file.  Might indicate a full
     * filesystem.  */

} matError;
```

## Introduced in R2011a

# matlab.engine.connect_matlab

Connect shared MATLAB session to MATLAB Engine for Python

## Syntax

```
eng = matlab.engine.connect_matlab(name=None)
eng = matlab.engine.connect_matlab( ___ ,background)
eng = matlab.engine.connect_matlab( ___ ,async)
```

## Description

`eng = matlab.engine.connect_matlab(name=None)` connects to the shared MATLAB session, `name`, and returns a `MatlabEngine` object as `eng`. The input argument `name` specifies the name of a MATLAB session that is already running on your local machine.

- If you specify `name` and the engine cannot find a shared MATLAB session of the same name, then you receive an `EngineError` exception.
- If you do not specify `name` and the engine cannot find any shared MATLAB sessions, then it starts a new shared MATLAB session.
- If you do not specify `name` and the engine finds multiple shared MATLAB sessions running, then it connects to the first created session.

`eng = matlab.engine.connect_matlab( ___ ,background)` connects asynchronously if `background` is `True`. You can use this syntax with the `name` input argument in the previous syntax.

`eng = matlab.engine.connect_matlab( ___ ,async)` connects asynchronously if `async` is `True`. Not recommended. Use the `background` argument instead. Do not use for Python® Version 3.7. For more information, see “Compatibility Considerations” on page 1-205.

## Examples

### Connect to MATLAB Session

Connect to a shared MATLAB session that is already running on your local machine.

```
import matlab.engine
eng = matlab.engine.connect_matlab()
eng.sqrt(4.0)
```

```
2.0
```

`matlab.engine.connect_matlab` connects to the first created shared MATLAB session. If no MATLAB sessions are shared, then `matlab.engine.connect_matlab` starts a new session.

### Connect to MATLAB Sessions by Name

When there are multiple shared MATLAB sessions on your local machine, connect to two different sessions one at a time by specifying their names.

Connect to the first created MATLAB session.

```
import matlab.engine
names = matlab.engine.find_matlab()
names
```

```
('MATLAB_6830', 'MATLAB_7090')
```

Connect to the next MATLAB session.

```
eng = matlab.engine.connect_matlab('MATLAB_7090')
eng.sqrt(4.0)
```

```
2.0
```

## Input Arguments

**name — Name of shared MATLAB session**
character array

Name of the shared MATLAB session, specified as a character array.

**`background` — Start MATLAB synchronously or asynchronously**
`False` (default) | logical

Connect to MATLAB synchronously or asynchronously, specified as a logical keyword argument.

Example: `matlab.engine.connect_matlab(background=True)`

**`async` — Start MATLAB synchronously or asynchronously**
`False` (default) | logical

Connect to MATLAB synchronously or asynchronously, specified as a logical keyword argument.

## Output Arguments

**eng — Python variable for communicating with MATLAB**
`MatlabEngine` object

Python variable for communicating with MATLAB, returned as a `MatlabEngine` object. `eng` communicates with a shared MATLAB session that is already running on your local machine

## Limitations

- Do not connect the engine multiple times to the same shared MATLAB session.

## Compatibility Considerations

### Use `background` Argument to Connect Asynchronously

For Python Version 3.7, `async` is a keyword and cannot be used as an argument for `matlab.engine.connect_matlab`. Use the `background` argument instead for all supported versions of Python.

## See Also

`matlab.engine.MatlabEngine` | `matlab.engine.find_matlab`

#### Topics

"Connect Python to Running MATLAB Session"
"Calling MATLAB from Python"

**Introduced in R2015b**

# matlab.engine.find_matlab

Find shared MATLAB sessions to connect to MATLAB Engine for Python

## Syntax

`names = matlab.engine.find_matlab()`

## Description

`names = matlab.engine.find_matlab()` finds all shared MATLAB sessions on your local machine and returns their names in a `tuple`. Any name in `names` can be the input argument to `matlab.engine.connect_matlab`. If there are no shared sessions running on your local machine, `matlab.engine.find_matlab` returns an empty `tuple`.

## Examples

### Find Shared MATLAB Sessions

Identify the shared MATLAB sessions running on your local machine and connect to one of them.

```
import matlab.engine
names = matlab.engine.find_matlab()
names
```

```
('MATLAB_6830', 'MATLAB_7090')
```

There are two shared MATLAB sessions running, so `matlab.engine.find_matlab` returns two names in a `tuple`.

Connect to the first shared MATLAB session.

```
eng = matlab.engine.connect_matlab('MATLAB_6830')
```

## See Also

`matlab.engine.connect_matlab`

**Topics**

“Connect Python to Running MATLAB Session”
“Calling MATLAB from Python”

**Introduced in R2015b**

# matlab.engine.FutureResult class

**Package:** `matlab.engine`

Results of asynchronous call to MATLAB function stored in Python object

## Description

The `FutureResult` class stores results of an asynchronous call to a MATLAB function in a Python object.

# Creation

The MATLAB Engine for Python creates a `FutureResult` object when a MATLAB function is called asynchronously. There is no need to call `matlab.engine.FutureResult()` to create `FutureResult` objects of your own.

## Methods

**Public Methods**

| | |
|---|---|
| cancel | Cancel asynchronous call to MATLAB function from Python |
| cancelled | Cancellation status of asynchronous call to MATLAB function from Python |
| done | Completion status of asynchronous call to MATLAB function from Python |
| result | Result of asynchronous call to MATLAB function from Python |

## Exceptions

| | |
|---|---|
| `SyntaxError` | Python exception, syntax error in function call |
| `TypeError` | Python exception, data type of output argument not supported |
| `matlab.engine.CancelledError` | MATLAB engine cannot cancel function call |
| `matlab.engine.InterruptedError` | Function call interrupted |
| `matlab.engine.MatlabExecutionError` | Function call fails to execute |
| `matlab.engine.RejectedExecutionError` | Engine terminated |
| `matlab.engine.TimeoutError` | Result cannot be returned within the timeout period |

## Examples

### Get Result of Asynchronous MATLAB Call from Python

Call the MATLAB `sqrt` function from Python. Set `async` to `True` to make the function call asynchronously.

```
import matlab.engine
eng = matlab.engine.start_matlab()
```

```
future = eng.sqrt(4.0,async=True)
ret = future.result()
print(ret)
```

```
2.0
```

## See Also

`matlab.engine.MatlabEngine`

**Topics**

"Call MATLAB Functions from Python"
"Call MATLAB Functions Asynchronously from Python"

**Introduced in R2014b**

# cancel

**Class:** `matlab.engine.FutureResult`
**Package:** `matlab.engine`

Cancel asynchronous call to MATLAB function from Python

## Syntax

`tf = FutureResult.cancel()`

## Description

`tf = FutureResult.cancel()` cancels a call to a MATLAB function called asynchronously from Python. `FutureResult.cancel` returns `True` if it successfully cancels the function, and `False` if it cannot cancel the function.

## Output Arguments

### `tf` — Cancellation status

`True` | `False`

Cancellation status, returned as either `True` or `False`. The status, `tf`, is `True` if `FutureResult.cancel` successfully cancels the asynchronous function call, and is `False` otherwise.

## Examples

### Cancel an Asynchronous Call

Start an endless loop in MATLAB with an asynchronous call to the `eval` function. Then, cancel it.

```
import matlab.engine
eng = matlab.engine.start_matlab()
ret = eng.eval("while 1; end",nargout=0,async=True)
tf = ret.cancel()
print(tf)
```

```
True
```

## See Also

# cancelled

**Class:** `matlab.engine.FutureResult`
**Package:** `matlab.engine`

Cancellation status of asynchronous call to MATLAB function from Python

## Syntax

`tf = FutureResult.cancelled()`

## Description

`tf = FutureResult.cancelled()` returns the cancellation status of a call to a MATLAB function called asynchronously from Python. `FutureResult.cancelled` returns `True` if a previous call to `FutureResult.cancel` succeeded, and `False` otherwise.

## Output Arguments

**`tf` — Cancellation status**
`True` | `False`

Cancellation status of an asynchronous function call, returned as either `True` or `False`.

## Examples

### Check Cancellation Status of Asynchronous Call

Start an endless loop in MATLAB with an asynchronous call to the `eval` function. Cancel it and check that the engine stopped the loop.

```
import matlab.engine
eng = matlab.engine.start_matlab()
ret = eng.eval("while 1; end",nargout=0,async=True)
eval_stop = ret.cancel()
tf = ret.cancelled()
print(tf)
```

```
True
```

## See Also

# done

**Class:** `matlab.engine.FutureResult`
**Package:** `matlab.engine`

Completion status of asynchronous call to MATLAB function from Python

## Syntax

`tf = FutureResult.done()`

## Description

`tf = FutureResult.done()` returns the completion status of a MATLAB function called asynchronously from Python. `FutureResult.done` returns `True` if the function has finished, and `False` if it has not finished.

## Output Arguments

**`tf` — Completion status of asynchronous function call**
`True` | `False`

Completion status of an asynchronous function call, returned as either `True` or `False`.

## Examples

### Check If Asynchronous Call Finished

Call the MATLAB `sqrt` function with `async = True`. Check the status of `ret` to learn if `sqrt` is finished.

```
import matlab.engine
eng = matlab.engine.start_matlab()
ret = eng.sqrt(4.0,async=True)
tf = ret.done()
print(tf)
```

```
True
```

When `ret.done()` returns `True`, then you can call `ret.result()` to return the square root.

## See Also

# result

**Class:** `matlab.engine.FutureResult`
**Package:** `matlab.engine`

Result of asynchronous call to MATLAB function from Python

## Syntax

```
ret = FutureResult.result(timeout=None)
```

## Description

`ret = FutureResult.result(timeout=None)` returns the actual result of a call to a MATLAB function called asynchronously from Python.

## Input Arguments

**`timeout` — Timeout value in seconds**
`None` (default) | Python `float`

Timeout value in seconds, specified as Python data type `float`, to wait for result of the function call. If `timeout = None`, the `FutureResult.result` function waits until the function call finishes, and then returns the result.

## Output Arguments

**`ret` — Result of asynchronous function call**
Python object

Result of an asynchronous function call, returned as a Python object, that is the actual output argument of a call to a MATLAB function.

## Examples

### Get MATLAB Output Argument from Asynchronous Call

Call the MATLAB `sqrt` function from Python. Set `async` to `True` and get the square root from the `FutureResult` object.

```
import matlab.engine
eng = matlab.engine.start_matlab()
future = eng.sqrt(4.0,async=True)
ret = future.result()
print(ret)
```

```
2.0
```

## See Also

# matlab.engine.MatlabEngine

Python object using MATLAB as computational engine within Python session

## Description

The `MatlabEngine` class uses a MATLAB process as a computational engine for Python. You can call MATLAB functions as methods of a `MatlabEngine` object because the functions are dynamically invoked when you call them. You also can call functions and scripts that you define. You can send data to, and retrieve data from, the MATLAB workspace associated with a `MatlabEngine` object.

## Creation

The `matlab.engine.start_matlab` method creates a `MatlabEngine` object each time it is called. There is no need to call `matlab.engine.MatlabEngine()` to create `MatlabEngine` objects of your own.

## Attributes

| Attribute | Description |
|---|---|
| `workspace` | Python dictionary containing references to MATLAB variables. You can assign data to, and get data from, a MATLAB variable through the `workspace`. The name of each MATLAB variable you create becomes a key in the `workspace` dictionary. The keys in `workspace` must be valid MATLAB identifiers (for example, you cannot use numbers as keys). |

## Methods

### Public Methods

The `matlab::engine::MATLABEngine` class provides these methods.

- `matlab.engine.start_matlab` — Start MATLAB
- `matlab.engine.find_matlab` — Find shared MATLAB sessions to connect to MATLAB Engine for Python
- `matlab.engine.connect_matlab` — Connect to shared MATLAB session

### Specialized Operators and Functions

You can call any MATLAB function as a method of a `MatlabEngine` object. The engine dynamically invokes a MATLAB function when you call it. The syntax shows positional, keyword, and output arguments of a function call.

```
ret =
MatlabEngine.matlabfunc(*args,nargout=1,background=False,stdout=sys.stsdout,s
tderr=sys.stderr)
```

Replace *matlabfunc* with the name of any MATLAB function (such as `isprime` or `sqrt`). Replace `*args` with input arguments for the MATLAB function you call. The keyword arguments specify:

- The number of output arguments the function returns
- Whether the engine calls the function asynchronously
- Where the engine sends standard output and standard error coming from the function

Specify keyword arguments only when specifying values that are not the default values shown in the syntax.

#### Input Arguments to MATLAB Function

| Argument | Description | Python Type |
|---|---|---|
| `*args` | Input arguments to MATLAB function, specified as positional arguments | Any Python types that the engine can convert to MATLAB types |

#### Keyword Arguments to Engine

| Argument | Description | Python Type |
|---|---|---|
| `nargout` | Number of output arguments from MATLAB function | `int`<br>**Default:** `1` |
| `background` | Flag to call MATLAB function asynchronously<br><br>`background` is an alias for `async`. However, for Python Version 3.7, `async` is a keyword and cannot be used as an argument. Use the `background` argument instead of `async` for all supported versions of Python. | `bool`<br>**Default:** `False` |
| `stdout` | Standard output | `StringIO.StringIO` object (Python 2.7)<br>`io.StringIO` object (Python 3.x)<br>**Default:** `sys.stdout` |
| `stderr` | Standard error | `StringIO.StringIO` object (Python 2.7)<br>`io.StringIO` object (Python 3.x)<br>**Default:** `sys.stderr` |

#### Output Arguments

| Output Type | Description | Required Keyword Arguments |
|---|---|---|
| Python variable | One output argument from MATLAB function | Default values |

| Output Type | Description | Required Keyword Arguments |
|---|---|---|
| `tuple` | Multiple output arguments from MATLAB function | `nargout=`*n* (where *n* `> 1`) |
| `None` | No output argument from MATLAB function | `nargout=0` |
| `FutureResult` object | A placeholder for output arguments from asynchronous call to MATLAB function | `background=True` |

## Exceptions

| Exception | Description |
|---|---|
| `MatlabExecutionError` | Function call fails to execute |
| `RejectedExecutionError` | MATLAB engine terminated |
| `SyntaxError` | Syntax error in a function call |
| `TypeError` | Data type of an input or output argument not supported |

## Examples

### Call MATLAB Functions from Python

Call the MATLAB `sqrt` function from Python using the engine.

```
import matlab.engine
eng = matlab.engine.start_matlab()
ret = eng.sqrt(4.0)
print(ret)
```

```
2.0
```

### Put Array Into MATLAB Workspace

Create an array in Python and put it into the MATLAB workspace.

```
import matlab.engine
eng = matlab.engine.start_matlab()
px = eng.linspace(0.0,6.28,1000)
```

`px` is a MATLAB array, but `eng.linspace` returned it to Python. To use it in MATLAB, put the array into the MATLAB workspace.

```
eng.workspace['mx'] = px
```

When you add an entry to the engine `workspace` dictionary, you create a MATLAB variable, as well. The engine converts the data to a MATLAB data type.

#### Get Data from MATLAB Workspace

Get pi from the MATLAB workspace and copy it to a Python variable.

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.eval('a = pi;',nargout=0)
mpi = eng.workspace['a']
print(mpi)
```

```
3.14159265359
```

## See Also

matlab.engine.FutureResult | matlab.engine.connect_matlab | matlab.engine.find_matlab | matlab.engine.start_matlab

#### Topics

"Call MATLAB Functions from Python"
"Call MATLAB Functions Asynchronously from Python"
"Redirect Standard Output and Error to Python"
"Calling MATLAB from Python"

**Introduced in R2014b**

# matlab.engine.start_matlab

Start MATLAB Engine for Python

## Syntax

```
eng = matlab.engine.start_matlab()

eng = matlab.engine.start_matlab(option)
eng = matlab.engine.start_matlab(background)
eng = matlab.engine.start_matlab(async)
eng = matlab.engine.start_matlab( ___ )
```

## Description

`eng = matlab.engine.start_matlab()` starts a new MATLAB process, and returns Python variable `eng`, which is a `MatlabEngine` object for communicating with the MATLAB process.

If MATLAB cannot be started, the engine raises an `EngineError` exception.

`eng = matlab.engine.start_matlab(option)` uses startup options specified by `option`.

For example, call `matlab.engine.start_matlab('-desktop')` to start the MATLAB desktop from Python.

`eng = matlab.engine.start_matlab(background)` starts MATLAB asynchronously if `background` is `True`.

`eng = matlab.engine.start_matlab(async)` starts MATLAB asynchronously if `async` is `True`. Not recommended. Use the `background` argument instead. Do not use for Python Version 3.7. For more information, see "Compatibility Considerations" on page 1-221.

`eng = matlab.engine.start_matlab( ___ )` can include any of the input arguments in previous syntaxes.

## Examples

### Start MATLAB Engine from Python

Start an engine and a new MATLAB process from the Python command line.

```
import matlab.engine
eng = matlab.engine.start_matlab()
```

### Start Multiple Engines

Start a different MATLAB process from each engine.

```
import matlab.engine
eng1 = matlab.engine.start_matlab()
eng2 = matlab.engine.start_matlab()
```

#### Start MATLAB Desktop with Engine

Start an engine with the MATLAB desktop.

```
import matlab.engine
eng = matlab.engine.start_matlab("-desktop")
```

You also can start the desktop after you start the engine.

```
import matlab.engine
eng = matlab.engine.start_matlab()
eng.desktop(nargout=0)
```

**Note** You can call MATLAB functions from both the desktop and Python.

#### Start Engine Asynchronously

Start the engine with `background=True`. While MATLAB starts, you can enter commands at the Python command line.

```
import matlab.engine
future = matlab.engine.start_matlab(background=True)
eng = future.result()
eng.sqrt(4.)
```

```
2.0
```

## Input Arguments

**`option` — Startup options for MATLAB process**
`'-nodesktop'` (default) | string

Startup options for the MATLAB process, specified as a string. You can specify multiple startup options. The engine supports all MATLAB startup options, except for the options listed in “Limitations” on page 1-221. For a list of options, see the platform-specific command `matlab (Windows)`, `matlab (macOS)`, or `matlab (Linux)`.

To start MATLAB with the desktop, use the `'-desktop'` option.

Example: `matlab.engine.start_matlab('-desktop -r "format short"')` starts the desktop from Python. The engine passes `'-r "format short"'` to MATLAB.

**`background` — Start MATLAB synchronously or asynchronously**
`False` (default) | logical

Start MATLAB synchronously or asynchronously, specified as a logical keyword argument. `background` is an alias for `async`.

Example: `matlab.engine.start_matlab(background=True)`

**`async` — Start MATLAB synchronously or asynchronously**
`False` (default) | logical

Start MATLAB synchronously or asynchronously, specified as a logical keyword argument.

## Output Arguments

**eng — Python variable for communicating with MATLAB**
`MatlabEngine` object | `FutureResult` object

Python variable for communicating with MATLAB, returned as a `MatlabEngine` object if `async` or `background` is set to `False` or a `FutureResult` object if `async` or `background` is set to `True`.

Each time you call `matlab.engine.start_matlab`, it starts a new MATLAB process.

## Limitations

The engine does not support these MATLAB startup options:

- `-h`
- `-help`
- `-?`
- `-n`
- `-e`
- `-softwareopengl`
- `-logfile`

## Compatibility Considerations

### Use `background` Argument to Start Engine Asynchronously

For Python Version 3.7, `async` is a keyword and cannot be used as an argument for `matlab.engine.start_matlab`. Use the `background` argument instead for all supported versions of Python.

## See Also

`matlab.engine.MatlabEngine` | `matlab.engine.connect_matlab` | `matlab.engine.find_matlab`

**Topics**
"Start and Stop MATLAB Engine for Python"
"Specify Startup Options"
"Commonly Used Startup Options"
"Calling MATLAB from Python"

**Introduced in R2014b**

# matOpen (C and Fortran)

Open MAT-file

## C Syntax

```
#include "mat.h"
MATFile *matOpen(const char *filename, const char *mode);
```

## Fortran Syntax

```
#include "mat.h"
mwPointer matOpen(filename, mode)
character*(*) filename, mode
```

## Arguments

`filename`

Name of file to open

`mode`

File opening mode. The following table lists valid values for `mode`.

| | |
|---|---|
| `r` | Opens file for reading only; determines the current version of the MAT-file by inspecting the files and preserves the current version. |
| `u` | Opens file for update, both reading and writing. If the file does not exist, does not create a file (equivalent to the `r+` mode of `fopen`). Determines the current version of the MAT-file by inspecting the files and preserves the current version. |
| `w` | Opens file for writing only; deletes previous contents, if any. |
| `w4` | Creates a MAT-file compatible with MATLAB Versions 4 software and earlier. |
| `w6` | Creates a MAT-file compatible with MATLAB Version 5 (R8) software or earlier. Equivalent to `wL` mode. |
| `wL` | Opens file for writing character data using the default character set for your system. Use MATLAB Version 6 or 6.5 software to read the resulting MAT-file.<br><br>If you do not use the `wL` mode switch, MATLAB writes character data to the MAT-file using Unicode® character encoding by default.<br><br>Equivalent to `w6` mode. |
| `w7` | Creates a MAT-file compatible with MATLAB Version 7.0 (R14) software or earlier. Equivalent to `wz` mode. |
| `wz` | Opens file for writing compressed data. By default, the MATLAB `save` function compresses workspace variables as they are saved to a MAT-file. To use the same compression ratio when creating a MAT-file with the `matOpen` function, use the `wz` option.<br><br>Equivalent to `w7` mode. |

| | |
|---|---|
| w7.3 | Creates a MAT-file in an HDF5-based format that can store objects that occupy more than 2 GB. |

## Returns

File handle, or NULL in C (0 in Fortran) if the open fails.

## Description

This routine opens a MAT-file for reading and writing.

## Examples

See these examples in *matlabroot*/extern/examples/eng_mat:

- matcreat.c
- matdgns.c
- matdemo1.F
- matdemo2.F

## See Also

matClose, save

**Introduced before R2006a**

# matPutVariable (C and Fortran)

Array to MAT-file

## C Syntax

```
#include "mat.h"
int matPutVariable(MATFile *mfp, const char *name, const mxArray *pm);
```

## Fortran Syntax

```
#include "mat.h"
integer*4 matPutVariable(mfp, name, pm)
mwPointer mfp, pm
character*(*) name
```

## Arguments

`mfp`

Pointer to MAT-file information

`name`

Name of `mxArray` to put into MAT-file

`pm`

`mxArray` pointer

## Returns

`0` if successful and nonzero if an error occurs. In C, use `feof` and `ferror` from the Standard C Library along with `matGetFp` to determine status. To interpret error codes returned by `matPutVariable`, call `matGetErrno`.

## Description

This routine puts an `mxArray` into a MAT-file.

`matPutVariable` writes `mxArray pm` to the MAT-file `mfp`. If the `mxArray` does not exist in the MAT-file, the function appends it to the end. If an `mxArray` with the same name exists in the file, the function replaces the existing `mxArray` with the new `mxArray` by rewriting the file.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function.

The size of the new `mxArray` can be different from the existing `mxArray`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matcreat.c`
- `matdemo1.F`

## See Also

`matGetErrno` | `matGetFp` | `matGetVariable`

**Introduced before R2006a**

# matPutVariableAsGlobal (C and Fortran)

Array to MAT-file as originating from global workspace

## C Syntax

```
#include "mat.h"
int matPutVariableAsGlobal(MATFile *mfp, const char *name, const mxArray *pm);
```

## Fortran Syntax

```
#include "mat.h"
integer*4 matPutVariableAsGlobal(mfp, name, pm)
mwPointer mfp, pm
character*(*) name
```

## Arguments

`mfp`

Pointer to MAT-file information

`name`

Name of `mxArray` to put into MAT-file

`pm`

`mxArray` pointer

## Returns

`0` if successful and nonzero if an error occurs. In C, use `feof` and `ferror` from the Standard C Library with `matGetFp` to determine status.

## Description

This routine puts an `mxArray` into a MAT-file. `matPutVariableAsGlobal` is like `matPutVariable`, except that MATLAB software loads the array into the global workspace and sets a reference to it in the local workspace. If you write to a MATLAB 4 format file, `matPutVariableAsGlobal` does not load it as global and has the same effect as `matPutVariable`.

`matPutVariableAsGlobal` writes `mxArray pm` to the MAT-file `mfp`. If the `mxArray` does not exist in the MAT-file, the function appends it to the end. If an `mxArray` with the same name exists in the file, the function replaces the existing `mxArray` with the new `mxArray` by rewriting the file.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function.

The size of the new `mxArray` can be different from the existing `mxArray`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matcreat.c`
- `matdemo1.F`

## See Also

`matPutVariable`, `matGetFp`

**Introduced before R2006a**

# mexAtExit (C and Fortran)

Register function to call when MEX function clears or MATLAB terminates

## C Syntax

```
#include "mex.h"
int mexAtExit(void (*ExitFcn)(void));
```

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mexAtExit(ExitFcn)
subroutine ExitFcn()
```

## Description

Use `mexAtExit` to register a function to call just before clearing the MEX function or terminating MATLAB. `mexAtExit` gives your MEX function a chance to perform tasks such as freeing persistent memory and closing files. Other typical tasks include closing streams or sockets.

Each MEX function can register only one active exit function at a time. If you call `mexAtExit` more than once, then MATLAB uses the `ExitFcn` from the more recent `mexAtExit` call as the exit function.

If a MEX function is locked, then you cannot clear the MEX file. Therefore, if you attempt to clear a locked MEX file, then MATLAB does not call the `ExitFcn`.

In Fortran, declare the `ExitFcn` as `external` in the Fortran routine that calls `mexAtExit` if it is not within the scope of the file.

---

**Caution** In C MEX files, throwing an exception in `ExitFcn` causes MATLAB to crash.

---

## Input Arguments

**`ExitFcn` — Function to run on exit**
`void *`

Function to run on exit, specified as a pointer.

## Output Arguments

**Res — Return code**
`0`

Always returns `0`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexatexit.c`

## See Also

`mexLock`, `mexUnlock`

**Introduced before R2006a**

# mexCallMATLAB (C)

Call MATLAB function, user-defined function, or MEX function

## C Syntax

```
#include "mex.h"
int mexCallMATLAB(int nlhs, mxArray *plhs[], int nrhs,
  mxArray *prhs[], const char *functionName);
```

## Description

**Note** To write MEX functions using modern C++ features and the “MATLAB Data API”, see “C++ MEX Applications”.

Call `mexCallMATLAB` to invoke internal MATLAB numeric functions, MATLAB operators, user-defined functions, or other MEX functions.

Both `mexCallMATLAB` and `mexEvalString` execute MATLAB commands. Use `mexCallMATLAB` for returning results (left side arguments) back to the MEX function. The `mexEvalString` function does not return values to the MEX function.

## Input Arguments

**`nlhs` — Number of output arguments**
`int`

Number of expected output `mxArrays`, specified as an integer less than or equal to 50.

**`plhs` — MATLAB arrays**
`mxArray*`

Array of pointers to the `mxArray` output arguments.

**Caution** The `plhs` argument for `mexCallMATLAB` is not the same as the `plhs` for `mexFunction`. Do not destroy an `mxArray` returned in `plhs` for `mexFunction`.

**`nrhs` — Number of input arguments**
`int`

Number of input `mxArrays`, specified as an integer less than or equal to 50.

**`prhs` — MATLAB arrays**
`mxArray*`

Array of pointers to the `mxArray` input arguments.

**`functionName` — MATLAB function name**
`const char*`

Name of the MATLAB built-in function, operator, user-defined function, or MEX function to call specified as `const char*`.

If `functionName` is an operator, place the operator inside a pair of double quotes, for example, `"+"`.

## Output Arguments

**`Status` — Status**
`int`

The function returns `0` if successful, and a nonzero value if unsuccessful.

## Error Handling

If `functionName` detects an error, MATLAB terminates the MEX function and returns control to the MATLAB prompt. To trap errors, use the `mexCallMATLABWithTrap` function.

## Limitations

- Avoid using the `mexCallMATLAB` function in Simulink® S-functions. If you do, do not store the resulting `plhs` `mxArray` pointers in any S-function block state that persists after the MEX function finishes. Outputs of `mexCallMATLAB` have temporary scope and are automatically destroyed at the end of the MEX function call.
- It is possible to generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. For example, this function returns two variables but only assigns one of them a value.

  ```
  function [a,b] = foo(c)
  a = 2*c;
  ```

  If you then call `foo` using `mexCallMATLAB`, the unassigned output variable is now type `mxUNKNOWN_CLASS`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexcallmatlab.c`
- `mexevalstring.c`
- `mexcallmatlabwithtrap.c`

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `sincall.c`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreatecellmatrix.c`
- `mxisclass.c`

## Tips

- MATLAB allocates dynamic memory to store the arrays in `plhs` for `mexCallMATLAB`. MATLAB automatically deallocates the dynamic memory when you exit the MEX function. However, if heap space is at a premium, call `mxDestroyArray` when you are finished with the arrays in `plhs`.

## See Also

`mexCallMATLABWithTrap` | `mexEvalString` | `mexFunction` | `mxDestroyArray`

### Topics

"matlab::engine::MATLABEngine::feval"

**Introduced before R2006a**

# mexCallMATLAB (Fortran)

Call MATLAB function, user-defined function, or MEX file

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mexCallMATLAB(nlhs, plhs, nrhs, prhs, functionName)
integer*4 nlhs, nrhs
mwPointer plhs(*), prhs(*)
character*(*) functionName
```

## Arguments

`nlhs`

Number of output arguments. Must be less than or equal to 50.

`plhs`

Array of pointers to output arguments

**Caution** The `plhs` argument for `mexCallMATLAB` is not the same as the `plhs` for `mexFunction`. Do not destroy an `mxArray` returned in `plhs` for `mexFunction`.

`nrhs`

Number of input arguments. Must be less than or equal to 50.

`prhs`

Array of pointers to input arguments

`functionName`

Character string containing name of the MATLAB built-in function, operator, user-defined function, or MEX function to call.

If `functionName` is an operator, place the operator inside a pair of single quotes, for example, `'+'`.

## Returns

`0` if successful, and a nonzero value if unsuccessful.

## Description

Call `mexCallMATLAB` to invoke internal MATLAB numeric functions, MATLAB operators, user-defined functions, or other MEX files. Both `mexCallMATLAB` and `mexEvalString` execute MATLAB commands. Use `mexCallMATLAB` for returning results (left side arguments) back to the MEX function. The `mexEvalString` function does not return values to the MEX function.

For a complete description of the input and output arguments passed to `functionName`, see `mexFunction`.

## Error Handling

If `functionName` detects an error, MATLAB terminates the MEX file and returns control to the MATLAB prompt. To trap errors, use the `mexCallMATLABWithTrap` function.

## Limitations

- Avoid using the `mexCallMATLAB` function in Simulink S-functions. If you do, do not store the resulting `plhs` `mxArray` pointers in any S-function block state that persists after the MEX function finishes. Outputs of `mexCallMATLAB` have temporary scope and are automatically destroyed at the end of the MEX function call.
- It is possible to generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. For example, this function returns two variables but only assigns one of them a value.

  ```
  function [a,b] = foo(c)
  a = 2*c;
  ```

  If you then call `foo` using `mexCallMATLAB`, the unassigned output variable is now type `mxUNKNOWN_CLASS`.

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `sincall.F`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxcreatecellmatrixf.F`

## Tips

- MATLAB allocates dynamic memory to store the arrays in `plhs` for `mexCallMATLAB`. MATLAB automatically deallocates the dynamic memory when you exit the MEX file. However, if heap space is at a premium, call `mxDestroyArray` when you are finished with the arrays in `plhs`.

## See Also

`mexCallMATLABWithTrap` | `mexEvalString` | `mexFunction` | `mxDestroyArray`

**Introduced before R2006a**

# mexCallMATLABWithTrap (C and Fortran)

Call MATLAB function, user-defined function, or MEX file and capture error information

## C Syntax

```
#include "mex.h"
mxArray *mexCallMATLABWithTrap(int nlhs, mxArray *plhs[], int nrhs,
   mxArray *prhs[], const char *functionName);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mexCallMATLABWithTrap(nlhs, plhs, nrhs, prhs, functionName)
integer*4 nlhs, nrhs
mwPointer plhs(*), prhs(*)
character*(*) functionName
```

## Description

The `mexCallMATLABWithTrap` function performs the same function as `mexCallMATLAB`. However, if MATLAB detects an error when executing `functionName`, MATLAB returns control to the line in the MEX file immediately following the call to `mexCallMATLABWithTrap`.

## Input Arguments

**`nlhs` — Number of output arguments**
`int`

Number of expected output `mxArrays`, specified as an integer less than or equal to 50.

**`plhs` — MATLAB arrays**
`mxArray*`

Array of pointers to the `mxArray` output arguments.

---

**Caution** The `plhs` argument for `mexCallMATLAB` is not the same as the `plhs` for `mexFunction`. Do not destroy an `mxArray` returned in `plhs` for `mexFunction`.

---

**`nrhs` — Number of input arguments**
`int`

Number of input `mxArrays`, specified as an integer less than or equal to 50.

**`prhs` — MATLAB arrays**
`mxArray*`

Array of pointers to the `mxArray` input arguments.

**functionName — MATLAB function name**
const char*

Name of the MATLAB built-in function, operator, user-defined function, or MEX function to call specified as `const char*`.

If `functionName` is an operator, place the operator inside a pair of single quotes, for example, `'+'`.

## Output Arguments

**ME — Exception**
mxArray* | mwPointer | NULL

`NULL` if no error occurred. Otherwise, returns a pointer specified as `mxArray*` in C or `mwPointer` in Fortran of class `MException`. For information about `MException`, see “Respond to an Exception”.

## See Also

`MException` | `mexCallMATLAB`

**Topics**
“Respond to an Exception”
“Automatic Cleanup of Temporary Arrays in MEX Files”

**Introduced in R2008b**

# mexErrMsgIdAndTxt (C and Fortran)

Display error message with identifier and return to MATLAB prompt

## C Syntax

```
#include "mex.h"
void mexErrMsgIdAndTxt(const char *errorid,
  const char *errormsg, ...);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mexErrMsgIdAndTxt(errorid, errormsg)
character*(*) errorid, errormsg
```

## Arguments

`errorid`

String containing a MATLAB message identifier. For information on creating identifiers, see `MException`.

`errormsg`

String to display, specified as `const char*` in C or `character*(*)` in Fortran. In C, the function supports either UTF-8 or local code page (LCP) encoding and the string can include conversion specifications, used by the ANSI® C `printf` function. The encoding for both the message text and the conversion arguments must be the same.

`...`

In C, any arguments used in the message. Each argument must have a corresponding conversion specification. Refer to your C documentation for `printf` conversion tables.

## Description

The `mexErrMsgIdAndTxt` function writes an error message to the MATLAB window. For more information, see the `error` function syntax statement using a message identifier. After the error message prints, MATLAB terminates the MEX file and returns control to the MATLAB prompt.

Calling `mexErrMsgIdAndTxt` does not clear the MEX file from memory. So, `mexErrMsgIdAndTxt` does not invoke the function registered through `mexAtExit`.

If your application called `mxCalloc` or one of the `mxCreate*` routines to allocate memory, `mexErrMsgIdAndTxt` automatically frees the allocated memory.

**Note** If you get warnings when using `mexErrMsgIdAndTxt`, you might have a memory management compatibility problem. For more information, see “Memory Management Issues”.

## Remarks

In addition to the `errorid` and `errormsg`, the `mexErrMsgIdAndTxt` function determines where the error occurred, and displays the following information. For example, in the function `foo`, `mexErrMsgIdAndTxt` displays:

```
Error using foo
```

If you compile your MEX file with the MinGW-w64 compiler, see the limitations with exception handling topic in “Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64”.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `arrayFillGetPr.c`
- `matrixDivide.c`
- `timestwo.F`
- `xtimesy.F`

### Validate char Input

The following code snippet checks if input argument, `prhs[0]`, is a string. If not, the code displays a warning. If there is an error reading the input string, the code displays an error message and terminates the MEX file.

```
char *buf;
int buflen;

// initialize variables

if (mxIsChar(prhs[0])) {
    if (mxGetString(prhs[0], buf, buflen) == 0) {
        mexPrintf("The input string is:  %s\n", buf);
    }
    else {
        mexErrMsgIdAndTxt("MyProg:ConvertString",
           "Could not convert string data.");
        // exit MEX file
    }
}
else {
    mexWarnMsgIdAndTxt("MyProg:InputString",
        "Input should be a string to print properly.");
}

// continue with processing
```

## See Also

`error` | `mexWarnMsgIdAndTxt`

### Topics

“Memory Considerations for Class Destructors”
“Troubleshooting and Limitations Compiling C/C++ MEX Files with MinGW-w64”

**Introduced before R2006a**

# mexErrMsgTxt (C and Fortran)

Display error message and return to MATLAB prompt

---

**Note** `mexErrMsgTxt` is not recommended. Use `mexErrMsgIdAndTxt` instead.

---

## C Syntax

```
#include "mex.h"
void mexErrMsgTxt(const char *errormsg);
```

## Fortran Syntax

```
subroutine mexErrMsgTxt(errormsg)
character*(*) errormsg
```

## Arguments

`errormsg`

String containing the error message to display

## Description

`mexErrMsgTxt` writes an error message to the MATLAB window. After the error message prints, MATLAB terminates the MEX-file and returns control to the MATLAB prompt.

Calling `mexErrMsgTxt` does not clear the MEX-file from memory. So, `mexErrMsgTxt` does not invoke the function registered through `mexAtExit`.

If your application called `mxCalloc` or one of the `mxCreate*` routines to allocate memory, `mexErrMsgTxt` automatically frees the allocated memory.

---

**Note** If you get warnings when using `mexErrMsgTxt`, you might have a memory management compatibility problem. For more information, see “Memory Management Issues”.

---

## Remarks

In addition to the `errormsg`, the `mexerrmsgtxt` function determines where the error occurred, and displays the following information. If an error labeled `Print my error message` occurs in the function `foo`, `mexerrmsgtxt` displays:

```
Error using foo
Print my error message
```

## See Also

`mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt`

# mexEvalString (C)

Execute MATLAB command in caller workspace

## C Syntax

```
#include "mex.h"
int mexEvalString(const char *command);
```

## Description

**Note** To write MEX functions using modern C++ features and the “MATLAB Data API”, see “C++ MEX Applications”.

Call `mexEvalString` to invoke a MATLAB `command` in the workspace of the caller.

`mexEvalString` and `mexCallMATLAB` both execute MATLAB commands. Use `mexCallMATLAB` for returning results (left side arguments) back to the MEX function. The `mexEvalString` function does not return values to the MEX function.

All arguments that appear to the right of an equal sign in the `command` string must be current variables of the caller workspace. Do not use MATLAB function names for variable names. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function. For more information, see “Variable Names”.

## Input Arguments

**command — MATLAB command name**
`const char*`

Name of the MATLAB command to execute, specified as `const char*`. The function supports UTF-8 characters.

## Output Arguments

**Status — Status**
`int`

The function returns `0` if successful, and `1` if an error occurs.

## Error Handling

If `command` detects an error, then MATLAB returns control to the MEX function and `mexEvalString` returns `1`. To trap errors, use the `mexEvalStringWithTrap` function.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexevalstring.c`

## See Also

`mexCallMATLAB` | `mexEvalStringWithTrap`

**Topics**

"matlab::engine::MATLABEngine::eval"

**Introduced before R2006a**

# mexEvalString (Fortran)

Execute MATLAB command in caller workspace

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mexEvalString(command)
character*(*) command
```

## Arguments

`command`

String containing MATLAB command to execute

## Returns

`0` if successful, and `1` if an error occurs.

## Description

Call `mexEvalString` to invoke a MATLAB `command` in the workspace of the caller.

`mexEvalString` and `mexCallMATLAB` both execute MATLAB commands. Use `mexCallMATLAB` for returning results (left side arguments) back to the MEX function. The `mexEvalString` function does not return values to the MEX function.

All arguments that appear to the right of an equal sign in the `command` string must be current variables of the caller workspace.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function. For more information, see “Variable Names”.

## Error Handling

If `command` detects an error, then MATLAB returns control to the MEX file and `mexEvalString` returns `1`. To trap errors, use the `mexEvalStringWithTrap` function.

## See Also

`mexCallMATLAB` | `mexEvalStringWithTrap`

**Introduced before R2006a**

# mexEvalStringWithTrap (C and Fortran)

Execute MATLAB command in caller workspace and capture error information

## C Syntax

```
#include "mex.h"
mxArray *mexEvalStringWithTrap(const char *command);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mexEvalStringWithTrap(command)
character*(*) command
```

## Description

The `mexEvalStringWithTrap` function performs the same function as `mexEvalString`. However, if MATLAB detects an error when executing `command`, MATLAB returns control to the line in the MEX file immediately following the call to `mexEvalStringWithTrap`.

## Input Arguments

**command — MATLAB command name**
`const char*` | `character*(*)`

Name of the MATLAB command to execute, specified as `const char*` in C or `character*(*)` in Fortran. In C, the function supports UTF-8 characters.

## Output Arguments

**ME — Exception**
`mxArray*` | `mwPointer` | `NULL`

`NULL` if no error occurred. Otherwise, returns a pointer specified as `mxArray*` in C or `mwPointer` in Fortran of class `MException`. For information about `MException`, see “Respond to an Exception”.

## See Also

`MException` | `mexCallMATLAB` | `mexEvalString`

**Topics**
“Respond to an Exception”

**Introduced before R2006a**

# mexFunction (C)

Entry point to C/C++ MEX function built with C Matrix API

## C Syntax

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
  const mxArray *prhs[])
```

## Description

**Note** To write MEX functions using modern C++ features and the “MATLAB Data API”, see “C++ MEX Applications”.

`mexFunction` is not a routine you call. Rather, `mexFunction` is the name of the gateway function in C which every MEX function requires. When you invoke a MEX function, MATLAB finds and loads the corresponding MEX function of the same name. MATLAB then searches for a symbol named `mexFunction` within the MEX function. If it finds one, it calls the MEX function using the address of the `mexFunction` symbol. MATLAB displays an error message if it cannot find a routine named `mexFunction` inside the MEX function.

When you invoke a MEX function, MATLAB automatically seeds `nlhs`, `plhs`, `nrhs`, and `prhs` with the calling arguments. In the syntax of the MATLAB language, functions have the general form:

```
[a,b,c,...] = fun(d,e,f,...)
```

where the `...` denotes more items of the same format. The `a,b,c...` are left-side output arguments, and the `d,e,f...` are right-side input arguments. The arguments `nlhs` and `nrhs` contain the number of left side and right side arguments, respectively. `prhs` is an array of `mxArray` pointers whose length is `nrhs`. `plhs` is an array whose length is `nlhs`, where your function must set pointers for the output `mxArrays`.

**Note** It is possible to return an output value even if `nlhs = 0`, which corresponds to returning the result in the `ans` variable.

To experiment with passing input arguments, build the `mexfunction.c` example, following the instructions in “Tables of MEX Function Source Code Examples”.

## Input Arguments

**`nlhs` — Number of output arguments**
`int`

Number of expected `mxArray` output arguments, specified as an integer.

**`plhs` — MATLAB arrays**
`mxArray*`

Array of pointers to the expected `mxArray` output arguments.

**`nrhs` — Number of input arguments**
`int`

Number of input `mxArrays`, specified as an integer.

**`prhs` — MATLAB arrays**
`const mxArray*`

Array of pointers to the `mxArray` input arguments. Do not modify any `prhs` values in your MEX file. Changing the data in these read-only `mxArrays` can produce undesired side effects.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexfunction.c`

## See Also

`matlab::mex::Function`

**Topics**
"Components of C MEX File"
"C MEX File Applications"
"C Matrix API"

**Introduced before R2006a**

# mexFunction (Fortran)

Entry point to Fortran MEX function

## Fortran Syntax

```
#include "fintrf.h"
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer nlhs, nrhs
mwPointer plhs(*), prhs(*)
```

## Arguments

`nlhs`

Number of expected output `mxArrays`

`plhs`

Array of pointers to the expected output `mxArrays`

`nrhs`

Number of input `mxArrays`

`prhs`

Array of pointers to the input `mxArrays`. Do not modify any `prhs` values in your MEX file. Changing the data in these read-only `mxArrays` can produce undesired side effects.

## Description

`mexFunction` is not a routine you call. Rather, `mexFunction` is the name of the gateway subroutine in Fortran which every MEX function requires. For more information, see "Components of Fortran MEX File". When you invoke a MEX function, MATLAB finds and loads the corresponding MEX function of the same name. MATLAB then searches for a symbol named `mexFunction` within the MEX function. If it finds one, it calls the MEX function using the address of the `mexFunction` symbol. MATLAB displays an error message if it cannot find a routine named `mexFunction` inside the MEX function.

When you invoke a MEX function, MATLAB automatically seeds `nlhs`, `plhs`, `nrhs`, and `prhs` with the calling arguments. In the syntax of the MATLAB language, functions have the general form:

```
[a,b,c,...] = fun(d,e,f,...)
```

where the `...` denotes more items of the same format. The `a,b,c...` are left-side output arguments, and the `d,e,f...` are right-side input arguments. The arguments `nlhs` and `nrhs` contain the number of left side and right side arguments, respectively. `prhs` is an array of `mxArray` pointers whose length is `nrhs`. `plhs` is an array whose length is `nlhs`, where your function must set pointers for the output `mxArrays`.

**Note** It is possible to return an output value even if `nlhs = 0`, which corresponds to returning the result in the `ans` variable.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexlockf.F`

## See Also

**Topics**
"Components of Fortran MEX File"
"Fortran MEX File Applications"
"Fortran Matrix API"
"Fortran MEX API"

**Introduced before R2006a**

# mexFunctionName (C and Fortran)

Name of current MEX function

## C Syntax

```
#include "mex.h"
const char *mexFunctionName(void);
```

## Fortran Syntax

```
#include "fintrf.h"
character*(*) mexFunctionName()
```

## Description

`mexFunctionName` returns the name of the current MEX function.

## Output Arguments

**`fName` — MEX function name**
`const char*` | `character*(*)`

Current MEX function name, returned as `const char*` in C or `character*(*)` in Fortran.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexgetarray.c`

**Introduced before R2006a**

# mexGet (C)

Value of specified graphics property

---

**Note** Do not use `mexGet`. Use `mxGetProperty` instead.

---

## C Syntax

```
#include "mex.h"
const mxArray *mexGet(double handle, const char *property);
```

## Arguments

`handle`

Handle to a particular graphics object

`property`

Graphics property

## Returns

Value of the specified property in the specified graphics object on success. Returns `NULL` on failure. Do not modify the return argument from `mexGet`. Changing the data in a `const` (read-only) `mxArray` can produce undesired side effects.

## Description

Call `mexGet` to get the value of the property of a certain graphics object. `mexGet` is the API equivalent of the MATLAB `get` function. To set a graphics property value, call `mexSet`.

## See Also

`mxGetProperty`, `mxSetProperty`

# mexGetVariable (C)

Copy of variable from specified workspace

## C Syntax

```
#include "mex.h"
mxArray *mexGetVariable(const char *workspace, const char
  *varname);
```

## Description

**Note** To write MEX functions using modern C++ features and the “MATLAB Data API”, see “C++ MEX Applications”.

Call `mexGetVariable` to get a copy of the specified variable. The returned `mxArray` contains a copy of all the data and characteristics that the variable had in the other workspace. Modifications to the returned `mxArray` do not affect the variable in the workspace unless you write the copy back to the workspace with `mexPutVariable`.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

## Input Arguments

### `workspace` — Workspace
`const char*`

Workspace `mexGetVariable` searches for `varname`, specified as `const char*`. The possible values are:

| | |
|---|---|
| `base` | Search for the variable in the base workspace. |
| `caller` | Search for the variable in the caller workspace. |
| `global` | Search for the variable in the global workspace. |

### `varname` — Variable name
`const char*`

Name of the variable to copy, specified as `const char*`.

## Output Arguments

### `var` — Copy of variable
`mxArray*`

Copy of variable, specified as `mxArray*`. The function returns `NULL` on failure. A common cause of failure is specifying a variable that is not currently in the workspace. Perhaps the variable was in the workspace at one time but has since been cleared.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexgetarray.c`

## See Also

`mexGetVariablePtr` | `mexPutVariable` | `mxDestroyArray`

### Topics

"matlab::engine::MATLABEngine::getVariable"

**Introduced before R2006a**

# mexGetVariable (Fortran)

Copy of variable from specified workspace

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mexGetVariable(workspace, varname)
character*(*) workspace, varname
```

## Arguments

`workspace`

Specifies where `mexGetVariable` searches for array `varname`. The possible values are:

| | |
|---|---|
| `base` | Search for the variable in the base workspace. |
| `caller` | Search for the variable in the caller workspace. |
| `global` | Search for the variable in the global workspace. |

`varname`

Name of the variable to copy

## Returns

Copy of the variable on success. Returns `0` on failure. A common cause of failure is specifying a variable that is not currently in the workspace. Perhaps the variable was in the workspace at one time but has since been cleared.

## Description

Call `mexGetVariable` to get a copy of the specified variable. The returned `mxArray` contains a copy of all the data and characteristics that the variable had in the other workspace. Modifications to the returned `mxArray` do not affect the variable in the workspace unless you write the copy back to the workspace with `mexPutVariable`.

Use `mxDestroyArray` to destroy the `mxArray` created by this routine when you are finished with it.

## See Also

`mexGetVariablePtr` | `mexPutVariable` | `mxDestroyArray`

**Introduced before R2006a**

# mexGetVariablePtr (C and Fortran)

Read-only pointer to variable from another workspace

## C Syntax

```
#include "mex.h"
const mxArray *mexGetVariablePtr(const char *workspace,
  const char *varname);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mexGetVariablePtr(workspace, varname)
character*(*) workspace, varname
```

## Description

Call `mexGetVariablePtr` to get a read-only pointer to the specified variable, `varname`, into your MEX-file workspace. This command is useful for examining an `mxArray`'s data and characteristics. If you want to change data or characteristics, use `mexGetVariable` (along with `mexPutVariable`) instead of `mexGetVariablePtr`.

If you simply want to examine data or characteristics, `mexGetVariablePtr` offers superior performance because the caller wants to pass only a pointer to the array.

## Input Arguments

**`workspace` — Workspace name**
`const char*` | `character*(*)`

Workspace name you want `mexGetVariablePtr` to search, specified as `const char*` in C or `character*(*)` in Fortran. The possible values are:

| | |
|---|---|
| `base` | Search for the variable in the base workspace. |
| `caller` | Search for the variable in the caller workspace. |
| `global` | Search for the variable in the global workspace. |

**`varname` — Variable name**
`const char*` | `character*(*)`

Name of a variable in another workspace, specified as `const char*` in C or `character*(*)` in Fortran. This is a variable name, not an `mxArray` pointer.

## Output Arguments

**`mxArray` — Pointer to mxArray**
`const mxArray*` | `mwPointer` | `NULL`

Read-only pointer to the `mxArray` on success, returned as `const mxArray*` in C or `mwPointer` in Fortran. Returns `NULL` in C or `0` in Fortran on failure.

## Limitations

- If you use this function in Simulink S-functions, do not store the resulting `plhs` `mxArray` pointers in any S-function block state that persists after the MEX function finishes. Outputs of this function have temporary scope and are automatically destroyed at the end of the MEX function call.

## See Also

`mexGetVariable`

**Introduced before R2006a**

# mexIsLocked (C and Fortran)

Determine if MEX file is locked

## C Syntax

```
#include "mex.h"
bool mexIsLocked(void);
```

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mexIsLocked()
```

## Description

Call `mexIsLocked` to determine if the MEX file is locked. By default, MEX files are unlocked, meaning you can clear the MEX file at any time.

To unlock a MEX file, call `mexUnlock`.

## Output Arguments

**res — Status**
bool | `integer*4`

Status, returned as `true` (logical `1` in C or `integer*4 1` in Fortran) if the MEX file is locked. Returns `false` (logical `0` in C or `integer*4 0` in Fortran) if the file is unlocked.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexlock.c`
- `mexlockf.F`

## See Also

`clear` | `mexLock` | `mexMakeArrayPersistent` | `mexMakeMemoryPersistent` | `mexUnlock`

# mexLock (C and Fortran)

Prevent clearing MEX file from memory

## C Syntax

```
#include "mex.h"
void mexLock(void);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mexLock()
```

## Description

By default, MEX files are unlocked, meaning you can clear them at any time. Call `mexLock` to prohibit clearing a MEX file.

To unlock a MEX file, call `mexUnlock`. Do not use the `munlock` function.

`mexLock` increments a lock count. If you call `mexLock` `n` times, call `mexUnlock` `n` times to unlock your MEX file.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexlock.c`
- `mexlockf.F`

## See Also

`mexIsLocked`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mexUnlock`, `clear`

# mexMakeArrayPersistent (C and Fortran)

Make array persist after MEX file completes

## C Syntax

```
#include "mex.h"
void mexMakeArrayPersistent(mxArray *pm);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mexMakeArrayPersistent(pm)
mwPointer pm
```

## Description

By default, an `mxArray` allocated by an `mxCreate*` function is not persistent. The MATLAB memory management facility automatically frees a nonpersistent `mxArray` when the MEX function finishes. If you want the `mxArray` to persist through multiple invocations of the MEX function, call the `mexMakeArrayPersistent` function.

**Warning** Do not assign an array created with the `mexMakeArrayPersistent` function to the `plhs` output argument of a MEX file.

**Note** If you create a persistent `mxArray`, you are responsible for destroying it using `mxDestroyArray` when the MEX file is cleared. If you do not destroy a persistent `mxArray`, MATLAB leaks memory. See `mexAtExit` to see how to register a function that gets called when the MEX file is cleared. See `mexLock` to see how to lock your MEX file so that it is never cleared.

## Input Arguments

**pm — Pointer to mxArray**
`mxArray * | mwPointer`

Pointer to an `mxArray` created by an `mxCreate*` function, specified as `mxArray *` in C or `mwPointer` in Fortran.

## See Also

`mexAtExit`, `mxDestroyArray`, `mexLock`, `mexMakeMemoryPersistent`, and the `mxCreate*` functions

**Introduced before R2006a**

# mexMakeMemoryPersistent (C and Fortran)

Make memory allocated by MATLAB persist after MEX function completes

## C Syntax

```
#include "mex.h"
void mexMakeMemoryPersistent(void *ptr);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mexMakeMemoryPersistent(ptr)
mwPointer ptr
```

## Description

By default, memory allocated by MATLAB is nonpersistent, so it is freed automatically when the MEX function finishes. If you want the memory to persist, call `mexMakeMemoryPersistent`.

**Note** If you create persistent memory, you are responsible for freeing it when the MEX function is cleared. If you do not free the memory, MATLAB leaks memory. To free memory, use `mxFree`. See `mexAtExit` to see how to register a function that gets called when the MEX function is cleared. See `mexLock` to see how to lock your MEX function so that it is never cleared.

## Input Arguments

**`ptr` — Pointer to memory**
`mxArray *` | `mwPointer`

Pointer to the beginning of memory allocated by one of the MATLAB memory allocation routines, specified as `mxArray *` in C or `mwPointer` in Fortran.

## See Also

`mexAtExit`, `mexLock`, `mexMakeArrayPersistent`, `mxCalloc`, `mxFree`, `mxMalloc`, `mxRealloc`

**Introduced before R2006a**

# mexPrintf (C and Fortran)

ANSI C PRINTF-style output routine

## C Syntax

```
#include "mex.h"
int mexPrintf(const char *message, ...);
```

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mexPrintf(message)
character*(*) message
```

## Description

This routine prints a string on the screen and in the diary (if the diary is in use). It provides a callback to the standard C `printf` routine already linked inside MATLAB software, which avoids linking the entire `stdio` library into your MEX file.

In a C MEX file, call `mexPrintf` instead of `printf` to display a string.

---

**Note** If you want the literal `%` in your message, use `%%` in the message string since `%` has special meaning to `printf`. Failing to do so causes unpredictable results.

---

## Input Arguments

**`message` — String to display**
`const char*` | `character*(*)`

String to display, specified as `const char*` in C or `character*(*)` in Fortran. In C, the function supports either UTF-8 or local code page (LCP) encoding and the string can include conversion specifications, used by the ANSI C `printf` function. The encoding for both the message text and the conversion arguments must be the same.

**`...` — Conversion arguments**
`const char*`

In C, any arguments used in the message. Each argument must have a corresponding conversion specification. Refer to your C documentation for `printf` conversion tables.

## Output Arguments

**`res` — Number of characters**
`int` | `integer*4`

Number of characters printed including characters specified with backslash codes, such as `\n` and `\b`, returned as `int` in C or `integer*4` in Fortran.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- `mexfunction.c`

See these examples in *matlabroot*/extern/examples/refbook:

- `phonebook.c`

## See Also

`mexErrMsgIdAndTxt` | `mexWarnMsgIdAndTxt` | `sprintf`

# mexPutVariable (C)

Array from MEX function into specified workspace

## C Syntax

```
#include "mex.h"
int mexPutVariable(const char *workspace, const char *varname,
  const mxArray *pm);
```

## Description

**Note** To write MEX functions using modern C++ features and the “MATLAB Data API”, see “C++ MEX Applications”.

Call `mexPutVariable` to copy the `mxArray`, at pointer `pm`, from your MEX function into the specified workspace. MATLAB assigns `varname` to the `mxArray` copied in the workspace.

`mexPutVariable` makes the array accessible to other entities, such as MATLAB, user-defined functions, or other MEX functions.

If a variable of the same name exists in the specified workspace, `mexPutVariable` overwrites the previous contents of the variable with the contents of the new `mxArray`. For example, suppose the MATLAB workspace defines variable `Peaches` as:

```
Peaches
1     2     3     4
```

and you call `mexPutVariable` to copy `Peaches` into the same workspace:

```
mexPutVariable("base", "Peaches", pm)
```

The value passed by `mexPutVariable` replaces the old value of `Peaches`.

## Input Arguments

**`workspace` — Array scope**
`const char*`

Scope of the array to copy, specified as `const char*`. The possible values are:

| | |
|---|---|
| `base` | Copy `mxArray` to the base workspace. |
| `caller` | Copy `mxArray` to the caller workspace. |
| `global` | Copy `mxArray` to the list of global variables. |

**`varname` — Variable name**
`const char*`

Name of `mxArray` in the workspace, specified as `const char*`.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, or `path`. To determine whether a particular name is associated with a MATLAB function, use the `which` function.

**pm — MATLAB array**
`const mxArray*`

Pointer to the `mxArray`.

## Output Arguments

**`status` — Status**
`int`

Status, returned as `0` on success. Returns `1` on failure. A possible cause of failure is that `pm` is `NULL`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexgetarray.c`

## See Also

`mexGetVariable`

**Topics**
"matlab::engine::MATLABEngine::setVariable"

**Introduced before R2006a**

# mexPutVariable (Fortran)

Array from MEX function into specified workspace

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mexPutVariable(workspace, varname, pm)
character*(*) workspace, varname
mwPointer pm
```

## Arguments

`workspace`

Specifies scope of the array you are copying. Values for `workspace` are:

| | |
|---|---|
| `base` | Copy `mxArray` to the base workspace. |
| `caller` | Copy `mxArray` to the caller workspace. |
| `global` | Copy `mxArray` to the list of global variables. |

`varname`

Name of `mxArray` in the workspace

`pm`

Pointer to the `mxArray`

## Returns

`0` on success; `1` on failure. A possible cause of failure is that `pm` is `0`.

## Description

Call `mexPutVariable` to copy the `mxArray`, at pointer `pm`, from your MEX function into the specified workspace. MATLAB software gives the name, `varname`, to the copied `mxArray` in the receiving workspace.

`mexPutVariable` makes the array accessible to other entities, such as MATLAB, user-defined functions, or other MEX functions.

If a variable of the same name exists in the specified workspace, `mexPutVariable` overwrites the previous contents of the variable with the contents of the new `mxArray`. For example, suppose the MATLAB workspace defines variable `Peaches` as:

```
Peaches
1      2      3      4
```

and you call `mexPutVariable` to copy `Peaches` into the same workspace:

```
mexPutVariable("base", "Peaches", pm)
```

The value passed by mexPutVariable replaces the old value of Peaches.

Do not use MATLAB function names for variable names. Common variable names that conflict with function names include i, j, mode, char, size, or path. To determine whether a particular name is associated with a MATLAB function, use the which function.

## See Also

mexGetVariable

**Introduced before R2006a**

# mexSet (C)

Set value of specified graphics property

---

**Note** Do not use `mexSet`. Use `mxSetProperty` instead.

---

## C Syntax

```
#include "mex.h"
int mexSet(double handle, const char *property,
  mxArray *value);
```

## Description

Call `mexSet` to set the value of the property of a certain graphics object. `mexSet` is the API equivalent of the MATLAB `set` function. To get the value of a graphics property, call `mexGet`.

## Input Arguments

**`handle` — Graphics object handle**
`double`

Graphics object handle, specified as `double`.

**`property` — Graphics property name**
`const char*`

Graphics property name, specified as `const char*`.

**`value` — Property value**
`mxArray*`

Property value, specified as a pointer to an `mxArray`.

## Output Arguments

**`status` — Status**
`int`

Status, returned as `0` on success. Returns `1` on failure. Possible causes of failure include:

- Specifying a nonexistent property.
- Specifying an illegal value for that property, for example, specifying a string value for a numerical property.

## See Also

`mxGetProperty`, `mxSetProperty`

# mexSetTrapFlag (C and Fortran)

(Removed) Control response of MEXCALLMATLAB to errors

**Note** `mexSetTrapFlag` has been removed. Use `mexCallMATLABWithTrap` instead. For more information, see “Compatibility Considerations”.

## C Syntax

```
#include "mex.h"
void mexSetTrapFlag(int trapflag);
```

## Fortran Syntax

```
subroutine mexSetTrapFlag(trapflag)
integer*4 trapflag
```

## Arguments

`trapflag`

Control flag.

- `0` - On error, control returns to the MATLAB prompt.
- `1` - On error, control returns to your MEX file.

## Description

Call `mexSetTrapFlag` to control the MATLAB response to errors in `mexCallMATLAB`.

If you do not call `mexSetTrapFlag`, then whenever MATLAB detects an error in a call to `mexCallMATLAB`, MATLAB automatically terminates the MEX file and returns control to the MATLAB prompt. Calling `mexSetTrapFlag` with `trapflag` set to `0` is equivalent to not calling `mexSetTrapFlag` at all.

If you call `mexSetTrapFlag` and set the `trapflag` to `1`, then whenever MATLAB detects an error in a call to `mexCallMATLAB`, MATLAB does not automatically terminate the MEX file. Rather, MATLAB returns control to the line in the MEX file immediately following the call to `mexCallMATLAB`. The MEX file is then responsible for taking an appropriate response to the error.

If you call `mexSetTrapFlag`, the value of the `trapflag` you set remains in effect until the next call to `mexSetTrapFlag` within that MEX file or, if there are no more calls to `mexSetTrapFlag`, until the MEX file exits. If a routine defined in a MEX file calls another MEX file, MATLAB:

1. Saves the current value of the `trapflag` in the first MEX file.
2. Calls the second MEX file with the `trapflag` initialized to `0` within that file.
3. Restores the saved value of `trapflag` in the first MEX file when the second MEX file exits.

## Compatibility Considerations

**`mexSetTrapFlag` has been removed**
*Errors starting in R2018a*

The `mexCallMATLABWithTrap` function, similar to `mexCallMATLAB`, lets you call MATLAB functions from within a MEX file. In addition, `mexCallMATLABWithTrap` lets you catch (trap) errors. Using this function for exception handling is more flexible that using `mexCallMATLAB` with the `mexSetTrapFlag` function.

Existing MEX files built with `mexSetTrapFlag` continue to run.

## See Also
`mexCallMATLABWithTrap`

**Introduced in R2008b**

# mexUnlock (C and Fortran)

Allow clearing MEX file from memory

## C Syntax

```
#include "mex.h"
void mexUnlock(void);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mexUnlock()
```

## Description

By default, MEX files are unlocked, meaning you can clear them at any time. Calling `mexLock` locks a MEX file so that you cannot clear it from memory. Call `mexUnlock` to remove the lock.

`mexLock` increments a lock count. If you called `mexLock` `n` times, call `mexUnlock` `n` times to unlock your MEX file.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexlock.c`
- `mexlockf.F`

## See Also

`mexIsLocked`, `mexLock`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `clear`

# mexWarnMsgIdAndTxt (C and Fortran)

Warning message with identifier

## C Syntax

```
#include "mex.h"
void mexWarnMsgIdAndTxt(const char *warningid,
  const char *warningmsg, ...);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mexWarnMsgIdAndTxt(warningid, warningmsg)
character*(*) warningid, warningmsg
```

## Description

The `mexWarnMsgIdAndTxt` function writes a warning message to the MATLAB command prompt. The warnings displayed are the same as warnings issued by the MATLAB `warning` function. To control the information displayed or suppressed, call the `warning` function with the desired settings before calling your MEX file.

Unlike `mexErrMsgIdAndTxt`, calling `mexWarnMsgIdAndTxt` does not terminate the MEX file.

## Input Arguments

**`warningid` — Warning identifier**
`const char*` | `character*(*)`

Warning identifier containing a MATLAB message identifier, specified as `const char*` in C or `character*(*)` in Fortran. For information on creating identifiers, see `MException`.

**`warningmsg` — Warning message**
`const char*` | `character*(*)`

String to display, specified as `const char*` in C or `character*(*)` in Fortran. In C, the function supports either UTF-8 or local code page (LCP) encoding and the string can include conversion specifications, used by the ANSI C `printf` function. The encoding for both the message text and the conversion arguments must be the same.

**`...` — Conversion arguments**
`const char*`

In C, any arguments used in the message. Each argument must have a corresponding conversion specification. Refer to your C documentation for `printf` conversion tables.

## See Also

`mexErrMsgIdAndTxt`, `warning`

**Introduced before R2006a**

# mexWarnMsgTxt (C and Fortran)

Warning message

---

**Note** `mexWarnMsgTxt` is not recommended. Use `mexWarnMsgIdAndTxt` instead.

---

## C Syntax

```
#include "mex.h"
void mexWarnMsgTxt(const char *warningmsg);
```

## Fortran Syntax

```
subroutine mexWarnMsgTxt(warningmsg)
character*(*) warningmsg
```

## Arguments

`warningmsg`

String containing the warning message to display

## Description

`mexWarnMsgTxt` causes MATLAB software to display the contents of `warningmsg`. `mexWarnMsgTxt` does not terminate the MEX-file.

## See Also

`mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt`

# mwIndex (C)

C type for mxArray index values

## Description

`mwIndex` is a type that represents index values, such as indices into arrays. Use this function for cross-platform flexibility. By default, `mwIndex` is equivalent to `size_t` in C.

The C header file containing this type is:

```
#include "matrix.h"
```

## See Also

`mex` | `mwSignedIndex` | `mwSize`

**Topics**

"Create 2-D Cell Array in C MEX File"
"Handling Large mxArrays in C MEX Files"

**Introduced before R2006a**

# mwIndex (Fortran)

Fortran type for mxArray index values

## Description

`mwIndex` is a type that represents index values, such as indices into arrays. Use this function for cross-platform flexibility. By default, `mwIndex` is equivalent to `INTEGER*4` or `INTEGER*8`, based on platform and compilation flags.

In Fortran, `mwIndex` is a preprocessor macro. The Fortran header file containing this type is:

```
#include "fintrf.h"
```

## See Also
`mex` | `mwSignedIndex` | `mwSize`

**Topics**
"Handling Large mxArrays"

**Introduced before R2006a**

# mwPointer (Fortran)

Fortran pointer type

## Description

The `mwPointer` preprocessor macro declares the appropriate Fortran type representing a pointer to an `mxArray`, the fundamental type underlying MATLAB data. The Fortran header file containing this macro is:

```
#include "fintrf.h"
```

The Fortran preprocessor translates `mwPointer` to the Fortran declaration that is appropriate for the platform on which you compile your file. On 64-bit platforms, the Fortran type that represents a pointer is `INTEGER*8`. On 32-bit platforms, the type is `INTEGER*4`. If your Fortran compiler supports preprocessing, you can use `mwPointer` to declare functions, arguments, and variables that represent pointers. If you cannot use `mwPointer`, then ensure that your declarations have the correct size for the platform on which you are compiling Fortran code.

## Examples

This example declares the arguments for `mexFunction` in a Fortran MEX file.

```
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
mwPointer plhs(*), prhs(*)
integer nlhs, nrhs
```

For additional examples, see the Fortran files with names ending in `.F` in the *`matlabroot`*`/extern/examples` folder.

## See Also
`mexFunction`

**Topics**
“Data Types”
“MATLAB Data”

**Introduced in R2006a**

# mwSignedIndex (C)

Signed integer C type for mxArray size values

## Description

`mwSignedIndex` is a signed integer type that represents size values, such as array dimensions. Use this function for cross-platform flexibility. By default, `mwSignedIndex` is equivalent to `ptrdiff_t` in `C++`.

The C header file containing this type is:

```
#include "matrix.h"
```

## See Also

`mwIndex` | `mwSize`

**Introduced in R2009a**

# mwSignedIndex (Fortran)

Signed integer Fortran type for mxArray size values

## Description

`mwSignedIndex` is a signed integer type that represents size values, such as array dimensions. Use this function for cross-platform flexibility. By default, `mwSignedIndex` is equivalent to `INTEGER*4` or `INTEGER*8`, based on platform and compilation flags.

The Fortran header file containing this type is:

```
#include "fintrf.h"
```

## See Also

`mwIndex` | `mwSize`

**Introduced in R2009a**

# mwSize (C)

C type for mxArray size values

## Description

`mwSize` is a type that represents size values, such as array dimensions. Use this function for cross-platform flexibility. By default, `mwSize` is equivalent to `size_t`. `mwSize` is an unsigned type, meaning a nonnegative integer.

When using the `mex -compatibleArrayDims` switch, `mwSize` is equivalent to `int`.

The C header file containing this type is:

```
#include "matrix.h"
```

## See Also

`mex` | `mwIndex` | `mwSignedIndex`

**Introduced before R2006a**

# mwSize (Fortran)

Fortran type for mxArray size values

## Description

`mwSize` is a type that represents size values, such as array dimensions. Use this function for cross-platform flexibility. `mwSize` is an unsigned type, meaning a nonnegative integer.

When using the `mex -compatibleArrayDims` switch, `mwSize` is equivalent to `INTEGER*4` or `INTEGER*8`, based on platform and compilation flags.

In Fortran, `mwSize` is a preprocessor macro. The Fortran header file containing this type is:

```
#include "fintrf.h"
```

## See Also

`mex` | `mwIndex` | `mwSignedIndex`

**Introduced before R2006a**

# mxAddField (C and Fortran)

Add field to structure array

## C Syntax

```
#include "matrix.h"
extern int mxAddField(mxArray *pm, const char *fieldname);
```

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxAddField(pm, fieldname)
mwPointer pm
character*(*) fieldname
```

## Arguments

`pm`

Pointer to a structure `mxArray`

`fieldname`

Name of the field you want to add

## Returns

Field number on success, or `-1` if inputs are invalid or an out-of-memory condition occurs.

## Description

Call `mxAddField` to add a field to a structure array. Create the values with the `mxCreate*` functions and use `mxSetFieldByNumber` to set the individual values for the field.

## See Also

`mxRemoveField`, `mxSetFieldByNumber`

**Introduced before R2006a**

# mxArray (C)

C type for MATLAB array

## Description

The fundamental type underlying MATLAB data. `mxArray` is a C language opaque type. The header file containing this type is:

```
#include "matrix.h"
```

For information on how `mxArray` works with MATLAB-supported variables, see “MATLAB Data”.

All C MEX files start with a gateway routine, called `mexFunction`, which requires `mxArray` for both input and output parameters. For information about the C MEX file gateway routine, see “Components of C MEX File”.

Once you have MATLAB data in your MEX file, use functions in the C Matrix API to manipulate the data and functions in the C MEX API to perform operations in the MATLAB environment. Use `mxArray` to pass data to and from these functions.

## Example

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreatecharmatrixfromstr.c`

## Limitations

- In Simulink S-functions, do not store `plhs` `mxArray` pointers in any S-function block state that persists after the MEX function finishes. An output `mxArray` has temporary scope and is automatically destroyed at the end of the MEX function call.

## Tips

- For information about data in MATLAB language scripts and functions, see “Data Types”.
- For troubleshooting `mxArray` errors in other MathWorks products, search the documentation for that product.

## See Also

`matlab::data::Array` | `mexFunction` | `mxClassID` | `mxCreateDoubleMatrix` | `mxCreateNumericArray` | `mxCreateString` | `mxDestroyArray`

**Topics**
“Components of C MEX File”
“Data Types”
“MATLAB Data”

**Introduced before R2006a**

# mxArrayToString (C)

Array to string

## C Syntax

```
#include "matrix.h"
char *mxArrayToString(const mxArray *array_ptr);
```

## Description

Call `mxArrayToString` to copy the character data of an `mxCHAR` array into a C-style string. The C-style string is always terminated with a `NULL` character and stored in column-major order. If the array contains multiple rows, then the rows are copied column-wise into a single array.

This function is similar to `mxGetString`, except that:

- `mxArrayToString` does not require the length of the string as an input.
- `mxArrayToString` supports both multi-byte and single-byte encoded characters. On Windows and Linux® platforms, the user locale setting specifies the default encoding.

## Input Arguments

**`array_ptr` — Pointer to `mxCHAR` array**
`const mxArray *`

Pointer to `mxCHAR` array, specified as `const mxArray *`.

## Output Arguments

**`str` — C-style string**
`char *` | `NULL`

C-style string in local code page (LCP) encoding, specified as `char *`. To convert an array to a string in UTF-8 encoding, use `mxArrayToUTF8String`.

Returns `NULL` on failure. Possible reasons for failure include out of memory and specifying an array that is not an `mxCHAR` array.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexatexit.c`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreatecharmatrixfromstr.c`

## See Also

`mxArrayToUTF8String` | `mxCreateCharArray` | `mxCreateCharMatrixFromStrings` | `mxCreateString` | `mxGetString`

**Introduced before R2006a**

# mxArrayToUTF8String (C)

Array to string in UTF-8 encoding

## C Syntax

```
#include "matrix.h"
char *mxArrayToUTF8String(const mxArray *array_ptr);
```

## Arguments

`array_ptr`

Pointer to `mxCHAR` array.

## Returns

C-style string in UTF-8 encoding. Returns `NULL` on failure. Possible reasons for failure include out of memory and specifying an array that is not an `mxCHAR` array.

## Description

Call `mxArrayToUTF8String` to copy the character data of an `mxCHAR` array into a C-style string. The data is stored in column-major order. If the array contains multiple rows, the rows are copied column-wise into a single array.

## See Also

`mxArrayToString`, `mxFree`, `mxCreateCharArray`, `mxCreateString`, `mxGetString`

**Introduced in R2015a**

# mxAssert (C)

Check assertion value for debugging purposes

## C Syntax

```
#include "matrix.h"
void mxAssert(int expr, char *error_message);
```

## Arguments

`expr`

Value of assertion

`error_message`

Description of why assertion failed

## Description

Like the ANSI C `assert` macro, `mxAssert` checks the value of an assertion, and continues execution only if the assertion holds. If `expr` evaluates to logical `1` (`true`), `mxAssert` does nothing. If `expr` evaluates to logical `0` (`false`), `mxAssert` terminates the MEX file and prints an error to the MATLAB command window. The error contains the expression of the failed assertion, the file name, and line number where the failed assertion occurred, and the `error_message` text. The `error_message` allows you to specify a better description of why the assertion failed. Use an empty string if you do not want a description to follow the failed assertion message.

The `mex` script turns off these assertions when building optimized MEX functions, so use assertions for debugging purposes only. To use `mxAssert`, build the MEX file using the `mex -g filename` syntax.

Assertions are a way of maintaining internal consistency of logic. Use them to keep yourself from misusing your own code and to prevent logical errors from propagating before they are caught. Do not use assertions to prevent users of your code from misusing it.

Assertions can be taken out of your code by the C preprocessor. You can use these checks during development and then remove them when the code works properly. Use assertions for troubleshooting during development without slowing down the final product.

## See Also

`mxAssertS`, `mexErrMsgIdAndTxt`

**Introduced before R2006a**

# mxAssertS (C)

Check assertion value without printing assertion text

## C Syntax

```
#include "matrix.h"
void mxAssertS(int expr, char *error_message);
```

## Arguments

`expr`

Value of assertion

`error_message`

Description of why assertion failed

## Description

`mxAssertS` is like `mxAssert`, except `mxAssertS` does not print the text of the failed assertion.

## See Also

`mxAssert`

**Introduced before R2006a**

# mxCalcSingleSubscript (C)

Offset from first element to desired element

## C Syntax

```
#include "matrix.h"
mwIndex mxCalcSingleSubscript(const mxArray *pm, mwSize nsubs, mwIndex *subs);
```

## Description

Call `mxCalcSingleSubscript` to determine how many elements there are between the beginning of the `mxArray` and a given element of that `mxArray`. The function converts subscripts to linear indices.

For example, given a subscript like `(5,7)`, `mxCalcSingleSubscript` returns the distance from the first element of the array to the `(5,7)` element. Remember that the `mxArray` data type internally represents all data elements in a one-dimensional array no matter how many dimensions the MATLAB `mxArray` appears to have. For examples showing the internal representation, see “Data Storage”.

Avoid using `mxCalcSingleSubscript` to traverse the elements of an array. In C, it is more efficient to find the starting address of the array and then use pointer autoincrementing to access successive elements. For example, to find the starting address of a numerical array, call one of the typed data access functions, for example, `mxGetDoubles` or `mxGetComplexDoubles`.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

**`nsubs` — Number of elements**
`mwSize`

Number of elements in the `subs` array, specified as `mwSize`. Typically, you set `nsubs` equal to the number of dimensions in the `mxArray` that `pm` points to.

**`subs` — Array of subscripts**
`mwIndex`

Array of subscripts, specified as `mwIndex`. Each value in the array specifies that dimension's subscript. The value in `subs(1)` specifies the row subscript, and the value in `subs(2)` specifies the column subscript. Use 1-based indexing for subscripts. For example, to express the starting element of a two-dimensional `mxArray` in `subs`, set `subs(1)` to `1` and `subs(2)` to `1`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcalcsinglesubscript.c`

## See Also

`mxGetCell` | `mxSetCell`

**Introduced before R2006a**

# mxCalcSingleSubscript (Fortran)

Offset from first element to desired element

## Fortran Syntax

```
#include "fintrf.h"
mwIndex mxCalcSingleSubscript(pm, nsubs, subs)
mwPointer pm
mwSize nsubs
mwIndex subs
```

## Description

Call `mxCalcSingleSubscript` to determine how many elements there are between the beginning of the `mxArray` and a given element of that `mxArray`. The function converts subscripts to linear indices.

For example, given a subscript like `(5,7)`, `mxCalcSingleSubscript` returns the distance from the first element of the array to the `(5,7)` element. Remember that the `mxArray` data type internally represents all data elements in a one-dimensional array no matter how many dimensions the MATLAB `mxArray` appears to have. For examples showing the internal representation, see "Data Storage".

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

**nsubs — Number of elements**
`mwSize`

Number of elements in the `subs` array, specified as `mwSize`. Typically, you set `nsubs` equal to the number of dimensions in the `mxArray` that `pm` points to.

**subs — Array of subscripts**
`mwIndex`

Array of subscripts, specified as `mwIndex`. Each value in the array specifies that dimension's subscript. The value in `subs(1)` specifies the row subscript, and the value in `subs(2)` specifies the column subscript. Use 1-based indexing for subscripts. For example, to express the starting element of a two-dimensional `mxArray` in `subs`, set `subs(1)` to `1` and `subs(2)` to `1`.

## Returns

The number of elements, or index, between the start of the `mxArray` and the specified subscript. This number is the linear index equivalent of the subscripts. Many Matrix Library routines (for example, `mxGetField`) require an index as an argument.

If `subs` describes the starting element of an `mxArray`, `mxCalcSingleSubscript` returns 0. If `subs` describes the final element of an `mxArray`, `mxCalcSingleSubscript` returns `N-1` (where `N` is the total number of elements).

## See Also

`mxGetCell` | `mxSetCell`

**Introduced before R2006a**

# mxCalloc (C and Fortran)

Allocate dynamic memory for array, initialized to 0, using MATLAB memory manager

## C Syntax

```
#include "matrix.h"
#include <stdlib.h>
void *mxCalloc(mwSize n, mwSize size);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCalloc(n, size)
mwSize n, size
```

## Arguments

`n`

Number of elements to allocate. This must be a nonnegative number.

`size`

Number of bytes per element. (The C `sizeof` operator calculates the number of bytes per element.)

## Returns

Pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a MAT or engine standalone application, `mxCalloc` returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX file, the MEX file terminates and control returns to the MATLAB prompt.

`mxCalloc` is unsuccessful when there is insufficient free heap space.

## Description

`mxCalloc` allocates contiguous heap space sufficient to hold `n` elements of `size` bytes each, and initializes this newly allocated memory to `0`. To allocate memory in MATLAB applications, use `mxCalloc` instead of the ANSI C `calloc` function.

In MEX files, but not MAT or engine applications, `mxCalloc` registers the allocated memory with the MATLAB memory manager. When control returns to the MATLAB prompt, the memory manager then automatically frees, or deallocates, this memory.

How you manage the memory created by this function depends on the purpose of the data assigned to it. If you assign it to an output argument in `plhs[]` using a function such as `mxSetDoubles`, then MATLAB is responsible for freeing the memory.

If you use the data internally, then the MATLAB memory manager maintains a list of all memory allocated by the function and automatically frees (deallocates) the memory when control returns to the MATLAB prompt. In general, we recommend that MEX file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX file than to rely on the automatic mechanism. Therefore, when you finish using the memory allocated by this function, call `mxFree` to deallocate the memory.

If you do not assign this data to an output argument, and you want it to persist after the MEX file completes, then call `mexMakeMemoryPersistent` after calling this function. If you write a MEX file with persistent memory, then be sure to register a `mexAtExit` function to free allocated memory in the event your MEX file is cleared.

## Examples

See these examples in *matlabroot*`/extern/examples/mex`:

- `explore.c`

See these examples in *matlabroot*`/extern/examples/refbook`:

- `arrayFillSetData.c`
- `phonebook.c`
- `revord.c`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxcalcsinglesubscript.c`
- `mxsetdimensions.c`

## See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxDestroyArray`, `mxFree`, `mxMalloc`, `mxRealloc`

**Introduced before R2006a**

# mxChar (C)

Type for string array

## Description

MATLAB stores an `mxArray` string as type `mxChar` to represent the C-style `char` type. MATLAB uses 16-bit unsigned integer character encoding for Unicode characters.

The header file containing this type is:

```
#include "matrix.h"
```

## Examples

See these examples in *matlabroot*/extern/examples/mx:

- `mxmalloc.c`
- `mxcreatecharmatrixfromstr.c`

See these examples in *matlabroot*/extern/examples/mex:

- `explore.c`

## See Also

`mxCreateCharArray`

## Tips

- For information about data in MATLAB language scripts and functions, see “Data Types”.

**Introduced before R2006a**

# mxClassID

Enumerated value identifying class of array

## C Syntax

```
typedef enum {
        mxUNKNOWN_CLASS,
        mxCELL_CLASS,
        mxSTRUCT_CLASS,
        mxLOGICAL_CLASS,
        mxCHAR_CLASS,
        mxVOID_CLASS,
        mxDOUBLE_CLASS,
        mxSINGLE_CLASS,
        mxINT8_CLASS,
        mxUINT8_CLASS,
        mxINT16_CLASS,
        mxUINT16_CLASS,
        mxINT32_CLASS,
        mxUINT32_CLASS,
        mxINT64_CLASS,
        mxUINT64_CLASS,
        mxFUNCTION_CLASS
} mxClassID;
```

## Description

Various C Matrix API functions require or return an `mxClassID` argument. `mxClassID` identifies how the `mxArray` represents its data elements.

## Constants

### mxUNKNOWN_CLASS

Undetermined class. You cannot specify this category for an `mxArray`. However, if `mxGetClassID` cannot identify the class, it returns this value.

### mxCELL_CLASS

Cell `mxArray`.

### mxSTRUCT_CLASS

Structure `mxArray`.

### mxLOGICAL_CLASS

Logical `mxArray` of `mxLogical` data.

### mxCHAR_CLASS

String `mxArray` of `mxChar` data.

### mxVOID_CLASS

Reserved.

### mxDOUBLE_CLASS

Numeric `mxArray` of either real or complex data types.

| | |
|---|---|
| MATLAB Type | `double` |
| C Real Data Type | `typedef double mxDouble;` |
| C Complex Data Type | `typedef struct { mxDouble real, imag; } mxComplexDouble;` |

### mxSINGLE_CLASS

Numeric `mxArray` of either real or complex data types.

| | |
|---|---|
| MATLAB Type | `single` |
| C Real Data Type | `typedef float mxSingle;` |
| C Complex Data Type | `typedef struct { mxSingle real, imag; } mxComplexSingle;` |

### mxINT8_CLASS

Numeric `mxArray` of either real or complex data types.

| | |
|---|---|
| MATLAB Type | `int8` |
| C Real Data Type | `typedef int8_T mxInt8;` |
| C Complex Data Type | `typedef struct { mxInt8 real, imag; } mxComplexInt8;` |

### mxUINT8_CLASS

Numeric `mxArray` of either real or complex data types.

| | |
|---|---|
| MATLAB Type | `uint8` |
| C Real Data Type | `typedef uint8_T mxUint8;` |
| C Complex Data Type | `typedef struct { mxUint8 real, imag; } mxComplexUint8;` |

### mxINT16_CLASS

Numeric `mxArray` of either real or complex data types.

| | |
|---|---|
| MATLAB Type | `int16` |
| C Real Data Type | `typedef int16_T mxInt16;` |
| C Complex Data Type | `typedef struct { mxInt16 real, imag; } mxComplexInt16;` |

### mxUINT16_CLASS

Numeric `mxArray` of either real or complex data types.

| | |
|---|---|
| MATLAB Type | `uint16` |
| C Real Data Type | `typedef uint16_T mxUint16;` |

| | |
|---|---|
| C Complex Data Type | `typedef struct { mxUint16 real, imag; } mxComplexUint16;` |

**mxINT32_CLASS**

Numeric `mxArray` of either real or complex data types.

| | |
|---|---|
| MATLAB Type | `int32` |
| C Real Data Type | `typedef int32_T mxInt32;` |
| C Complex Data Type | `typedef struct { mxInt32 real, imag; } mxComplexInt32;` |

**mxUINT32_CLASS**

Numeric `mxArray` of either real or complex data types.

| | |
|---|---|
| MATLAB Type | `uint32` |
| C Real Data Type | `typedef uint32_T mxUint32;` |
| C Complex Data Type | `typedef struct { mxUint32 real, imag; } mxComplexUint32;` |

**mxINT64_CLASS**

Numeric `mxArray` of either real or complex data types.

| | |
|---|---|
| MATLAB Type | `int64` |
| C Real Data Type | `typedef int64_T mxInt64;` |
| C Complex Data Type | `typedef struct { mxInt64 real, imag; } mxComplexInt64;` |

**mxUINT64_CLASS**

Numeric `mxArray` of either real or complex data types.

| | |
|---|---|
| MATLAB Type | `uint64` |
| C Real Data Type | `typedef uint64_T mxUint64;` |
| C Complex Data Type | `typedef struct { mxUint64 real, imag; } mxComplexUint64;` |

**mxFUNCTION_CLASS**

Identifies a function handle `mxArray`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## See Also

`matlab::data::ArrayType` | `mxCreateNumericArray` | `mxGetClassID`

**Introduced before R2006a**

# mxClassIDFromClassName (Fortran)

Identifier corresponding to class

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxClassIDFromClassName(classname)
character*(*) classname
```

## Arguments

`classname`

`character` array specifying a MATLAB class name. For a list of valid `classname` choices, see the `mxIsClass` reference page.

## Returns

Numeric identifier used internally by MATLAB software to represent the MATLAB class, `classname`. Returns `unknown` if `classname` is not a recognized MATLAB class.

## Description

Use `mxClassIDFromClassName` to obtain an identifier for any MATLAB class. This function is most commonly used to provide a `classid` argument to `mxCreateNumericArray` and `mxCreateNumericMatrix`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `matsqint8.F`

## See Also

`mxCreateNumericArray` | `mxCreateNumericMatrix` | `mxGetClassName` | `mxIsClass`

**Introduced before R2006a**

# mxComplexity (C)

Flag specifying whether array has imaginary components

## C Syntax

```
typedef enum mxComplexity {mxREAL=0, mxCOMPLEX};
```

## Constants

`mxREAL`

Identifies an `mxArray` with no imaginary components.

`mxCOMPLEX`

Identifies an `mxArray` with imaginary components.

## Description

Various Matrix Library functions require an `mxComplexity` argument. You can set an `mxComplex` argument to either `mxREAL` or `mxCOMPLEX`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcalcsinglesubscript.c`

## See Also

`mxCreateNumericArray`, `mxCreateDoubleMatrix`, `mxCreateSparse`

**Introduced before R2006a**

# mxCopyCharacterToPtr (Fortran)

CHARACTER values from Fortran array to pointer array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyCharacterToPtr(y, px, n)
character*(*) y
mwPointer px
mwSize n
```

## Arguments

`y`

`character` Fortran array

`px`

Pointer to `character` or `name` array

`n`

Number of elements to copy

## Description

`mxCopyCharacterToPtr` copies `n` `character` values from the Fortran `character` array `y` into the MATLAB character vector pointed to by `px`. This subroutine is essential for copying character data between MATLAB pointer arrays and ordinary Fortran `character` arrays.

## See Also

`mxCopyPtrToCharacter` | `mxCreateCharArray` | `mxCreateCharMatrixFromStrings` | `mxCreateString`

**Introduced before R2006a**

# mxCopyComplex16ToPtr (Fortran)

`COMPLEX*16` values from Fortran array to pointer array

**Note** The function signature for `mxCopyComplex16ToPtr` is different in the Interleaved Complex API.

## Fortran Syntax

Interleaved complex API

```
#include "fintrf.h"
integer*4 mxCopyComplex16ToPtr(y, pd, n)
complex*16 y(n)
mwPointer pd
mwSize n
```

Separate complex API

```
#include "fintrf.h"
subroutine mxCopyComplex16ToPtr(y, pr, pi, n)
complex*16 y(n)
mwPointer pr, pi
mwSize n
```

## Input Arguments

`y`
: `COMPLEX*16` Fortran array

`pd`
: Pointer to a complex double-precision MATLAB array

`pr`
: Pointer to the real data of a double-precision MATLAB array

`pi`
: Pointer to the imaginary data of a double-precision MATLAB array

`n`
: Number of elements to copy

## Output Arguments

`status`
: Function status, returned as `integer*4` when using the interleaved complex API.

## Description

`mxCopyComplex16ToPtr` copies `n` `COMPLEX*16` values from the Fortran `COMPLEX*16` array `y` into the MATLAB array pointed to by:

- `pd` when using the interleaved complex API, built with the `-R2018a` option.
- `pr` and `pi` when using the separate complex API, built with the `-R2017b` option.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `convec.F`
- `complexAdd.F`

## See Also

`mxCopyPtrToComplex16` | `mxCreateNumericArray` | `mxCreateNumericMatrix`

**Introduced before R2006a**

# mxCopyComplex8ToPtr (Fortran)

`COMPLEX*8` values from Fortran array to pointer array

**Note** The function signature for `mxCopyComplex8ToPtr` is different in the Interleaved Complex API.

## Fortran Syntax

Interleaved complex API

```
#include "fintrf.h"
integer*4 mxCopyComplex8ToPtr(y, pd, n)
complex*8 y(n)
mwPointer pd
mwSize n
```

Separate complex API

```
#include "fintrf.h"
subroutine mxCopyComplex8ToPtr(y, pr, pi, n)
complex*8 y(n)
mwPointer pr, pi
mwSize n
```

## Input Arguments

`y`

`COMPLEX*8` Fortran array

`pd`

Pointer to a complex double-precision MATLAB array

`pr`

Pointer to the real data of a single-precision MATLAB array

`pi`

Pointer to the imaginary data of a single-precision MATLAB array

`n`

Number of elements to copy

## Output Arguments

`status`

Function status, returned as `integer*4` when using the interleaved complex API.

## Description

`mxCopyComplex8ToPtr` copies `n` `COMPLEX*8` values from the Fortran `COMPLEX*8` array `y` into the MATLAB arrays pointed to by:

- `pd` when using the interleaved complex API, built with the `-R2018a` option.
- `pr` and `pi` when using the separate complex API, built with the `-R2017b` option.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

## See Also

`mxCopyPtrToComplex8` | `mxCreateNumericArray` | `mxCreateNumericMatrix` | `mxGetData` | `mxGetImagData`

**Introduced before R2006a**

# mxCopyInteger1ToPtr (Fortran)

INTEGER*1 values from Fortran array to pointer array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyInteger1ToPtr(y, px, n)
integer*1 y(n)
mwPointer px
mwSize n
```

## Arguments

y

INTEGER*1 Fortran array

px

Pointer to the real or imaginary data of the array

n

Number of elements to copy

## Description

mxCopyInteger1ToPtr copies n INTEGER*1 values from the Fortran INTEGER*1 array y into the MATLAB array pointed to by px, either a real or an imaginary array.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

## Examples

See these examples in *matlabroot*/extern/examples/refbook:

- matsqint8.F

## See Also

mxCopyPtrToInteger1 | mxCreateNumericArray | mxCreateNumericMatrix

**Introduced before R2006a**

# mxCopyInteger2ToPtr (Fortran)

INTEGER*2 values from Fortran array to pointer array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyInteger2ToPtr(y, px, n)
integer*2 y(n)
mwPointer px
mwSize n
```

## Arguments

y

INTEGER*2 Fortran array

px

Pointer to the real or imaginary data of the array

n

Number of elements to copy

## Description

mxCopyInteger2ToPtr copies n INTEGER*2 values from the Fortran INTEGER*2 array y into the MATLAB array pointed to by px, either a real or an imaginary array.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

## See Also

mxCopyPtrToInteger2 | mxCreateNumericArray | mxCreateNumericMatrix

**Introduced before R2006a**

# mxCopyInteger4ToPtr (Fortran)

INTEGER*4 values from Fortran array to pointer array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyInteger4ToPtr(y, px, n)
integer*4 y(n)
mwPointer px
mwSize n
```

## Arguments

`y`

INTEGER*4 Fortran array

`px`

Pointer to the real or imaginary data of the array

`n`

Number of elements to copy

## Description

`mxCopyInteger4ToPtr` copies `n` `INTEGER*4` values from the Fortran `INTEGER*4` array `y` into the MATLAB array pointed to by `px`, either a real or an imaginary array.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

## See Also

`mxCopyPtrToInteger4` | `mxCreateNumericArray` | `mxCreateNumericMatrix`

**Introduced before R2006a**

# mxCopyPtrToCharacter (Fortran)

CHARACTER values from pointer array to Fortran array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToCharacter(px, y, n)
mwPointer px
character*(*) y
mwSize n
```

## Arguments

`px`

Pointer to `character` or `name` array

`y`

`character` Fortran array

`n`

Number of elements to copy

## Description

`mxCopyPtrToCharacter` copies `n` `character` values from the MATLAB array pointed to by `px` into the Fortran `character` array `y`. This subroutine is essential for copying character data from MATLAB pointer arrays into ordinary Fortran `character` arrays.

## Examples

See these examples in *matlabroot*`/extern/examples/eng_mat`:

- `matdemo2.F`

## See Also

`mxCopyCharacterToPtr` | `mxCreateCharArray` | `mxCreateCharMatrixFromStrings` | `mxCreateString`

**Introduced before R2006a**

# mxCopyPtrToComplex16 (Fortran)

COMPLEX*16 values from pointer array to Fortran array

**Note** The function signature for mxCopyPtrToComplex16 is different in the Interleaved Complex API.

## Fortran Syntax

Interleaved complex API

```
#include "fintrf.h"
integer*4 mxCopyPtrToComplex16(pd, y, n)
mwPointer pd
complex*16 y(n)
mwSize n
```

Separate complex API

```
#include "fintrf.h"
subroutine mxCopyPtrToComplex16(pr, pi, y, n)
mwPointer pr, pi
complex*16 y(n)
mwSize n
```

## Input Arguments

pd

Pointer to a complex double-precision MATLAB array

pr

Pointer to the real data of a double-precision MATLAB array

pi

Pointer to the imaginary data of a double-precision MATLAB array

y

COMPLEX*16 Fortran array

n

Number of elements to copy

## Output Arguments

status

Function status, returned as integer*4 when using the interleaved complex API.

## Description

mxCopyPtrToComplex16 copies n COMPLEX*16 values from the specified MATLAB arrays into the Fortran COMPLEX*16 array y. The MATLAB arrays are pointed to by:

- `pd` when using the interleaved complex API, built with the `-R2018a` option.
- `pr` and `pi` when using the separate complex API, built with the `-R2017b` option.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `convec.F`
- `complexAdd.F`

## See Also

`mxCopyComplex16ToPtr` | `mxCreateNumericArray` | `mxCreateNumericMatrix`

**Introduced before R2006a**

# mxCopyPtrToComplex8 (Fortran)

COMPLEX*8 values from pointer array to Fortran array

**Note** The function signature for mxCopyPtrToComplex8 is different in the Interleaved Complex API.

## Fortran Syntax

Interleaved complex API

```
#include "fintrf.h"
integer*4 mxCopyPtrToComplex8(pd, y, n)
mwPointer pd
complex*8 y(n)
mwSize n
```

Separate complex API

```
#include "fintrf.h"
subroutine mxCopyPtrToComplex8(pr, pi, y, n)
mwPointer pr, pi
complex*8 y(n)
mwSize n
```

## Input Arguments

pd

Pointer to a complex double-precision MATLAB array

pr

Pointer to the real data of a single-precision MATLAB array

pi

Pointer to the imaginary data of a single-precision MATLAB array

y

COMPLEX*8 Fortran array

n

Number of elements to copy

## Output Arguments

status

Function status, returned as integer*4 when using the interleaved complex API.

## Description

mxCopyPtrToComplex8 copies n COMPLEX*8 values from the specified MATLAB arrays into the Fortran COMPLEX*8 array y. The MATLAB arrays are pointed to by:

- `pd` when using the interleaved complex API, built with the `-R2018a` option.
- `pr` and `pi` when using the separate complex API, built with the `-R2017b` option.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

## See Also

`mxCopyComplex8ToPtr` | `mxCreateNumericArray` | `mxCreateNumericMatrix` | `mxGetData` | `mxGetImagData`

**Introduced before R2006a**

# mxCopyPtrToInteger1 (Fortran)

`INTEGER*1` values from pointer array to Fortran array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToInteger1(px, y, n)
mwPointer px
integer*1 y(n)
mwSize n
```

## Arguments

`px`

Pointer to the real or imaginary data of the array

`y`

`INTEGER*1` Fortran array

`n`

Number of elements to copy

## Description

`mxCopyPtrToInteger1` copies `n` `INTEGER*1` values from the MATLAB array pointed to by `px`, either a real or imaginary array, into the Fortran `INTEGER*1` array `y`.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matsqint8.F`

## See Also

`mxCopyInteger1ToPtr` | `mxCreateNumericArray` | `mxCreateNumericMatrix`

**Introduced before R2006a**

# mxCopyPtrToInteger2 (Fortran)

INTEGER*2 values from pointer array to Fortran array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToInteger2(px, y, n)
mwPointer px
integer*2 y(n)
mwSize n
```

## Arguments

px

Pointer to the real or imaginary data of the array

y

INTEGER*2 Fortran array

n

Number of elements to copy

## Description

mxCopyPtrToInteger2 copies n INTEGER*2 values from the MATLAB array pointed to by px, either a real or an imaginary array, into the Fortran INTEGER*2 array y.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

## See Also

mxCopyInteger2ToPtr | mxCreateNumericArray | mxCreateNumericMatrix

**Introduced before R2006a**

# mxCopyPtrToInteger4 (Fortran)

INTEGER*4 values from pointer array to Fortran array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToInteger4(px, y, n)
mwPointer px
integer*4 y(n)
mwSize n
```

## Arguments

px

Pointer to the real or imaginary data of the array

y

INTEGER*4 Fortran array

n

Number of elements to copy

## Description

mxCopyPtrToInteger4 copies n INTEGER*4 values from the MATLAB array pointed to by px, either a real or an imaginary array, into the Fortran INTEGER*4 array y.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

## See Also

mxCopyInteger4ToPtr | mxCreateNumericArray | mxCreateNumericMatrix

**Introduced before R2006a**

# mxCopyPtrToPtrArray (Fortran)

Pointer values from pointer array to Fortran array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToPtrArray(px, y, n)
mwPointer px
mwPointer y(n)
mwSize n
```

## Arguments

`px`

Pointer to pointer array

`y`

Fortran array of `mwPointer` values

`n`

Number of pointers to copy

## Description

`mxCopyPtrToPtrArray` copies `n` pointers from the MATLAB array pointed to by `px` into the Fortran array `y`. This subroutine is essential for copying the output of `matGetDir` into an array of pointers. After calling this function, each element of `y` contains a pointer to a string. You can convert these strings to Fortran character arrays by passing each element of `y` as the first argument to `mxCopyPtrToCharacter`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matdemo2.F`

## See Also

`matGetDir` | `mxCopyPtrToCharacter`

**Introduced before R2006a**

# mxCopyPtrToReal4 (Fortran)

REAL*4 values from pointer array to Fortran array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToReal4(px, y, n)
mwPointer px
real*4 y(n)
mwSize n
```

## Arguments

px

Pointer to the real or imaginary data of a single-precision MATLAB array

y

REAL*4 Fortran array

n

Number of elements to copy

## Description

mxCopyPtrToReal4 copies n REAL*4 values from the MATLAB array pointed to by px, either a pr or pi array, into the Fortran REAL*4 array y.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

## See Also

mxCopyReal4ToPtr | mxCreateNumericArray | mxCreateNumericMatrix | mxGetData | mxGetImagData

**Introduced before R2006a**

# mxCopyPtrToReal8 (Fortran)

REAL*8 values from pointer array to Fortran array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyPtrToReal8(px, y, n)
mwPointer px
real*8 y(n)
mwSize n
```

## Arguments

`px`

Pointer to the real or imaginary data of a double-precision MATLAB array

`y`

`REAL*8` Fortran array

`n`

Number of elements to copy

## Description

`mxCopyPtrToReal8` copies `n` `REAL*8` values from the MATLAB array pointed to by `px`, either a `pr` or `pi` array, into the Fortran `REAL*8` array `y`.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the `%VAL` construct.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `fengdemo.F`

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `timestwo.F`
- `xtimesy.F`

## See Also

`mxCopyReal8ToPtr` | `mxCreateNumericArray` | `mxCreateNumericMatrix` | `mxGetData` | `mxGetImagData`

**Introduced before R2006a**

# mxCopyReal4ToPtr (Fortran)

REAL*4 values from Fortran array to pointer array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyReal4ToPtr(y, px, n)
real*4 y(n)
mwPointer px
mwSize n
```

## Arguments

y

REAL*4 Fortran array

px

Pointer to the real or imaginary data of a single-precision MATLAB array

n

Number of elements to copy

## Description

mxCopyReal4ToPtr copies n REAL*4 values from the Fortran REAL*4 array y into the MATLAB array pointed to by px, either a pr or pi array.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

## See Also

mxCopyPtrToReal4 | mxCreateNumericArray | mxCreateNumericMatrix | mxGetData | mxGetImagData

**Introduced before R2006a**

# mxCopyReal8ToPtr (Fortran)

REAL*8 values from Fortran array to pointer array

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxCopyReal8ToPtr(y, px, n)
real*8 y(n)
mwPointer px
mwSize n
```

## Arguments

y

REAL*8 Fortran array

px

Pointer to the real or imaginary data of a double-precision MATLAB array

n

Number of elements to copy

## Description

mxCopyReal8ToPtr copies n REAL*8 values from the Fortran REAL*8 array y into the MATLAB array pointed to by px, either a pr or pi array.

Sets up standard Fortran arrays for passing as arguments to or from the computation routine of a MEX-file. Use this subroutine with Fortran compilers that do not support the %VAL construct.

## Examples

See these examples in *matlabroot*/extern/examples/eng_mat:

- matdemo1.F
- fengdemo.F

See these examples in *matlabroot*/extern/examples/refbook:

- timestwo.F
- xtimesy.F

## See Also

mxCopyPtrToReal8 | mxCreateNumericArray | mxCreateNumericMatrix | mxGetData | mxGetImagData

**Introduced before R2006a**

# mxCreateCellArray (C and Fortran)

N-D cell array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateCellArray(mwSize ndim, const mwSize *dims);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateCellArray(ndim, dims)
mwSize ndim
mwSize dims(ndim)
```

## Arguments

`ndim`

Number of dimensions in the created cell. For example, to create a three-dimensional cell `mxArray`, set `ndim` to 3.

`dims`

Dimensions array. Each element in the dimensions array contains the size of the `mxArray` in that dimension. For example, in C, setting `dims[0]` to `5` and `dims[1]` to `7` establishes a `5`-by-`7` `mxArray`. In Fortran, setting `dims(1)` to `5` and `dims(2)` to `7` establishes a `5`-by-`7` `mxArray`. Usually, the `dims` array contains `ndim` elements.

## Returns

Pointer to the created `mxArray`. If unsuccessful in a standalone (non-MEX file) application, returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX file, the MEX file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

## Description

Use `mxCreateCellArray` to create a cell `mxArray` with size defined by `ndim` and `dims`. For example, in C, to establish a three-dimensional cell `mxArray` having dimensions `4`-by-`8`-by-`7`, set:

```
ndim = 3;
dims[0] = 4; dims[1] = 8; dims[2] = 7;
```

In Fortran, to establish a three-dimensional cell `mxArray` having dimensions `4`-by-`8`-by-`7`, set:

```
ndim = 3;
dims(1) = 4; dims(2) = 8; dims(3) = 7;
```

The created cell `mxArray` is unpopulated; `mxCreateCellArray` initializes each cell to `NULL`. To put data into a cell, call `mxSetCell`.

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, then the resulting array has the dimensions 4-by-1-by-7.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `phonebook.c`

## See Also

`mxCreateCellMatrix`, `mxGetCell`, `mxSetCell`, `mxIsCell`

**Introduced before R2006a**

# mxCreateCellMatrix (C and Fortran)

2-D cell array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateCellMatrix(mwSize m, mwSize n);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateCellMatrix(m, n)
mwSize m, n
```

## Arguments

`m`

Number of rows

`n`

Number of columns

## Returns

Pointer to the created `mxArray`. If unsuccessful in a standalone (non-MEX file) application, returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX file, the MEX file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

## Description

Use `mxCreateCellMatrix` to create an `m`-by-`n` two-dimensional cell `mxArray`. The created cell `mxArray` is unpopulated; `mxCreateCellMatrix` initializes each cell to `NULL` in C (`0` in Fortran). To put data into cells, call `mxSetCell`.

`mxCreateCellMatrix` is identical to `mxCreateCellArray` except that `mxCreateCellMatrix` can create two-dimensional `mxArrays` only, but `mxCreateCellArray` can create `mxArrays` having any number of dimensions greater than 1.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreatecellmatrix.c`
- `mxcreatecellmatrixf.F`

## See Also

`mxCreateCellArray`

**Introduced before R2006a**

# mxCreateCharArray (C)

N-D mxChar array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateCharArray(mwSize ndim, const mwSize *dims);
```

## Description

Use mxCreateCharArray to create an N-dimensional mxChar array with each element set to NULL.

MATLAB automatically removes any trailing singleton dimensions specified in the dims argument. For example, if ndim equals 5 and dims equals [4 1 7 1 1], then the resulting array has the dimensions 4-by-1-by-7.

## Input Arguments

**ndim — Number of dimensions**
mwSize

Number of dimensions, specified as mwSize. If you specify 0, 1, or 2, then mxCreateCharArray creates a two-dimensional mxArray.

**dims — Dimensions array**
const mwSize *

Dimensions array, specified as const mwSize *.

Each element in the dimensions array contains the size of the array in that dimension. For example, to create a 5-by-7 array, set dims[0] to 5 and dims[1] to 7.

Usually, the dims array contains ndim elements.

## Output Arguments

**pm — Pointer to mxArray**
mxArray * | NULL

Pointer to an mxArray of type mxChar, specified as mxArray *.

The function is unsuccessful when there is not enough free heap space to create the mxArray.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL.

## See Also

mxCreateCharMatrixFromStrings | mxCreateString

**Introduced before R2006a**

# mxCreateCharArray (Fortran)

N-D mxChar array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateCharArray(ndim, dims)
mwSize ndim
mwSize dims(ndim)
```

## Description

Use mxCreateCharArray to create an N-dimensional mxChar array with each element set to 0.

MATLAB automatically removes any trailing singleton dimensions specified in the dims argument. For example, if ndim equals 5 and dims equals [4 1 7 1 1], then the resulting array has the dimensions 4-by-1-by-7.

## Input Arguments

**ndim — Number of dimensions**
mwSize

Number of dimensions, specified as mwSize. If you specify 0, 1, or 2, then mxCreateCharArray creates a two-dimensional mxArray.

**dims — Dimensions array**
array of mwSize

Dimensions array, specified as an array of mwSize.

Each element in the dimensions array contains the size of the array in that dimension. For example, to create a 5-by-7 array, set dims(1) to 5 and dims(2) to 7.

Usually, the dims array contains ndim elements.

## Output Arguments

**pm — Pointer to mxArray**
mwPointer | 0

Pointer to an mxArray of type mxChar, specified as mwPointer.

The function is unsuccessful when there is not enough free heap space to create the mxArray.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## See Also

mxCreateCharMatrixFromStrings | mxCreateString

**Introduced before R2006a**

# mxCreateCharMatrixFromStrings (C)

2-D mxChar array initialized to specified value

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateCharMatrixFromStrings(mwSize m, const char **str);
```

## Description

Use mxCreateCharMatrixFromStrings to create a two-dimensional mxArray, where each row is initialized to a string from str. The mxArray has dimensions m-by-max, where max is the length of the longest string in str.

The mxArray represents its data elements as mxChar rather than as C char.

## Input Arguments

**m — Number of strings**
mwSize

Number of strings, specified as mwSize.

**str — Array of strings**
const char **

Array of strings, specified as const char **. The array must contain at least m strings.

## Output Arguments

**pm — Pointer to mxArray**
mxArray * | NULL

Pointer to an mxArray of type mxChar, specified as mxArray *.

The function is unsuccessful when str contains fewer than m strings or there is not enough free heap space to create the mxArray.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL.

## Examples

See these examples in *matlabroot*/extern/examples/mx:

- mxcreatecharmatrixfromstr.c

## See Also

mxCreateCharArray | mxCreateString | mxGetString

**Introduced before R2006a**

# mxCreateCharMatrixFromStrings (Fortran)

2-D mxChar array initialized to specified value

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateCharMatrixFromStrings(m, str)
mwSize m
character*(*) str(m)
```

## Description

Use mxCreateCharMatrixFromStrings to create a two-dimensional mxArray, where each row is initialized to a string from str. The mxArray has dimensions m-by-n, where n is the number of characters in str(i).

## Input Arguments

**m — Number of strings**
mwSize

Number of strings, specified as mwSize.

**str — Array of strings**
character*(*)

Array of strings, specified as character*n array of size m, where each element of the array is n bytes.

## Output Arguments

**pm — Pointer to mxArray**
mwPointer | 0

Pointer to an mxArray of type mxChar, specified as mwPointer.

The function is unsuccessful when str contains fewer than m strings or there is not enough free heap space to create the mxArray.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## See Also
mxCreateCharArray | mxCreateString | mxGetString

**Introduced before R2006a**

# mxCreateDoubleMatrix (C)

2-D, double-precision, floating-point array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateDoubleMatrix(mwSize m, mwSize n, mxComplexity ComplexFlag);
```

## Description

Use `mxCreateDoubleMatrix` to create an `m`-by-`n` `mxArray`.

Call `mxDestroyArray` when you finish using the `mxArray`. The `mxDestroyArray` function deallocates the `mxArray` and its associated real and imaginary elements.

## Input Arguments

**`m` — Number of rows**
`mwSize`

Number of rows, specified as `mwSize`.

**`n` — Number of columns**
`mwSize`

Number of columns, specified as `mwSize`.

**`ComplexFlag` — Complex array indicator**
`mxComplexity`

Complex array indicator, specified as an `mxComplexity` value.

For applications built with the `mex -R2018a` command, the function initializes each data element to `0`.

For all other `mex` release-specific build options, the function sets each element in the `pr` array. If `ComplexFlag` is `mxCOMPLEX`, then the function sets the `pi` array to `0`.

## Output Arguments

**pm — Pointer to `mxArray`**
`mxArray *` | NULL

Pointer to an `mxArray` of type `mxDouble`, specified as `mxArray *`.

The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `NULL`.

## Examples

See these examples in *matlabroot*/extern/examples/refbook:

- convec.c
- findnz.c
- matrixDivide.c
- sincall.c
- timestwo.c
- xtimesy.c

## See Also

mxCreateNumericArray | mxDestroyArray

**Introduced before R2006a**

# mxCreateDoubleMatrix (Fortran)

2-D, double-precision, floating-point array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateDoubleMatrix(m, n, ComplexFlag)
mwSize m, n
integer*4 ComplexFlag
```

## Description

Use `mxCreateDoubleMatrix` to create an `m`-by-`n` `mxArray`.

Call `mxDestroyArray` when you finish using the `mxArray`. `mxDestroyArray` deallocates the `mxArray` and its associated real and imaginary elements.

## Input Arguments

**m — Number of rows**
`mwSize`

Number of rows, specified as `mwSize`.

**n — Number of columns**
`mwSize`

Number of columns, specified as `mwSize`.

**ComplexFlag — Complex array indicator**
`0` | `1`

Complex array indicator, specified as an `0` or `1`.

For applications built with the `mex -R2018a` command, the function initializes each data element to `0`.

For all other `mex` release-specific build options, the function sets each element in the `pr` array. If `ComplexFlag` is `1`, then the function sets the `pi` array to `0`.

## Output Arguments

**pm — Pointer to mxArray**
`mwPointer` | `0`

Pointer to an `mxArray` of type `mxDouble`, specified as `mwPointer`, if successful.

The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.

- Standalone (non-MEX file) application — Function returns `0`.

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `convec.F`
- `dblmat.F`
- `matsq.F`
- `timestwo.F`
- `xtimesy.F`

## See Also

`mxCreateNumericArray` | `mxCreateNumericMatrix` | `mxDestroyArray`

**Introduced before R2006a**

# mxCreateDoubleScalar (C)

Scalar, double-precision array initialized to specified value

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateDoubleScalar(double value);
```

## Description

Call `mxCreateDoubleScalar` to create a scalar `mxArray` of type `mxDouble`.

You can use `mxCreateDoubleScalar` instead of `mxCreateDoubleMatrix` in the following situation.

| Replace: | With: |
|---|---|
| `pa = mxCreateDoubleMatrix(1, 1, mxREAL);`<br>`*mxGetDoubles(pa) = value;` | `pa = mxCreateDoubleScalar(value);` |

Call `mxDestroyArray` when you finish using the `mxArray`.

## Input Arguments

**value — Scalar value**
`double`

Scalar value, specified as `double`.

## Output Arguments

**pm — Pointer to `mxArray`**
`mxArray *` | `NULL`

Pointer to an `mxArray` of type `mxDouble`, specified as `mxArray *`, if successful.

The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `NULL`.

## See Also

`mxCreateDoubleMatrix` | `mxDestroyArray`

**Introduced before R2006a**

# mxCreateDoubleScalar (Fortran)

Scalar, double-precision array initialized to specified value

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateDoubleScalar(value)
real*8 value
```

## Description

Call `mxCreateDoubleScalar` to create a scalar `mxArray` of type `mxDouble`.

## Description

Call `mxCreateDoubleScalar` to create a scalar double `mxArray`.

Call `mxDestroyArray` when you finish using the `mxArray`.

You can use `mxCreateDoubleScalar` instead of `mxCreateDoubleMatrix` in the following situation.

| Replace: | With: |
|---|---|
| `pm = mxCreateDoubleMatrix(1, 1, 0)`<br>`mxCopyReal8ToPtr(value, mxGetDoubles(pm), 1)` | `pm = mxCreateDoubleScalar(value)` |

## Input Arguments

**value — Scalar value**
`real*8`

Scalar value, specified as `real*8`.

## Output Arguments

**pm — Pointer to `mxArray`**
`mwPointer` | `0`

Pointer to an `mxArray`, specified as `mwPointer`, if successful.

The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`.

## See Also

`mxCreateDoubleMatrix` | `mxDestroyArray`

**Introduced before R2006a**

# mxCreateLogicalArray (C)

N-D logical array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateLogicalArray(mwSize ndim, const mwSize *dims);
```

## Arguments

`ndim`

Number of dimensions. If you specify a value for `ndim` that is less than 2, `mxCreateLogicalArray` automatically sets the number of dimensions to 2.

`dims`

Dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, setting `dims[0]` to `5` and `dims[1]` to `7` establishes a `5`-by-`7` `mxArray`. There are `ndim` elements in the `dims` array.

## Returns

Pointer to the created `mxArray`. If unsuccessful in a standalone (non-MEX file) application, returns `NULL`. If unsuccessful in a MEX file, the MEX file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

## Description

Call `mxCreateLogicalArray` to create an N-dimensional `mxArray` of `mxLogical` elements. After creating the `mxArray`, `mxCreateLogicalArray` initializes all its elements to logical `0`. `mxCreateLogicalArray` differs from `mxCreateLogicalMatrix` in that the latter can create two-dimensional arrays only.

`mxCreateLogicalArray` allocates dynamic memory to store the created `mxArray`. When you finish with the created `mxArray`, call `mxDestroyArray` to deallocate its memory.

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, then the resulting array has the dimensions 4-by-1-by-7.

## See Also

`mxCreateLogicalMatrix` | `mxCreateLogicalScalar` | `mxCreateSparseLogicalMatrix`

**Introduced before R2006a**

# mxCreateLogicalMatrix (C)

2-D logical array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateLogicalMatrix(mwSize m, mwSize n);
```

## Arguments

`m`

Number of rows

`n`

Number of columns

## Returns

Pointer to the created `mxArray`. If unsuccessful in a standalone (non-MEX file) application, returns `NULL`. If unsuccessful in a MEX file, the MEX file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

## Description

Use `mxCreateLogicalMatrix` to create an `m`-by-`n` `mxArray` of `mxLogical` elements. `mxCreateLogicalMatrix` initializes each element in the array to logical `0`.

Call `mxDestroyArray` when you finish using the `mxArray`. `mxDestroyArray` deallocates the `mxArray`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxislogical.c`

## See Also

`mxCreateLogicalArray` | `mxCreateLogicalScalar` | `mxCreateSparseLogicalMatrix`

**Introduced before R2006a**

# mxCreateLogicalScalar (C)

Scalar, logical array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateLogicalScalar(mxLogical value);
```

## Arguments

`value`

Logical value to which you want to initialize the array

## Returns

Pointer to the created `mxArray`. If unsuccessful in a standalone (non-MEX file) application, returns `NULL`. If unsuccessful in a MEX file, the MEX file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

## Description

Call `mxCreateLogicalScalar` to create a scalar logical `mxArray`. `mxCreateLogicalScalar` is a convenience function that replaces the following code:

```
pa = mxCreateLogicalMatrix(1, 1);
*mxGetLogicals(pa) = value;
```

When you finish using the `mxArray`, call `mxDestroyArray` to destroy it.

## See Also

`mxCreateLogicalArray` | `mxCreateLogicalMatrix` | `mxDestroyArray` | `mxGetLogicals` | `mxIsLogicalScalar` | `mxIsLogicalScalarTrue`

**Introduced before R2006a**

# mxCreateNumericArray (C)

N-D numeric array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateNumericArray(mwSize ndim, const mwSize *dims,
    mxClassID classid, mxComplexity ComplexFlag);
```

## Description

Use `mxCreateNumericArray` to create an N-dimensional `mxArray`. The data elements have the numeric data type specified by `classid`.

`mxCreateNumericArray` differs from `mxCreateDoubleMatrix` as follows:

- All data elements in `mxCreateDoubleMatrix` are double-precision, floating-point numbers. The data elements in `mxCreateNumericArray` can be any numerical type, including different integer precisions.
- `mxCreateDoubleMatrix` creates two-dimensional arrays only. `mxCreateNumericArray` can create arrays of two or more dimensions.

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, then the resulting array has the dimensions 4-by-1-by-7.

This table shows the C `classid` values that are equivalent to MATLAB classes.

| MATLAB Class Name | C classid Value |
|---|---|
| int8 | mxINT8_CLASS |
| uint8 | mxUINT8_CLASS |
| int16 | mxINT16_CLASS |
| uint16 | mxUINT16_CLASS |
| int32 | mxINT32_CLASS |
| uint32 | mxUINT32_CLASS |
| int64 | mxINT64_CLASS |
| uint64 | mxUINT64_CLASS |
| single | mxSINGLE_CLASS |
| double | mxDOUBLE_CLASS |

Call `mxDestroyArray` when you finish using the `mxArray`. The `mxDestroyArray` function deallocates the `mxArray` and its associated real and imaginary elements.

## Input Arguments

**`ndim` — Number of dimensions**
`mwSize`

Number of dimensions, specified as `mwSize`. If `ndim` is less than 2, then `mxCreateNumericArray` sets the number of dimensions to 2.

**`dims` — Dimensions array**
`const mwSize *`

Dimensions array, specified as `const mwSize *`.

Each element in the dimensions array contains the size of the array in that dimension. For example, to create a 5-by-7 array, set `dims[0]` to 5 and `dims[1]` to 7.

Usually, the `dims` array contains `ndim` elements.

**`classid` — Class identifier**
`mxClassID`

Class identifier, specified as an `mxClassID` enumeration. `classid` determines how the numerical data is represented in memory. For example, `mxCreateNumericMatrix` stores `mxINT16_CLASS` values as 16-bit signed integers.

**`ComplexFlag` — Complex array indicator**
`mxComplexity`

Complex array indicator, specified as an `mxComplexity` value.

For applications built with the `mex -R2018a` command, the function initializes each data element to `0`.

For all other `mex` release-specific build options, the function sets each element in the `pr` array. If `ComplexFlag` is `mxCOMPLEX`, then the function sets the `pi` array to `0`.

## Output Arguments

**`pm` — Pointer to `mxArray`**
`mxArray * | NULL`

Pointer to an `mxArray` of type `classid`, specified as `mxArray *`.

The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `NULL`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `phonebook.c`

- `doubleelement.c`
- `matrixDivide.c`
- `matsqint8.F`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxisfinite.c`

## See Also

`mxClassID` | `mxComplexity` | `mxCreateNumericMatrix` | `mxCreateUninitNumericArray` | `mxDestroyArray`

**Introduced before R2006a**

# mxCreateNumericArray (Fortran)

N-D numeric array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateNumericArray(ndim, dims, classid, ComplexFlag)
mwSize ndim
mwSize dims(ndim)
integer*4 classid, ComplexFlag
```

## Description

Use `mxCreateNumericArray` to create an N-dimensional `mxArray`.

`mxCreateNumericArray` differs from `mxCreateDoubleMatrix` as follows:

- All data elements in `mxCreateDoubleMatrix` are double-precision, floating-point numbers. The data elements in `mxCreateNumericArray` can be any numerical type, including different integer precisions.
- `mxCreateDoubleMatrix` creates two-dimensional arrays only. `mxCreateNumericArray` can create arrays of two or more dimensions.

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, then the resulting array has the dimensions 4-by-1-by-7.

This table shows the Fortran types that are equivalent to MATLAB classes.

| MATLAB Class Name | Fortran Type |
|---|---|
| `int8` | `BYTE` |
| `int16` | `INTEGER*2` |
| `int32` | `INTEGER*4` |
| `int64` | `INTEGER*8` |
| `single` | `REAL*4`<br>`COMPLEX*8` |
| `double` | `REAL*8`<br>`COMPLEX*16` |

Call `mxDestroyArray` when you finish using the `mxArray`. The `mxDestroyArray` function deallocates the `mxArray` and its associated real and imaginary elements.

## Input Arguments

**`ndim` — Number of dimensions**
`mwSize`

Number of dimensions, specified as mwSize. If ndim is less than 2, then mxCreateNumericArray sets the number of dimensions to 2.

**dims — Dimensions array**
array of mwSize

Dimensions array, specified as an array of mwSize.

Each element in the dimensions array contains the size of the array in that dimension. For example, to create a 5-by-7 array, set dims(1) to 5 and dims(2) to 7.

Usually, the dims array contains ndim elements.

**classid — Class identifier**
integer*4

Class identifier, specified as integer*4. classid determines how the numerical data is represented in memory. Use the mxClassIdFromClassName function to derive the classid value from a MATLAB class name.

**ComplexFlag — Complex array indicator**
0 | 1

Complex array indicator, specified as an 0 or 1.

For applications built with the mex -R2018a command, the function initializes each data element to 0.

For all other mex release-specific build options, the function sets each element in the pr array. If ComplexFlag is 1, then the function sets the pi array to 0.

## Output Arguments

**pm — Pointer to mxArray**
mwPointer | 0

Pointer to an mxArray of type classid, specified as mwPointer.

The function is unsuccessful when there is not enough free heap space to create the mxArray.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## See Also
mxClassIdFromClassName | mxCreateNumericMatrix | mxDestroyArray

**Introduced before R2006a**

# mxCreateNumericMatrix (C)

2-D numeric matrix

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateNumericMatrix(mwSize m, mwSize n, mxClassID classid, mxComplexity ComplexFlag);
```

## Description

Use `mxCreateNumericMatrix` to create a 2-D `mxArray`. The `classid` specifies the numeric data type of the elements in the array.

This table shows the C `classid` values that are equivalent to MATLAB classes.

| MATLAB Class Name | C classid Value |
|---|---|
| int8 | mxINT8_CLASS |
| uint8 | mxUINT8_CLASS |
| int16 | mxINT16_CLASS |
| uint16 | mxUINT16_CLASS |
| int32 | mxINT32_CLASS |
| uint32 | mxUINT32_CLASS |
| int64 | mxINT64_CLASS |
| uint64 | mxUINT64_CLASS |
| single | mxSINGLE_CLASS |
| double | mxDOUBLE_CLASS |

Call `mxDestroyArray` when you finish using the `mxArray`. The `mxDestroyArray` function deallocates the `mxArray` and its associated real and imaginary elements.

## Input Arguments

**m — Number of rows**
`mwSize`

Number of rows, specified as `mwSize`.

**n — Number of columns**
`mwSize`

Number of columns, specified as `mwSize`.

**classid — Class identifier**
`mxClassID`

Class identifier, specified as an mxClassID enumeration. The classid argument determines how the numerical data is represented in memory. For example, mxCreateNumericMatrix stores mxINT16_CLASS values as 16-bit signed integers.

**ComplexFlag — Complex array indicator**
mxComplexity

Complex array indicator, specified as an mxComplexity value.

For applications built with the mex -R2018a command, the function initializes each data element to 0.

For all other mex release-specific build options, the function sets each element in the pr array. If ComplexFlag is mxCOMPLEX, then the function sets the pi array to 0.

## Output Arguments

**pm — Pointer to mxArray**
mxArray * | NULL

Pointer to an mxArray of type classid, specified as mxArray *, if successful.

The function is unsuccessful when there is not enough free heap space to create the mxArray.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL.

## Examples

See these examples in *matlabroot*/extern/examples/refbook:

- arrayFillGetPr.c

## See Also

mxClassID | mxComplexity | mxCreateNumericArray | mxCreateUninitNumericMatrix | mxDestroyArray

**Introduced before R2006a**

# mxCreateNumericMatrix (Fortran)

2-D numeric matrix

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateNumericMatrix(m, n, classid, ComplexFlag)
mwSize m, n
integer*4 classid, ComplexFlag
```

## Description

Use `mxCreateNumericMatrix` to create a 2-D `mxArray`. The `classid` specifies the numeric data type of the elements in the array.

This table shows the Fortran types that are equivalent to MATLAB classes.

| MATLAB Class Name | Fortran Type |
|---|---|
| int8 | BYTE |
| int16 | INTEGER*2 |
| int32 | INTEGER*4 |
| int64 | INTEGER*8 |
| single | REAL*4<br>COMPLEX*8 |
| double | REAL*8<br>COMPLEX*16 |

Call `mxDestroyArray` when you finish using the `mxArray`. The `mxDestroyArray` function deallocates the `mxArray` and its associated real and imaginary elements.

## Input Arguments

**m — Number of rows**
mwSize

Number of rows, specified as `mwSize`.

**n — Number of columns**
mwSize

Number of columns, specified as `mwSize`.

**classid — Class identifier**
integer*4

Class identifier, specified as `integer*4`. The `classid` argument determines how the numerical data is represented in memory. Use the `mxClassIdFromClassName` function to derive the `classid` value from a MATLAB class name.

**ComplexFlag — Complex array indicator**
0 | 1

Complex array indicator, specified as an 0 or 1.

For applications built with the mex -R2018a command, the function initializes each data element to 0.

For all other mex release-specific build options, the function sets each element in the pr array. If ComplexFlag is 1, then the function sets the pi array to 0.

## Output Arguments

**pm — Pointer to mxArray**
mwPointer | 0

Pointer to an mxArray of type mxChar, specified as mwPointer, if successful.

The function is unsuccessful when there is not enough free heap space to create the mxArray.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

These Fortran statements create a 4-by-3 matrix of REAL*4 elements having no imaginary components:

```
C      Create 4x3 mxArray of REAL*4
       mxCreateNumericMatrix(4, 3,
     +                mxClassIDFromClassName('single'), 0)
```

## See Also

mxClassIdFromClassName | mxCreateNumericArray | mxDestroyArray

**Introduced before R2006a**

# mxCreateSparse (C and Fortran)

2-D sparse array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateSparse(mwSize m, mwSize n, mwSize nzmax,
         mxComplexity ComplexFlag);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateSparse(m, n, nzmax, ComplexFlag)
mwSize m, n, nzmax
integer*4 ComplexFlag
```

## Arguments

`m`

Number of rows

`n`

Number of columns

`nzmax`

Number of elements that `mxCreateSparse` should allocate to hold the `pr`, `ir`, and, if `ComplexFlag` is `mxCOMPLEX` in C (`1` in Fortran), `pi` arrays. Set the value of `nzmax` to be greater than or equal to the number of nonzero elements you plan to put into the `mxArray`, but make sure that `nzmax` is less than or equal to `m*n`. `nzmax` is greater than or equal to 1.

`ComplexFlag`

If the `mxArray` you are creating is to contain imaginary data, set `ComplexFlag` to `mxCOMPLEX` in C (`1` in Fortran). Otherwise, set `ComplexFlag` to `mxREAL` in C (`0` in Fortran).

## Returns

Pointer to the created `mxArray`. If unsuccessful in a standalone (non-MEX file) application, returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX file, the MEX file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`. In that case, try reducing `nzmax`, `m`, or `n`.

## Description

Call `mxCreateSparse` to create an unpopulated sparse double `mxArray`. The returned sparse `mxArray` contains no sparse information and cannot be passed as an argument to any MATLAB sparse functions. To make the returned sparse `mxArray` useful, initialize the `pr`, `ir`, `jc`, and (if it exists) `pi` arrays.

`mxCreateSparse` allocates space for:

- A `pr` array of length `nzmax`.
- A `pi` array of length `nzmax`, but only if `ComplexFlag` is `mxCOMPLEX` in C (`1` in Fortran).
- An `ir` array of length `nzmax`.
- A `jc` array of length `n+1`.

When you finish using the sparse `mxArray`, call `mxDestroyArray` to reclaim all its heap space.

## Examples

See these examples in *matlabroot*/extern/examples/refbook:

- `fulltosparse.c`
- `fulltosparse.F`

## See Also

`mxComplexity` | `mxDestroyArray` | `mxSetIr` | `mxSetJc` | `mxSetNzmax`

**Introduced before R2006a**

# mxCreateSparseLogicalMatrix (C)

2-D, sparse, logical array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateSparseLogicalMatrix(mwSize m, mwSize n,
  mwSize nzmax);
```

## Arguments

`m`

Number of rows

`n`

Number of columns

`nzmax`

Number of elements that `mxCreateSparseLogicalMatrix` should allocate to hold the data. Set the value of `nzmax` to be greater than or equal to the number of nonzero elements you plan to put into the `mxArray`, but make sure that `nzmax` is less than or equal to `m*n`. `nzmax` is greater than or equal to 1.

## Returns

Pointer to the created `mxArray`. If unsuccessful in a standalone (non-MEX file) application, returns `NULL`. If unsuccessful in a MEX file, the MEX file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

## Description

Use `mxCreateSparseLogicalMatrix` to create an `m`-by-`n` `mxArray` of `mxLogical` elements. `mxCreateSparseLogicalMatrix` initializes each element in the array to logical `0`.

Call `mxDestroyArray` when you finish using the `mxArray`. `mxDestroyArray` deallocates the `mxArray` and its elements.

## See Also

`mxCreateLogicalArray` | `mxCreateLogicalMatrix` | `mxCreateLogicalScalar` | `mxCreateSparse` | `mxIsLogical`

**Introduced before R2006a**

# mxCreateString (C)

1-N array initialized to specified string

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateString(const char *str);
```

## Description

Use `mxCreateString` to create an `mxArray` initialized from `str`.

Call `mxDestroyArray` when you finish using the `mxArray`.

## Input Arguments

**`str` — String**
`const char *`

String, specified as `const char *`. This string can be encoded using UTF-8 or, for backwards compatibility, the local code page (LCP) encoding.

## Output Arguments

**pm — Pointer to `mxArray`**
`mxArray *` | `NULL`

Pointer to an `mxArray`, specified as `mxArray *`, if successful.

The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `NULL`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `revord.c`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreatestructarray.c`
- `mxisclass.c`

## See Also

`mxCreateCharArray` | `mxCreateCharMatrixFromStrings`

**Introduced before R2006a**

# mxCreateString (Fortran)

1-N array initialized to specified string

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateString(str)
character*(*) str
```

## Description

Use `mxCreateString` to create an `mxArray` initialized to `str`. Many MATLAB functions, such as `strcmp` and `upper`, require string array inputs.

`mxCreateString` supports both multibyte and single-byte encoded characters. On Windows and Linux platforms, the user locale setting specifies the default encoding.

Call `mxDestroyArray` when you finish using the `mxArray`.

## Input Arguments

**`str` — String**
`character*(*)`

String, specified as `character*(*)`. Only ASCII characters are supported.

## Output Arguments

**pm — Pointer to `mxArray`**
`mwPointer` | `0`

Pointer to an `mxArray` of type `mxChar`, specified as `mwPointer`, if successful.

The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `revord.F`

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matdemo1.F`

## See Also

`mxCreateCharArray` | `mxCreateCharMatrixFromStrings`

**Introduced before R2006a**

# mxCreateStructArray (C)

N-D structure array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateStructArray(
    mwSize ndim, const mwSize *dims, int nfields, const char **fieldnames);
```

## Description

Call `mxCreateStructArray` to create an unpopulated structure `mxArray`. Each element of a structure `mxArray` contains the same number of fields (specified in `nfields`). Each field has a name, specified in `fieldnames`. A MATLAB structure `mxArray` is conceptually identical to an array of `structs` in the C language.

Each field holds one `mxArray` pointer initialized to `NULL`. Call `mxSetField` or `mxSetFieldByNumber` to place a non-`NULL` `mxArray` pointer in a field.

The function automatically removes trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, then the dimensions of the resulting array are `4`-by-`1`-by-`7`.

Call `mxDestroyArray` when you finish using the `mxArray` to deallocate the `mxArray` and its associated real and imaginary elements.

## Input Arguments

**`ndim` — Number of dimensions**
`mwSize`

Number of dimensions, specified as `mwSize`. If `ndim` is less than 2, then `mxCreateStructArray` sets the number of dimensions to 2.

**`dims` — Dimensions array**
array of `const mwSize`

Dimensions array, specified as an array of `const mwSize`.

Each element in the dimensions array contains the size of the array in that dimension. For example, to create a `5`-by-`7` array, set `dims[0]` to 5 and `dims[1]` to 7.

Usually, the `dims` array contains `ndim` elements.

**`nfields` — Number of fields**
`int`

Number of fields in each element, specified as `int`.

**`fieldnames` — Field names**
`const char **`

One or more field names, specified as `const char **`.

Field names must be valid MATLAB identifiers, which means they cannot be `NULL` or empty. Field names are case-sensitive. To determine the maximum length of a field name, use the `namelengthmax` function.

## Output Arguments

**pm — Pointer to `mxArray`**
`mxArray *` | `NULL`

Pointer to an `mxArray`, specified as `mxArray *`.

The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `NULL`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreatestructarray.c`

## See Also

`mxAddField` | `mxDestroyArray` | `mxRemoveField` | `mxSetField` | `mxSetFieldByNumber` | `namelengthmax`

**Introduced before R2006a**

# mxCreateStructArray (Fortran)

N-D structure array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateStructArray(ndim, dims, nfields, fieldnames)
mwSize ndim
mwSize dims(ndim)
integer*4 nfields
character*(*) fieldnames(nfields)
```

## Description

Call `mxCreateStructArray` to create an unpopulated structure `mxArray`. Each element of a structure `mxArray` contains the same number of fields, specified in `nfields`. Each field has a name, specified in `fieldnames`.

Each field holds one `mxArray` pointer initialized to `0`. Call `mxSetField` or `mxSetFieldByNumber` to place a non-`0` `mxArray` pointer in a field.

The function automatically removes trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, then the dimensions of the resulting array are 4-by-1-by-7.

Call `mxDestroyArray` when you finish using the `mxArray`. The `mxDestroyArray` function deallocates the `mxArray` and its associated real and imaginary elements.

## Input Arguments

**`ndim` — Number of dimensions**
`mwSize`

Number of dimensions, specified as `mwSize`. If `ndim` is less than 2, then `mxCreateStructArray` sets the number of dimensions to 2.

**`dims` — Dimensions array**
array of `mwSize`

Dimensions array, specified as an `ndim` array of `mwSize`.

Each element in the dimensions array contains the size of the array in that dimension. For example, to create a `5`-by-`7` array, set `dims(1)` to 5 and `dims(2)` to 7.

Usually, the `dims` array contains `ndim` elements.

**`nfields` — Number of fields**
`integer*4`

Number of fields in each element, specified as `integer*4`.

**fieldnames — Field names**
character*(*)

One or more field names, specified as character*(*).

Field names must be valid MATLAB identifiers, which means they cannot be empty. Field names are case-sensitive. To determine the maximum length of a field name, use the namelengthmax function.

## Output Arguments

**pm — Pointer to mxArray**
mwPointer | 0

Pointer to an mxArray, specified as mwPointer.

The function is unsuccessful when there is not enough free heap space to create the mxArray.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## See Also

mxAddField | mxCreateStructMatrix | mxDestroyArray | mxRemoveField | mxSetField | mxSetFieldByNumber | namelengthmax

**Introduced before R2006a**

# mxCreateStructMatrix (C)

2-D structure array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateStructMatrix(mwSize m, mwSize n, int nfields, const char **fieldnames);
```

## Description

Call `mxCreateStructMatrix` to create an unpopulated, two-dimensional, structure `mxArray`. For information about the structure, see `mxCreateStructArray`.

Call `mxDestroyArray` when you finish using the `mxArray` to deallocate the `mxArray` and its associated elements.

## Input Arguments

### m — Number of rows

`mwSize`

Number of rows, specified as `mwSize`.

### n — Number of columns

`mwSize`

Number of columns, specified as `mwSize`.

### nfields — Number of fields

`int`

Number of fields in each element, specified as `int`.

### fieldnames — Field names

`const char **`

One or more field names, specified as `const char **`.

Field names must be valid MATLAB identifiers, which means they cannot be `NULL` or empty. Field names are case-sensitive. To determine the maximum length of a field name, use the `namelengthmax` function.

## Output Arguments

### pm — Pointer to mxArray

`mxArray * | NULL`

Pointer to an `mxArray`, specified as `mxArray *`, if successful.

The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `phonebook.c`

## See Also

`mxCreateStructArray` | `namelengthmax`

**Introduced before R2006a**

# mxCreateStructMatrix (Fortran)

2-D structure array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxCreateStructMatrix(m, n, nfields, fieldnames)
mwSize m, n
integer*4 nfields
character*(*) fieldnames(nfields)
```

## Description

Call `mxCreateStructMatrix` to create an unpopulated, two-dimensional, structure `mxArray`. For information about the structure, see `mxCreateStructArray`.

Call `mxDestroyArray` when you finish using the `mxArray`. `mxDestroyArray` deallocates the `mxArray` and its associated elements.

## Input Arguments

**`m` — Number of rows**
`mwSize`

Number of rows, specified as `mwSize`.

**`n` — Number of columns**
`mwSize`

Number of columns, specified as `mwSize`.

**`nfields` — Number of fields**
`integer*4`

Number of fields in each element, specified as `integer*4`.

**`fieldnames` — Field names**
`character*(*)`

One or more field names, specified as `character*(*)`.

Field names must be valid MATLAB identifiers, which means they cannot be empty. Field names are case-sensitive. To determine the maximum length of a field name, use the `namelengthmax` function.

**`pm` — Pointer to `mxArray`**
`mwPointer | 0`

Pointer to an `mxArray`, specified as `mwPointer`, if successful.

The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`.

## See Also

`mxCreateStructArray` | `mxDestroyArray` | `namelengthmax`

**Introduced before R2006a**

# mxCreateUninitNumericArray (C)

Uninitialized N-D numeric array

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateUninitNumericArray(size_t ndim, size_t *dims,
  mxClassID classid, mxComplexity ComplexFlag);
```

## Arguments

`ndim`

Number of dimensions. If you specify a value for `ndim` that is less than 2, `mxCreateUninitNumericArray` automatically sets the number of dimensions to 2.

`dims`

Dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, setting `dims[0]` to 5 and `dims[1]` to 7 establishes a `5`-by-`7` `mxArray`. Usually, the `dims` array contains `ndim` elements.

`classid`

Identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying `mxINT16_CLASS` causes each piece of numerical data in the `mxArray` to be represented as a 16-bit signed integer.

`ComplexFlag`

If the `mxArray` you are creating is to contain imaginary data, set `ComplexFlag` to `mxCOMPLEX`. Otherwise, set `ComplexFlag` to `mxREAL`.

## Returns

Pointer to the created `mxArray`. If unsuccessful in a standalone (non-MEX-file) application, returns `NULL`. If unsuccessful in a MEX-file, the MEX-file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

## Description

Call `mxCreateUninitNumericArray` to create an N-dimensional `mxArray` in which all data elements have the numeric data type specified by `classid`. Data elements are not initialized.

`mxCreateUninitNumericArray` allocates dynamic memory to store the created `mxArray`. Call `mxDestroyArray` to deallocate the memory.

The following table shows the C `classid` values that are equivalent to MATLAB classes.

| MATLAB Class Name | C classid Value |
| --- | --- |
| int8 | mxINT8_CLASS |

| MATLAB Class Name | C classid Value |
|---|---|
| uint8 | mxUINT8_CLASS |
| int16 | mxINT16_CLASS |
| uint16 | mxUINT16_CLASS |
| int32 | mxINT32_CLASS |
| uint32 | mxUINT32_CLASS |
| int64 | mxINT64_CLASS |
| uint64 | mxUINT64_CLASS |
| single | mxSINGLE_CLASS |
| double | mxDOUBLE_CLASS |

## See Also

mxDestroyArray, mxCreateUninitNumericMatrix, mxCreateNumericArray

**Introduced in R2015a**

# mxCreateUninitNumericMatrix (C)

Uninitialized 2-D numeric matrix

## C Syntax

```
#include "matrix.h"
mxArray *mxCreateUninitNumericMatrix(size_t m, size_t n,
  mxClassID classid, mxComplexity ComplexFlag);
```

## Arguments

`m`

Number of rows

`n`

Number of columns

`classid`

Identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying `mxINT16_CLASS` causes each piece of numerical data in the `mxArray` to be represented as a 16-bit signed integer.

`ComplexFlag`

If the `mxArray` you are creating is to contain imaginary data, set `ComplexFlag` to `mxCOMPLEX`. Otherwise, set `ComplexFlag` to `mxREAL`.

## Returns

Pointer to the created `mxArray`, if successful. If unsuccessful in a standalone (non-MEX-file) application, returns `NULL`. If unsuccessful in a MEX-file, the MEX-file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

## Example

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreateuninitnumericmatrix.c`

## Description

Call `mxCreateUninitNumericMatrix` to create a 2-D `mxArray` in which all data elements have the numeric data type specified by `classid`. Data elements are not initialized.

`mxCreateUninitNumericMatrix` allocates dynamic memory to store the created `mxArray`. Call `mxDestroyArray` to deallocate the memory.

The following table shows the C `classid` values that are equivalent to MATLAB classes.

| MATLAB Class Name | C classid Value |
|---|---|
| int8 | mxINT8_CLASS |
| uint8 | mxUINT8_CLASS |
| int16 | mxINT16_CLASS |
| uint16 | mxUINT16_CLASS |
| int32 | mxINT32_CLASS |
| uint32 | mxUINT32_CLASS |
| int64 | mxINT64_CLASS |
| uint64 | mxUINT64_CLASS |
| single | mxSINGLE_CLASS |
| double | mxDOUBLE_CLASS |

## See Also

mxDestroyArray, mxCreateUninitNumericArray, mxCreateNumericMatrix

**Introduced in R2015a**

# mxDestroyArray (C)

Free dynamic memory allocated by MXCREATE* functions

## C Syntax

```
#include "matrix.h"
void mxDestroyArray(mxArray *pm);
```

## Description

`mxDestroyArray` deallocates memory for the specified `mxArray` including:

- Characteristics fields of the `mxArray`, such as size (`m` and `n`) and type
- Associated data arrays, such as `ir` and `jc` for sparse arrays
- Fields of structure arrays
- Cells of cell arrays

Do not call `mxDestroyArray` on an `mxArray`:

- Returned in a left-side argument of a MEX file
- Returned by the `mxGetField` or `mxGetFieldByNumber` functions
- Returned by the `mxGetCell` function

## Input Arguments

**pm — Pointer to `mxArray`**
`mxArray *`

Pointer to the `mxArray` to free, specified as `mxArray *`. If `pm` is a `NULL` pointer, then the function does nothing.

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `matrixDivide.c`
- `matrixDivideComplex.c`
- `sincall.c`

See these examples in *matlabroot*`/extern/examples/mex`:

- `mexcallmatlab.c`
- `mexgetarray.c`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxisclass.c`

## See Also

`mexMakeArrayPersistent` | `mexMakeMemoryPersistent` | `mxCalloc` | `mxFree` | `mxMalloc`

**Introduced before R2006a**

# mxDestroyArray (Fortran)

Free dynamic memory allocated by MXCREATE* functions

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxDestroyArray(pm)
mwPointer pm
```

## Description

`mxDestroyArray` deallocates memory for the specified `mxArray` including:

- Characteristics fields of the `mxArray`, such as size (`m` and `n`) and type
- Associated data arrays, such as `ir` and `jc` for sparse arrays
- Fields of structure arrays
- Cells of cell arrays

Do not call `mxDestroyArray` on an `mxArray`:

- Returned in a left-side argument of a MEX file
- Returned by the `mxGetField` or `mxGetFieldByNumber` functions
- Returned by the `mxGetCell` function

## Input Arguments

**pm — Pointer to `mxArray`**
`mwPointer`

Pointer to the `mxArray` to free, specified as `mwPointer`. If `pm` is `0`, then the function does nothing.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `sincall.F`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreatecellmatrixf.F`

## See Also

`mexMakeArrayPersistent` | `mexMakeMemoryPersistent` | `mxCalloc` | `mxFree` | `mxMalloc`

**Introduced before R2006a**

# mxDuplicateArray (C and Fortran)

Make deep copy of array

## C Syntax

```
#include "matrix.h"
mxArray *mxDuplicateArray(const mxArray *in);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxDuplicateArray(in)
mwPointer in
```

## Arguments

`in`

Pointer to the `mxArray` you want to copy

## Returns

Pointer to the created `mxArray`. If unsuccessful in a standalone (non-MEX file) application, returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX file, the MEX file terminates and returns control to the MATLAB prompt. The function is unsuccessful when there is not enough free heap space to create the `mxArray`.

## Description

`mxDuplicateArray` makes a deep copy of an array, and returns a pointer to the copy. A deep copy refers to a copy in which all levels of data are copied. For example, a deep copy of a cell array copies each cell and the contents of each cell (if any).

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `phonebook.c`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxcreatecellmatrix.c`
- `mxcreatecellmatrixf.F`
- `mxgetinf.c`
- `mxsetdimensions.c`
- `mxsetdimensionsf.F`
- `mxsetnzmax.c`

**Introduced before R2006a**

# mxFree (C and Fortran)

Free dynamic memory allocated by mxCalloc, mxMalloc, mxRealloc, mxArrayToString, or mxArrayToUTF8String functions

## C Syntax

```
#include "matrix.h"
void mxFree(void *ptr);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxFree(ptr)
mwPointer ptr
```

## Arguments

`ptr`

Pointer to the beginning of any memory parcel allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`. If `ptr` is a `NULL` pointer, the function does nothing.

## Description

`mxFree` deallocates heap space using the MATLAB memory management facility. This function ensures correct memory management in error and abort (**Ctrl+C**) conditions.

To deallocate heap space in C MATLAB applications, call `mxFree` instead of the ANSI C `free` function.

In MEX files, but excluding MAT or engine standalone applications, the MATLAB memory management facility maintains a list of all memory allocated by the following functions:

- `mxCalloc`
- `mxMalloc`
- `mxRealloc`
- `mxArrayToString`
- `mxArrayToUTF8String`

The memory management facility automatically deallocates all parcels managed by a MEX file when the MEX file completes and control returns to the MATLAB prompt. `mxFree` also removes the memory parcel from the memory management list of parcels.

When `mxFree` appears in a MAT or engine standalone MATLAB application, it simply deallocates the contiguous heap space that begins at address `ptr`.

In MEX files, your use of `mxFree` depends on whether the specified memory parcel is persistent or nonpersistent. By default, memory parcels created by `mxCalloc`, `mxMalloc`, `mxRealloc`, `mxArrayToString`, and `mxArrayToUTF8String` are nonpersistent. The memory management facility automatically frees all nonpersistent memory whenever a MEX file completes. Thus, even if you do not call `mxFree`, MATLAB takes care of freeing the memory for you. Nevertheless, it is good programming practice to deallocate memory when you are through using it. Doing so generally makes the entire system run more efficiently.

If an application calls `mexMakeMemoryPersistent`, the specified memory parcel becomes persistent. When a MEX file completes, the memory management facility does not free persistent memory parcels. Therefore, the only way to free a persistent memory parcel is to call `mxFree`. Typically, MEX files call `mexAtExit` to register a cleanup handler. The cleanup handler calls `mxFree`.

Do not use `mxFree` for an `mxArray` created by any other functions in the Matrix Library API. Use `mxDestroyArray` instead.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcalcsinglesubscript.c`
- `mxcreatecharmatrixfromstr.c`
- `mxisfinite.c`
- `mxmalloc.c`
- `mxsetdimensions.c`

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `phonebook.c`

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxCalloc`, `mxDestroyArray`, `mxMalloc`, `mxRealloc`, `mxArrayToString`, `mxArrayToUTF8String`

**Introduced before R2006a**

# mxGetCell (C and Fortran)

Pointer to element in cell array

## C Syntax

```
#include "matrix.h"
mxArray *mxGetCell(const mxArray *pm, mwIndex index);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetCell(pm, index)
mwPointer pm
mwIndex index
```

## Arguments

`pm`

Pointer to a cell `mxArray`

`index`

Number of elements in the cell `mxArray` between the first element and the desired one. See `mxCalcSingleSubscript` for details on calculating an index in a multidimensional cell array.

## Returns

Pointer to the `i`th cell `mxArray` if successful. Otherwise, returns `NULL` in C (`0` in Fortran). Causes of failure include:

- Specifying the index of a cell array element that has not been populated.
- Specifying a `pm` that does not point to a cell `mxArray`.
- Specifying an `index` to an element outside the bounds of the `mxArray`.
- Insufficient heap space.

Do not call `mxDestroyArray` on an `mxArray` returned by the `mxGetCell` function.

## Description

Call `mxGetCell` to get a pointer to the `mxArray` held in the indexed element of the cell `mxArray`.

**Note** Inputs to a MEX-file are constant read-only `mxArrays`. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## See Also

`mxCreateCellArray`, `mxIsCell`, `mxSetCell`

**Introduced before R2006a**

# mxGetChars (C)

Pointer to character array data

## C Syntax

```
#include "matrix.h"
mxChar *mxGetChars(const mxArray *array_ptr);
```

## Arguments

`array_ptr`

Pointer to an `mxArray`

## Returns

Pointer to the first character in the `mxArray`. Returns `NULL` if the specified array is not a character array.

## Description

Call `mxGetChars` to access the first character in the `mxArray` that `array_ptr` points to. Once you have the starting address, you can access any other element in the `mxArray`.

## See Also

`mxGetString`

**Introduced before R2006a**

# mxGetClassID (C)

Class of mxArray

## C Syntax

```
#include "matrix.h"
mxClassID mxGetClassID(const mxArray *pm);
```

## Description

Use `mxGetClassId` to determine the class of an `mxArray`. The class of an `mxArray` identifies the kind of data the `mxArray` is holding. For example, if `pm` points to a logical `mxArray`, then `mxGetClassId` returns `mxLOGICAL_CLASS` (in C).

`mxGetClassId` is like `mxGetClassName`, except that the former returns the class as an integer identifier and the latter returns the class as a string.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Output Arguments

**`ID` — Numeric identifier of class**
`mxClassID`

Numeric identifier of the class (category) of the `mxArray`, specified as `mxClassID`. For a list of C-language class identifiers, see the `mxClassID` function. For user-defined types, `mxGetClassId` returns a unique value identifying the class of the array contents. Use `mxIsClass` to determine whether an array is of a specific user-defined type.

## Examples

See these examples in *matlabroot*`/extern/examples/mex`:

- `explore.c`

See these examples in *matlabroot*`/extern/examples/refbook`:

- `phonebook.c`

## See Also

`mxClassID` | `mxGetClassName` | `mxIsClass`

**Introduced before R2006a**

# mxGetClassName (C)

Class of mxArray as string

**Note** Use `mxGetClassName` for classes defined without a `classdef` statement.

## C Syntax

```
#include "matrix.h"
const char *mxGetClassName(const mxArray *pm);
```

## Description

`mxGetClassName` returns the class of an `mxArray`. The class identifies the kind of data the `mxArray` is holding. For example, if `pm` points to a logical `mxArray`, `mxGetClassName` returns `logical`.

`mxGetClassID` is similar to the `mxGetClassName` function.

- `mxGetClassID` returns the class as an integer identifier, as described in `mxClassID`.
- `mxGetClassName` returns the class as a string, as described in `mxIsClass`.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Output Arguments

**name — Class name**
`const char*`

Class name, specified as `const char*`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `mexfunction.c`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxisclass.c`

## See Also

`mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxGetClassID (Fortran)

Class of mxArray

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxGetClassID(pm)
mwPointer pm
```

## Description

Use `mxGetClassId` to determine the class of an `mxArray`. The class of an `mxArray` identifies the kind of data the `mxArray` is holding. For example, if `pm` points to a logical `mxArray`, then `mxGetClassId` returns `mxLOGICAL_CLASS` (in C).

`mxGetClassId` is like `mxGetClassName`, except that the former returns the class as an integer identifier and the latter returns the class as a string.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## Output Arguments

**ID — Numeric identifier of class**
`integer*4`

Numeric identifier of the class (category) of the `mxArray`, specified as `integer*4`. For user-defined types, `mxGetClassId` returns a unique value identifying the class of the array contents. Use `mxIsClass` to determine whether an array is of a specific user-defined type.

## See Also

`mxClassIDFromClassName` | `mxGetClassName` | `mxIsClass`

**Introduced before R2006a**

# mxGetClassName (Fortran)

Class of mxArray as string

**Note** Use `mxGetClassName` for classes defined without a `classdef` statement.

## Fortran Syntax

```
#include "fintrf.h"
character*(*) mxGetClassName(pm)
mwPointer pm
```

## Description

`mxGetClassName` returns the class of an `mxArray`. The class identifies the kind of data the `mxArray` is holding. For example, if `pm` points to a logical `mxArray`, `mxGetClassName` returns `logical`.

`mxGetClassName` is similar to the `mxGetClassID` function.

- `mxGetClassName` returns the class as a string, as described in `mxIsClass`.
- `mxGetClassID` returns the class as an integer identifier, as described in `mxClassID`.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## Output Arguments

**name — Class name**
`character*(*)`

Class name, specified as `character*(*)`.

## See Also

`mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxGetData (C)

Data elements in nonnumeric `mxArray`

---

**Note** `mxGetData` is not recommended for numeric arrays. Use typed, data-access functions instead. For more information, see “Compatibility Considerations”.

---

## C Syntax

```
#include "matrix.h"
void *mxGetData(const mxArray *pm);
```

## Description

Use `mxGetData` to get data elements for nonnumeric arrays only.

For numeric arrays, MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- “Typed Data Access in C MEX Files”
- “MATLAB Support for Interleaved Complex API in MEX Functions”
- “Upgrade MEX Files to Use Interleaved Complex API”
- Example `explore.c`

To build the MEX file, call `mex` with the `-R2018a` option.

## Input Arguments

**pm — Pointer to nonnumeric MATLAB array**
`mxArray *`

Pointer to a nonnumeric MATLAB array, specified as `mxArray *`.

## Output Arguments

**pa — Pointer to data array**
`void *` | `NULL`

Pointer to the data array within an `mxArray`, specified as `void *`. Since void pointers point to a value that has no type, cast the return value to the pointer type that matches the type specified by `pm`. For information on mapping MATLAB types to their equivalent C types, see `mxClassID`.

If `pm` is `NULL`, then the function returns `NULL`.

## Compatibility Considerations

**For complex numeric `mxArray`, casting `mxGetData` return value depends on build option**
*Behavior changed in R2018a*

The `mxGetData` function returns a void pointer. Your code must declare a pointer type that matches the type specified by the `mxArray` input argument. Use `mxClassID` to choose the correct type. For complex numeric input, the correct type depends on the build option used to create the MEX file.

If you build the MEX file with the default release-specific option (`-R2017b`), then the function returns a pointer to the first element of the real-only values.

If you build the MEX file with the `-R2018a` option, then:

- When input argument `pm` points to a real MATLAB array, the function returns a pointer to the first element of the data.
- When `pm` is a complex array, the function returns a pointer to the first element of the interleaved real and imaginary values, not to the real-only values.

## See Also

`mxClassID`

**Topics**

`explore.c`
"Typed Data Access in C MEX Files"
"MATLAB Support for Interleaved Complex API in MEX Functions"

**Introduced before R2006a**

# mxGetData (Fortran)

Data elements in nonnumeric `mxArray`

**Note** `mxGetData` is not recommended for numeric arrays. Use typed, data-access functions instead. For more information, see "Compatibility Considerations".

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetData(pm)
mwPointer pm
```

## Description

Use `mxGetData` to get data elements for nonnumeric arrays only.

For numeric arrays, MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- "Typed Data Access in C MEX Files"
- "MATLAB Support for Interleaved Complex API in MEX Functions"
- "Upgrade MEX Files to Use Interleaved Complex API"

To build the MEX file, call `mex` with the `-R2018a` option.

## Input Arguments

### pm — Pointer to nonnumeric MATLAB array
`mwPointer`

Pointer to a nonnumeric MATLAB array, specified as `mwPointer`.

## Output Arguments

### pa — Pointer to data array
`mwPointer` | `0`

Pointer to the data array within an `mxArray`, specified as `mwPointer`. Since void pointers point to a value that has no type, cast the return value to the pointer type that matches the type specified by `pm`.

To copy values from the returned pointer, use one of the `mxCopyPtrTo*` functions. For example:

```
C      Get the data in mxArray, pm
       mxCopyPtrToReal8(mxGetData(pm),data,
     +                  mxGetNumberOfElements(pm))
```

If `pm` is `0`, then the function returns `0`.

## Compatibility Considerations

**For complex numeric `mxArray`, casting `mxGetData` return value depends on build option**
*Behavior changed in R2018b*

The `mxGetData` function returns `mwPointer`. Your code must declare a pointer type that matches the type specified by the `mxArray` input argument. For complex numeric input, the correct type depends on the build option used to create the MEX file.

If you build the MEX file with the default release-specific option (`-R2017b`), then the function returns a pointer to the first element of the real-only values.

If you build the MEX file with the `-R2018a` option, then:

- When input argument `pm` points to a real MATLAB array, the function returns a pointer to the first element of the data.
- When `pm` is a complex array, the function returns a pointer to the first element of the interleaved real and imaginary values, not to the real-only values.

## See Also

**Topics**
"Typed Data Access in C MEX Files"
"MATLAB Support for Interleaved Complex API in MEX Functions"

**Introduced before R2006a**

# mxGetDimensions (C)

Pointer to dimensions array

## C Syntax

```
#include "matrix.h"
const mwSize *mxGetDimensions(const mxArray *pm);
```

## Description

`mxGetDimensions` returns a pointer to the first element in the dimensions array. Each integer in the dimensions array represents the number of elements in a particular dimension. The array is not `NULL` terminated.

Use `mxGetDimensions` to determine how many elements are in each dimension of the `mxArray` that `pm` points to. Call `mxGetNumberOfDimensions` to get the number of dimensions in the `mxArray`.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcalcsinglesubscript.c`
- `mxgeteps.c`
- `mxisfinite.c`

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `findnz.c`
- `phonebook.c`

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## See Also
`mxGetNumberOfDimensions`

**Introduced before R2006a**

# mxGetDimensions (Fortran)

Pointer to dimensions array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetDimensions(pm)
mwPointer pm
```

## Description

`mxGetDimensions` returns a pointer to the first element in the dimensions array. Each integer in the dimensions array represents the number of elements in a particular dimension. The array is not `NULL` terminated.

Use `mxGetDimensions` to determine how many elements are in each dimension of the `mxArray` that `pm` points to. Call `mxGetNumberOfDimensions` to get the number of dimensions in the `mxArray`.

To copy the values to Fortran, use `mxCopyPtrToInteger4` as follows:

```
C      Get dimensions of mxArray, pm
       mxCopyPtrToInteger4(mxGetDimensions(pm), dims,
     +                     mxGetNumberOfDimensions(pm))
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxGetNumberOfDimensions`

**Introduced before R2006a**

# mxGetElementSize (C)

Number of bytes required to store each data element

**Note** For a complex `mxArray` built with the interleaved complex API, `mxGetElementSize` returns twice the value that the function in the separate complex API returns. For more information, see “Compatibility Considerations”.

## C Syntax

```
#include "matrix.h"
size_t mxGetElementSize(const mxArray *pm);
```

## Description

Call `mxGetElementSize` to determine the number of bytes in each data element of the `mxArray`. For example, if the MATLAB class of an `mxArray` is `int16`, the `mxArray` stores each data element as a 16-bit (2-byte) signed integer. Thus, `mxGetElementSize` returns `2`.

`mxGetElementSize` is helpful when using a non-MATLAB routine to manipulate data elements. For example, the C function `memcpy` requires the size of the elements you intend to copy.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray`, specified as `const mxArray*`.

## Output Arguments

**`nbytes` — Number of bytes**
`size_t | 0`

Number of bytes required to store one element of the specified `mxArray`, returned as `size_t`.

If `pm` is complex numeric, then the data in the output argument depends on which version of the C Matrix API you use.

- If you build with the interleaved complex API (`mex -R2018a` option), then the return value is `sizeof(std::complex<T>)`, where `T` is the data type of the array.
- If you build with the separate complex API (`mex -R2017b` option), then the function returns the number of bytes for the data type of the array regardless whether the array is complex or real.

If `pm` points to a cell or structure, then `mxGetElementSize` returns the size of a pointer. The function does not return the size of all the elements in each cell or structure field.

Returns 0 on failure. The primary reason for failure is that `pm` points to an `mxArray` having an unrecognized class.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `doubleelement.c`
- `phonebook.c`

## Compatibility Considerations

### `mxGetElementSize` returns different values based on build option

*Behavior changed in R2018a*

For a complex numeric `mxArray`, the `mxGetElementSize` function returns different values based on the `mex` build option. For more information, see the `nbytes` output argument.

## See Also

`mxGetM` | `mxGetN`

**Introduced before R2006a**

# mxGetElementSize (Fortran)

Number of bytes required to store each data element

**Note** For a complex `mxArray` built with the interleaved complex API, `mxGetElementSize` returns twice the value that the function in the separate complex API returns. For more information, see "Compatibility Considerations".

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetElementSize(pm)
mwPointer pm
```

## Description

Call `mxGetElementSize` to determine the number of bytes in each data element of the `mxArray`. For example, if the MATLAB class of an `mxArray` is `int16`, the `mxArray` stores each data element as a 16-bit (2-byte) signed integer. Thus, `mxGetElementSize` returns `2`.

`mxGetElementSize` is helpful when using a non-MATLAB routine to manipulate data elements.

**Note** Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray`, specified as `mwPointer`.

## Output Arguments

**nbytes — Number of bytes**
`integer*4 | 0`

Number of bytes required to store one element of the specified `mxArray`, returned as `integer*4`.

If `pm` is complex numeric, then the data in the output argument depends on which version of the Fortran Matrix API you use.

- If you build with the separate complex API (`mex -R2017b` option), then the function returns the number of bytes for the data type of the array regardless whether the array is complex or real.
- If you build with the interleaved complex API (`mex -R2018a` option), then the return value is twice the number of bytes for the data type.

If `pm` points to a cell or structure, then `mxGetElementSize` returns the size of a pointer. The function does not return the size of all the elements in each cell or structure field.

Returns 0 on failure. The primary reason for failure is that `pm` points to an `mxArray` having an unrecognized class.

## Compatibility Considerations

**`mxGetElementSize` returns different values based on build option**
*Behavior changed in R2018b*

For a complex numeric `mxArray`, the `mxGetElementSize` function returns different values based on the `mex` build option. For more information, see the `nbytes` output argument.

## See Also
`mxGetM` | `mxGetN`

**Introduced before R2006a**

# mxGetEps (C and Fortran)

Value of EPS

## C Syntax

```
#include "matrix.h"
double mxGetEps(void);
```

## Fortran Syntax

```
real*8 mxGetEps
```

## Returns

Value of the MATLAB `eps` variable

## Description

Call `mxGetEps` to return the value of the MATLAB `eps` variable. This variable holds the distance from 1.0 to the next largest floating-point number. As such, it is a measure of floating-point accuracy. The MATLAB `pinv` and `rank` functions use `eps` as a default tolerance.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxgeteps.c`
- `mxgetepsf.F`

## See Also

`mxGetInf`, `mxGetNan`

**Introduced before R2006a**

# mxGetField (C and Fortran)

Pointer to field value from structure array, given index and field name

## C Syntax

```
#include "matrix.h"
mxArray *mxGetField(const mxArray *pm, mwIndex index, const char *fieldname);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetField(pm, index, fieldname)
mwPointer pm
mwIndex index
character*(*) fieldname
```

## Arguments

`pm`

Pointer to a structure `mxArray`

`index`

Index of the desired element.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

`fieldname`

Name of the field whose value you want to extract.

## Returns

Pointer to the `mxArray` in the specified field at the specified `fieldname`, on success. Returns `NULL` in C (`0` in Fortran) if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include:

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. To determine whether `pm` points to a structure `mxArray`, call `mxIsStruct`.
- Specifying an `index` to an element outside the bounds of the `mxArray`. For example, given a structure `mxArray` that contains 10 elements, you cannot specify an `index` greater than `9` in C (`10` in Fortran).
- Specifying a nonexistent `fieldname`. Call `mxGetFieldNameByNumber` or `mxGetFieldNumber` to get existing field names.
- Insufficient heap space.

## Description

Call mxGetField to get the value held in the specified element of the specified field. In pseudo-C terminology, mxGetField returns the value at:

```
pm[index].fieldname
```

mxGetFieldByNumber is like mxGetField. Both functions return the same value. The only difference is in the way you specify the field. mxGetFieldByNumber takes a field number as its third argument, and mxGetField takes a field name as its third argument.

Do not call mxDestroyArray on an mxArray returned by the mxGetField function.

---

**Note** Inputs to a MEX-file are constant read-only mxArrays. Do not modify the inputs. Using mxSetCell* or mxSetField* functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

---

In C, calling:

```
mxGetField(pa, index, "field_name");
```

is equivalent to calling:

```
field_num = mxGetFieldNumber(pa, "field_name");
mxGetFieldByNumber(pa, index, field_num);
```

where, if you have a 1-by-1 structure, index is 0.

In Fortran, calling:

```
mxGetField(pm, index, 'fieldname')
```

is equivalent to calling:

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxGetFieldByNumber(pm, index, fieldnum)
```

where, if you have a 1-by-1 structure, index is 1.

## Examples

See the following example in *matlabroot*/extern/examples/eng_mat.

- matreadstructarray.c

## See Also

mxGetFieldByNumber, mxGetFieldNameByNumber, mxGetFieldNumber, mxGetNumberOfFields, mxIsStruct, mxSetField, mxSetFieldByNumber

**Introduced before R2006a**

# mxGetFieldByNumber (C and Fortran)

Pointer to field value from structure array, given index and field number

## C Syntax

```
#include "matrix.h"
mxArray *mxGetFieldByNumber(const mxArray *pm, mwIndex index, int fieldnumber);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetFieldByNumber(pm, index, fieldnumber)
mwPointer pm
mwIndex index
integer*4 fieldnumber
```

## Arguments

`pm`

Pointer to a structure `mxArray`

`index`

Index of the desired element.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

See `mxCalcSingleSubscript` for more details on calculating an `index`.

`fieldnumber`

Position of the field whose value you want to extract

In C, the first field within each element has a field number of `0`, the second field has a field number of `1`, and so on. The last field has a field number of `N-1`, where `N` is the number of fields.

In Fortran, the first field within each element has a field number of `1`, the second field has a field number of `2`, and so on. The last field has a field number of `N`, where `N` is the number of fields.

## Returns

Pointer to the `mxArray` in the specified field for the desired element, on success. Returns `NULL` in C (`0` in Fortran) if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include:

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.

- Specifying an `index` to an element outside the bounds of the `mxArray`. For example, given a structure `mxArray` that contains 10 elements, you cannot specify an `index` greater than `9` in C (`10` in Fortran).
- Specifying a nonexistent field number. Call `mxGetFieldNumber` to determine the field number that corresponds to a given field name.

## Description

Call `mxGetFieldByNumber` to get the value held in the specified `fieldnumber` at the indexed element.

Do not call `mxDestroyArray` on an `mxArray` returned by the `mxGetFieldByNumber` function.

---

**Note** Inputs to a MEX-file are constant read-only `mxArray`s. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

---

In C, if you have a `1`-by-`1` structure, then calling:

```
mxGetField(pa, index, "field_name");
```

is equivalent to calling:

```
field_num = mxGetFieldNumber(pa, "field_name");
mxGetFieldByNumber(pa, index, field_num);
```

where `index` is `0`.

In Fortran, if you have a `1`-by-`1` structure, then calling:

```
mxGetField(pm, index, 'fieldname')
```

is equivalent to calling:

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxGetFieldByNumber(pm, index, fieldnum)
```

where `index` is `1`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `phonebook.c`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxisclass.c`

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## See Also

`mxGetField`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

**Introduced before R2006a**

# mxGetFieldNameByNumber (C and Fortran)

Pointer to field name from structure array, given field number

## C Syntax

```
#include "matrix.h"
const char *mxGetFieldNameByNumber(const mxArray *pm, int fieldnumber);
```

## Fortran Syntax

```
#include "fintrf.h"
character*(*) mxGetFieldNameByNumber(pm, fieldnumber)
mwPointer pm
integer*4 fieldnumber
```

## Arguments

`pm`

Pointer to a structure `mxArray`

`fieldnumber`

Position of the desired field. For instance, in C, to get the name of the first field, set `fieldnumber` to 0; to get the name of the second field, set `fieldnumber` to 1; and so on. In Fortran, to get the name of the first field, set `fieldnumber` to `1`; to get the name of the second field, set `fieldnumber` to `2`; and so on.

## Returns

Pointer to the `n`th field name, on success. Returns `NULL` in C (`0` in Fortran) on failure. Common causes of failure include

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.
- Specifying a value of `fieldnumber` outside the bounds of the number of fields in the structure `mxArray`. In C, `fieldnumber` 0 represents the first field, and `fieldnumber` `N-1` represents the last field, where `N` is the number of fields in the structure `mxArray`. In Fortran, `fieldnumber` `1` represents the first field, and `fieldnumber` `N` represents the last field.

## Description

Call `mxGetFieldNameByNumber` to get the name of a field in the given structure `mxArray`. A typical use of `mxGetFieldNameByNumber` is to call it inside a loop to get the names of all the fields in a given `mxArray`.

Consider a MATLAB structure initialized to:

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

In C, the field number `0` represents the field `name`; field number `1` represents field `billing`; field number `2` represents field `test`. A field number other than `0`, `1`, or `2` causes `mxGetFieldNameByNumber` to return `NULL`.

In Fortran, the field number `1` represents the field `name`; field number `2` represents field `billing`; field number `3` represents field `test`. A field number other than `1`, `2`, or `3` causes `mxGetFieldNameByNumber` to return `0`.

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `phonebook.c`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxisclass.c`

See these examples in *matlabroot*`/extern/examples/mex`:

- `explore.c`

## See Also

`mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

**Introduced before R2006a**

# mxGetFieldNumber (C and Fortran)

Field number from structure array, given field name

## C Syntax

```
#include "matrix.h"
int mxGetFieldNumber(const mxArray *pm, const char *fieldname);
```

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxGetFieldNumber(pm, fieldname)
mwPointer pm
character*(*) fieldname
```

## Arguments

`pm`

Pointer to a structure `mxArray`

`fieldname`

Name of a field in the structure `mxArray`

## Returns

Field number of the specified `fieldname`, on success. In C, the first field has a field number of `0`, the second field has a field number of `1`, and so on. In Fortran, the first field has a field number of `1`, the second field has a field number of `2`, and so on. Returns `-1` in C (`0` in Fortran) on failure. Common causes of failure include

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.
- Specifying the `fieldname` of a nonexistent field.

## Description

If you know the name of a field but do not know its field number, call `mxGetFieldNumber`. Conversely, if you know the field number but do not know its field name, call `mxGetFieldNameByNumber`.

For example, consider a MATLAB structure initialized to:

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

In C, the field `name` has a field number of `0`; the field `billing` has a field number of `1`; and the field `test` has a field number of `2`. If you call `mxGetFieldNumber` and specify a field name of anything other than `name`, `billing`, or `test`, `mxGetFieldNumber` returns `-1`.

If you have a 1-by-1 structure, then calling:

```
mxGetField(pa, index, "field_name");
```

is equivalent to calling:

```
field_num = mxGetFieldNumber(pa, "field_name");
mxGetFieldByNumber(pa, index, field_num);
```

where `index` is `0`.

In Fortran, the field `name` has a field number of `1`; the field `billing` has a field number of `2`; and the field `test` has a field number of `3`. If you call `mxGetFieldNumber` and specify a field name of anything other than `name`, `billing`, or `test`, `mxGetFieldNumber` returns `0`.

If you have a 1-by-1 structure, then calling:

```
mxGetField(pm, index, 'fieldname');
```

is equivalent to calling:

```
fieldnum = mxGetFieldNumber(pm, 'fieldname');
mxGetFieldByNumber(pm, index, fieldnum);
```

where `index` is `1`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreatestructarray.c`

## See Also

`mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

**Introduced before R2006a**

# mxGetImagData (C)

Imaginary data elements in numeric `mxArray`

**Note** `mxGetImagData` is not available in the interleaved complex API. Use typed, data-access functions instead. For more information, see “Compatibility Considerations”.

## C Syntax

```
#include "matrix.h"
void *mxGetImagData(const mxArray *pm);
```

## Description

The `mxGetImagData` function is similar to `mxGetPi`, except that in C it returns a `void *`. For more information, see the description for the `mxGetData` function.

## Input Arguments

### `pm` — Pointer to MATLAB array

`mxArray *`

Pointer to a MATLAB array, specified as `mxArray *`.

## Output Arguments

### `pi` — Pointer to complex data array

`void *` | `NULL`

Pointer to the complex data array within an `mxArray`, specified as `void *`. Since void pointers point to a value that has no type, cast the return value to the pointer type that matches the type specified by `pm`. For information on mapping MATLAB types to their equivalent C types, see `mxClassID`.

If `pm` is `NULL`, then the function returns `NULL`.
Complex Number Support: Yes

## Compatibility Considerations

### Do not use separate complex API

*Not recommended starting in R2018a*

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. These functions verify that the input array is complex and of the correct type for the function. For more information, see:

- “Typed Data Access in C MEX Files”
- “MATLAB Support for Interleaved Complex API in MEX Functions”

- “Upgrade MEX Files to Use Interleaved Complex API”
- Example `explore.c`

To build the MEX file, call `mex` with the `-R2018a` option.

**Error building `mxGetImagData` with interleaved complex API**
*Errors starting in R2018a*

The `mxGetImagData` function is only available in the separate complex API. To build `myMexFile.c` using this function, type:

```
mex -R2017b myMexFile.c
```

Existing MEX files built with this function continue to run.

## See Also
`mxClassID`

**Topics**
`explore.c`
“Typed Data Access in C MEX Files”
“MATLAB Support for Interleaved Complex API in MEX Functions”

**Introduced before R2006a**

# mxGetImagData (Fortran)

Imaginary data elements in numeric `mxArray`

**Note** `mxGetImagData` is not available in the interleaved complex API. Use typed, data-access functions instead. For more information, see "Compatibility Considerations".

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetImagData(pm)
mwPointer pm
```

## Description

The `mxGetImagData` function is similar to `mxGetPi`, except that it returns a `mwPointer`. For more information, see the description for the `mxGetData` function.

## Input Arguments

**pm — Pointer to MATLAB array**
`mwPointer`

Pointer to a MATLAB array, specified as `mwPointer`.

## Output Arguments

**pi — Pointer to complex data array**
`mwPointer | 0`

Pointer to the complex data array within an `mxArray`, specified as `mwPointer`. Since void pointers point to a value that has no type, cast the return value to the pointer type that matches the type specified by `pm`.

If `pm` is `0`, then the function returns `0`.
Complex Number Support: Yes

## Compatibility Considerations

**Do not use separate complex API**
*Not recommended starting in R2018b*

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. These functions verify that the input array is complex and of the correct type for the function. For more information, see:

- "Typed Data Access in C MEX Files"
- "MATLAB Support for Interleaved Complex API in MEX Functions"

- “Upgrade MEX Files to Use Interleaved Complex API”

To build the MEX file, call `mex` with the `-R2018a` option.

**Error building `mxGetImagData` with interleaved complex API**
*Errors starting in R2018b*

The `mxGetImagData` function is only available in the separate complex API. To build `myMexFile.F` using this function, type:

```
mex -R2017b myMexFile.F
```

Existing MEX files built with this function continue to run.

## See Also

**Topics**
“Typed Data Access in C MEX Files”
“Upgrade MEX Files to Use Interleaved Complex API”

**Introduced before R2006a**

# mxGetInf (C and Fortran)

Value of infinity

## C Syntax

```
#include "matrix.h"
double mxGetInf(void);
```

## Fortran Syntax

```
real*8 mxGetInf
```

## Returns

Value of infinity on your system.

## Description

Call `mxGetInf` to return the value of the MATLAB internal `inf` variable. `inf` is a permanent variable representing IEEE® arithmetic positive infinity. Your system specifies the value of `inf`; you cannot modify it.

Operations that return infinity include:

- Division by 0. For example, 5/0 returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxgetinf.c`

## See Also

`mxGetEps`, `mxGetNaN`

**Introduced before R2006a**

# mxGetIr (C and Fortran)

Sparse matrix IR array

## C Syntax

```
#include "matrix.h"
mwIndex *mxGetIr(const mxArray *pm);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetIr(pm)
mwPointer pm
```

## Arguments

`pm`

Pointer to a sparse `mxArray`

## Returns

Pointer to the first element in the `ir` array, if successful, and `NULL` in C (`0` in Fortran) otherwise. Possible causes of failure include:

- Specifying a full (nonsparse) `mxArray`.
- Specifying a value for `pm` that is `NULL` in C (`0` in Fortran). This failure usually means that an earlier call to `mxCreateSparse` failed.

## Description

Use `mxGetIr` to obtain the starting address of the `ir` array. The `ir` array is an array of integers. The length of `ir` is `nzmax`, the storage allocated for the sparse array, or `nnz`, the number of nonzero matrix elements. For example, if `nzmax` equals `100`, then the `ir` array contains 100 integers.

Each value in an `ir` array indicates a row (offset by 1) at which a nonzero element can be found. (The `jc` array is an index that indirectly specifies a column where nonzero elements can be found.)

For details on the `ir` and `jc` arrays, see `mxSetIr` and `mxSetJc`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `fulltosparse.c`
- `fulltosparse.F`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxsetdimensions.c`
- `mxsetnzmax.c`

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## See Also

`mxGetJc`, `mxGetNzmax`, `mxSetIr`, `mxSetJc`, `mxSetNzmax`, `nzmax`, `nnz`

**Introduced before R2006a**

# mxGetJc (C and Fortran)

Sparse matrix JC array

## C Syntax

```
#include "matrix.h"
mwIndex *mxGetJc(const mxArray *pm);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetJc(pm)
mwPointer pm
```

## Arguments

`pm`

Pointer to a sparse `mxArray`

## Returns

Pointer to the first element in the `jc` array, if successful, and `NULL` in C (`0` in Fortran) otherwise. Possible causes of failure include

- Specifying a full (nonsparse) `mxArray`.
- Specifying a value for `pm` that is `NULL` in C (`0` in Fortran). This failure usually means that an earlier call to `mxCreateSparse` failed.

## Description

Use `mxGetJc` to obtain the starting address of the `jc` array. The `jc` array is an integer array having `n+1` elements, where `n` is the number of columns in the sparse `mxArray`. The values in the `jc` array indirectly indicate columns containing nonzero elements. For a detailed explanation of the `jc` array, see `mxSetJc`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `fulltosparse.c`
- `fulltosparse.F`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxgetnzmax.c`
- `mxsetdimensions.c`

- `mxsetnzmax.c`

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## See Also

`mxGetIr`, `mxGetNzmax`, `mxSetIr`, `mxSetJc`, `mxSetNzmax`

**Introduced before R2006a**

# mxGetLogicals (C)

Pointer to logical array data

## C Syntax

```
#include "matrix.h"
mxLogical *mxGetLogicals(const mxArray *array_ptr);
```

## Arguments

`array_ptr`

Pointer to an `mxArray`

## Returns

Pointer to the first logical element in the `mxArray`. The result is unspecified if the `mxArray` is not a logical array.

## Description

Call `mxGetLogicals` to access the first logical element in the `mxArray` that `array_ptr` points to. Once you have the starting address, you can access any other element in the `mxArray`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxislogical.c`

## See Also

`mxCreateLogicalArray`, `mxCreateLogicalMatrix`, `mxCreateLogicalScalar`, `mxIsLogical`, `mxIsLogicalScalar`, `mxIsLogicalScalarTrue`

**Introduced before R2006a**

# mxGetM (C)

Number of rows in mxArray

## C Syntax

```
#include "matrix.h"
size_t mxGetM(const mxArray *pm);
```

## Description

`mxGetM` returns the number of rows in the specified array. The term *rows* always means the first dimension of the array, no matter how many dimensions the array has. For example, if `pm` points to a four-dimensional array having dimensions 8-by-9-by-5-by-3, then `mxGetM` returns 8.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `convec.c`
- `fulltosparse.c`
- `matrixDivide.c`
- `matrixDivideComplex.c`
- `revord.c`
- `timestwo.c`
- `xtimesy.c`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxmalloc.c`
- `mxsetdimensions.c`
- `mxgetnzmax.c`
- `mxsetnzmax.c`

See these examples in *matlabroot*`/extern/examples/mex`:

- `explore.c`
- `mexlock.c`
- `yprime.c`

## See Also

`mxGetN` | `mxSetM` | `mxSetN`

**Introduced before R2006a**

# mxGetM (Fortran)

Number of rows in mxArray

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetM(pm)
mwPointer pm
```

## Description

`mxGetM` returns the number of rows in the specified array. The term *rows* always means the first dimension of the array, no matter how many dimensions the array has. For example, if `pm` points to a four-dimensional array having dimensions 8-by-9-by-5-by-3, then `mxGetM` returns 8.

**Note** Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `convec.F`
- `dblmat.F`
- `fulltosparse.F`
- `matsq.F`
- `timestwo.F`
- `xtimesy.F`

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matdemo2.F`

## See Also
`mxGetN` | `mxSetM` | `mxSetN`

**Introduced before R2006a**

# mxGetN (C)

Number of columns in mxArray

## C Syntax

```
#include "matrix.h"
size_t mxGetN(const mxArray *pm);
```

## Description

`mxGetN` returns the number of columns in the specified `mxArray`.

If `pm` is an N-dimensional `mxArray`, `mxGetN` is the product of dimensions 2 through N. For example, if `pm` points to a four-dimensional `mxArray` having dimensions 13-by-5-by-4-by-6, `mxGetN` returns the value `120` (5 × 4 × 6). If the specified `mxArray` has more than two dimensions, then call `mxGetDimensions` to find out how many elements are in each dimension.

If `pm` points to a sparse `mxArray`, `mxGetN` still returns the number of columns, not the number of occupied columns.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `convec.c`
- `fulltosparse.c`
- `revord.c`
- `timestwo.c`
- `xtimesy.c`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxmalloc.c`
- `mxsetdimensions.c`
- `mxgetnzmax.c`
- `mxsetnzmax.c`

See these examples in *matlabroot*`/extern/examples/mex`:

- `explore.c`

- `mexlock.c`
- `yprime.c`
- `matdemo2.F`

## See Also

`mxGetDimensions` | `mxGetM` | `mxSetM` | `mxSetN`

**Introduced before R2006a**

# mxGetN (Fortran)

Number of columns in mxArray

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetN(pm)
mwPointer pm
```

## Description

`mxGetN` return the numbers of columns in the specified `mxArray`.

If `pm` is an N-dimensional `mxArray`, `mxGetN` is the product of dimensions 2 through N. For example, if `pm` points to a four-dimensional `mxArray` having dimensions `13`-by-`5`-by-`4`-by-`6`, `mxGetN` returns the value `120` (5 × 4 × 6). If the specified `mxArray` has more than two dimensions, then call `mxGetDimensions` to find out how many elements are in each dimension.

If `pm` points to a sparse `mxArray`, `mxGetN` still returns the number of columns, not the number of occupied columns.

**Note** Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matdemo2.F`

## See Also

`mxGetDimensions` | `mxGetM` | `mxSetM` | `mxSetN`

**Introduced before R2006a**

# mxGetNaN (C and Fortran)

Value of NaN (Not-a-Number)

## C Syntax

```
#include "matrix.h"
double mxGetNaN(void);
```

## Fortran Syntax

```
real*8 mxGetNaN
```

## Returns

Value of `NaN` (Not-a-Number) on your system

## Description

Call `mxGetNaN` to return the value of `NaN` for your system. `NaN` is the IEEE arithmetic representation for Not-a-Number. Certain mathematical operations return `NaN` as a result, for example,

- `0.0/0.0`
- `Inf-Inf`

Your system specifies the value of Not-a-Number. You cannot modify it.

## C Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxgetinf.c`

## See Also

`mxGetEps`, `mxGetInf`

**Introduced before R2006a**

# mxGetNumberOfDimensions (C)

Number of dimensions in mxArray

## C Syntax

```
#include "matrix.h"
mwSize mxGetNumberOfDimensions(const mxArray *pm);
```

## Description

`mxGetNumberOfDimensions` returns the number of dimensions in the specified `mxArray`. The returned value is always `2` or greater.

To determine how many elements are in each dimension, call `mxGetDimensions`.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *matlabroot*`/extern/examples/mex`:

- `explore.c`

See these examples in *matlabroot*`/extern/examples/refbook`:

- `findnz.c`
- `fulltosparse.c`
- `phonebook.c`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxcalcsinglesubscript.c`
- `mxgeteps.c`
- `mxisfinite.c`

## See Also
`mxGetDimensions` | `mxSetM` | `mxSetN`

**Introduced before R2006a**

# mxGetNumberOfDimensions (Fortran)

Number of dimensions in mxArray

## Fortran Syntax

```
#include "fintrf.h"
mwSize mxGetNumberOfDimensions(pm)
mwPointer pm
```

## Description

`mxGetNumberOfDimensions` returns the number of dimensions in the specified `mxArray`. The returned value is always `2` or greater.

To determine how many elements are in each dimension, call `mxGetDimensions`.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxGetDimensions` | `mxSetM` | `mxSetN`

**Introduced before R2006a**

# mxGetNumberOfElements (C)

Number of elements in numeric mxArray

## C Syntax

```
#include "matrix.h"
size_t mxGetNumberOfElements(const mxArray *pm);
```

## Description

`mxGetNumberOfElements` returns the number of elements in the specified `mxArray`, returned as `size_t`. For example, if the dimensions of an array are 3-by-5-by-10, then `mxGetNumberOfElements` returns the number `150`.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `findnz.c`
- `phonebook.c`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcalcsinglesubscript.c`
- `mxgeteps.c`
- `mxgetinf.c`
- `mxisfinite.c`
- `mxsetdimensions.c`

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## See Also

`mxGetClassID` | `mxGetClassName` | `mxGetDimensions` | `mxGetM` | `mxGetN`

**Introduced before R2006a**

# mxGetNumberOfElements (Fortran)

Number of elements in numeric mxArray

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetNumberOfElements(pm)
mwPointer pm
```

## Description

`mxGetNumberOfElements` returns the number of elements in the specified `mxArray`, returned as the appropriate Fortran type. For example, if the dimensions of an array are 3-by-5-by-10, then `mxGetNumberOfElements` returns the number `150`.

**Note** Fortran does not have an equivalent of `size_t`. `mwPointer` is a preprocessor macro that provides the appropriate Fortran type. The value returned by this function, however, is not a pointer.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxgetepsf.F`
- `mxsetdimensionsf.F`

## See Also

`mxGetClassID` | `mxGetClassName` | `mxGetDimensions` | `mxGetM` | `mxGetN`

**Introduced before R2006a**

# mxGetNumberOfFields (C and Fortran)

Number of fields in structure array

## C Syntax

```
#include "matrix.h"
int mxGetNumberOfFields(const mxArray *pm);
```

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxGetNumberOfFields(pm)
mwPointer pm
```

## Arguments

`pm`

Pointer to a structure `mxArray`

## Returns

Number of fields, on success. Returns `0` on failure. The most common cause of failure is that `pm` is not a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` is a structure.

## Description

Call `mxGetNumberOfFields` to determine how many fields are in the specified structure `mxArray`.

Once you know the number of fields in a structure, you can loop through every field to set or to get field values.

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `phonebook.c`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxisclass.c`

See these examples in *matlabroot*`/extern/examples/mex`:

- `explore.c`

## See Also

`mxGetField`, `mxIsStruct`, `mxSetField`

**Introduced before R2006a**

# mxGetNzmax (C and Fortran)

Number of elements in IR, PR, and PI arrays

## C Syntax

```
#include "matrix.h"
mwSize mxGetNzmax(const mxArray *pm);
```

## Fortran Syntax

```
#include "fintrf.h"
mwSize mxGetNzmax(pm)
mwPointer pm
```

## Arguments

`pm`

Pointer to a sparse `mxArray`

## Returns

Number of elements allocated to hold nonzero entries in the specified sparse `mxArray`, on success. Returns an indeterminate value on error. The most likely cause of failure is that `pm` points to a full (nonsparse) `mxArray`.

## Description

Use `mxGetNzmax` to get the value of the `nzmax` field. The `nzmax` field holds an integer value that signifies the number of elements in the `ir`, `pr`, and, if it exists, the `pi` arrays. The value of `nzmax` is always greater than or equal to the number of nonzero elements in a sparse `mxArray`. In addition, the value of `nzmax` is always less than or equal to the number of rows times the number of columns.

As you adjust the number of nonzero elements in a sparse `mxArray`, MATLAB software often adjusts the value of the `nzmax` field. MATLAB adjusts `nzmax` to reduce the number of costly reallocations and to optimize its use of heap space.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxgetnzmax.c`
- `mxsetnzmax.c`

## See Also

`mxSetNzmax`

**Introduced before R2006a**

# mxGetPi (C)

(Not recommended) Imaginary data elements in mxDOUBLE_CLASS array

**Note** mxGetPi is not available in the interleaved complex API. Use mxGetComplexDoubles instead. For more information, see "Compatibility Considerations".

## C Syntax

```
#include "matrix.h"
mxDouble *mxGetPi(const mxArray *pm);
```

## Description

When building MEX files using the separate complex API, call mxGetPi to get the contents of the pi field. pi is an array containing the imaginary data of the mxArray. Use mxGetPi on arrays of type mxDOUBLE_CLASS only. For other numeric mxArray types, use mxGetImagData.

Call mxIsDouble to validate the mxArray type. Call mxIsComplex to determine whether the data is complex.

If any of the input matrices to a function are complex, then MATLAB allocates the imaginary parts of all input matrices.

## Input Arguments

**pm — Pointer to MATLAB array**
mxArray *

Pointer to a MATLAB array of type mxDOUBLE_CLASS, specified as mxArray *.
Complex Number Support: Yes

## Output Arguments

**pi — Pointer to data array**
mxDouble * | NULL

Pointer to the first mxDouble element of the imaginary part of the data array within an mxArray, specified as mxDouble *. The function returns NULL if no imaginary data exists or if an error occurs.
Complex Number Support: Yes

## Compatibility Considerations

**Do not use separate complex API**
*Not recommended starting in R2018a*

Use the mxGetComplexDoubles function in the interleaved complex API instead of the mxGetPr and mxGetPi functions. This function verifies that the input array is complex and of type mxDOUBLE_CLASS.

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- “Typed Data Access in C MEX Files”
- “MATLAB Support for Interleaved Complex API in MEX Functions”
- “Upgrade MEX Files to Use Interleaved Complex API”
- Example `convec.c`

To build the MEX file, call `mex` with the `-R2018a` option.

**Error building `mxGetPi` with interleaved complex API**
*Errors starting in R2018a*

The `mxGetPi` function is only available in the separate complex API. To build `myMexFile.c` using this function, type:

```
mex -R2017b myMexFile.c
```

Existing MEX files built with this function continue to run.

## See Also

`mxGetComplexDoubles`

### Topics

`convec.c`
“Typed Data Access in C MEX Files”
“MATLAB Support for Interleaved Complex API in MEX Functions”

**Introduced before R2006a**

# mxGetPi (Fortran)

(Not recommended) Imaginary data elements in mxDOUBLE_CLASS array

---

**Note** mxGetPi is not available in the interleaved complex API. Use mxGetComplexDoubles instead. For more information, see "Compatibility Considerations".

---

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetPi(pm)
mwPointer pm
```

## Description

When building MEX files using the separate complex API, call mxGetPi to get the contents of the pi field. pi is an array containing the imaginary data of the mxArray. Use mxGetPi on arrays of type mxDOUBLE_CLASS only. For other numeric mxArray types, use mxGetImagData.

Call mxIsDouble to validate the mxArray type. Call mxIsComplex to determine whether the data is complex.

If any of the input matrices to a function are complex, then MATLAB allocates the imaginary parts of all input matrices.

## Input Arguments

**pm — Pointer to MATLAB array**
mwPointer

Pointer to a MATLAB array of type mxDOUBLE_CLASS, specified as mwPointer.
Complex Number Support: Yes

## Output Arguments

**pi — Pointer to data array**
mwPointer | 0

Pointer to the first mxDouble element of the imaginary part of the data array within an mxArray, specified as mwPointer. The function returns 0 if no imaginary data exists or if an error occurs.
Complex Number Support: Yes

## Compatibility Considerations

**Do not use separate complex API**
*Not recommended starting in R2018b*

Use the `mxGetComplexDoubles` function in the interleaved complex API instead of the `mxGetPr` and `mxGetPi` functions. This function verifies that the input array is complex and of type `mxDOUBLE_CLASS`.

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- "Typed Data Access in C MEX Files"
- "MATLAB Support for Interleaved Complex API in MEX Functions"
- "Upgrade MEX Files to Use Interleaved Complex API"
- Example `convec.F`

To build the MEX file, call `mex` with the `-R2018a` option.

**Error building `mxGetPi` with interleaved complex API**
*Errors starting in R2018b*

The `mxGetPi` function is only available in the separate complex API. To build `myMexFile.F` using this function, type:

```
mex -R2017b myMexFile.F
```

Existing MEX files built with this function continue to run.

## See Also

`mxGetComplexDoubles`

**Topics**

`convec.F`
"Typed Data Access in C MEX Files"
"MATLAB Support for Interleaved Complex API in MEX Functions"

**Introduced before R2006a**

# mxGetPr (C)

(Not recommended) Real data elements in `mxDOUBLE_CLASS` array

---

**Note** `mxGetPr` is not recommended. Use `mxGetDoubles` or `mxGetComplexDoubles` instead. For more information, see "Compatibility Considerations".

---

## C Syntax

```
#include "matrix.h"
mxDouble *mxGetPr(const mxArray *pm);
```

## Description

Use `mxGetPr` on real arrays of type `mxDOUBLE_CLASS` only. For other numeric `mxArray` types, use "Typed Data Access in C MEX Files" functions. For complex arrays, see the description for output argument dt on page 1-0 .

Call `mxIsDouble` to validate the `mxArray` type. Call `mxIsComplex` to determine whether the data is real.

## Input Arguments

### pm — Pointer to MATLAB array

`mxArray *`

Pointer to a MATLAB array of type `mxDOUBLE_CLASS`, specified as `mxArray *`.

## Output Arguments

### dt — Pointer to data array

`mxDouble *` | `NULL`

Pointer to the data array within an `mxArray`, specified as `mxDouble *`. The data in the output argument depends on which version of the C Matrix API you use:

- If you build with the separate complex API (`mex -R2017b` option), then the function returns a pointer to the first `mxDouble` element of the real part of the data.
- If you build with the interleaved complex API (`mex -R2018a` option) and `pm` is complex, then the function terminates the MEX file and returns control to the MATLAB prompt. In a non-MEX file application, the function returns `NULL`.

If `pm` is `NULL`, then the function returns `NULL`.

## Compatibility Considerations

### Do not use separate complex API

*Not recommended starting in R2018a*

Use the `mxGetDoubles` function in the interleaved complex API for real input arrays of type `mxDOUBLE_CLASS`. Use `mxGetComplexDoubles` for complex input arrays of type `mxDOUBLE_CLASS`. These functions validate the type and complexity of the input.

MathWorks recommends that you create MEX files and update existing MEX files using the typed, data-access functions in the interleaved complex API. For more information, see:

- “Typed Data Access in C MEX Files”
- “MATLAB Support for Interleaved Complex API in MEX Functions”
- “Upgrade MEX Files to Use Interleaved Complex API”
- Example `xtimesy.c`

To build the MEX file, call `mex` with the `-R2018a` option.

**Runtime error calling `mxGetPr` on complex `mxArrays` in applications built with interleaved complex API**
*Errors starting in R2018a*

Use the `mxGetComplexDoubles` function instead of `mxGetPr` and `mxGetPi`. For more information, see the dt on page 1-0 output argument. For an example showing how to update code that uses `mxGetPr`, see `convec.c`.

## See Also

`mxGetComplexDoubles` | `mxGetDoubles`

**Topics**
`convec.c`
`xtimesy.c`
“Typed Data Access in C MEX Files”
“MATLAB Support for Interleaved Complex API in MEX Functions”

**Introduced before R2006a**

# mxGetPr (Fortran)

(Not recommended) Real data elements in `mxDOUBLE_CLASS` array

---

**Note** `mxGetPr` is not recommended. Use `mxGetDoubles` or `mxGetComplexDoubles` instead. For more information, see “Compatibility Considerations”.

---

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetPr(pm)
mwPointer pm
```

## Description

Use `mxGetPr` on real arrays of type `mxDOUBLE_CLASS` only. For other numeric `mxArray` types, use “Typed Data Access in C MEX Files” functions. For complex arrays, see the description for output argument `dt`.

Call `mxIsDouble` to validate the `mxArray` type. Call `mxIsComplex` to determine whether the data is real.

## Input Arguments

**`pm` — Pointer to MATLAB array**
`mwPointer`

Pointer to a MATLAB array of type `mxDOUBLE_CLASS`, specified as `mwPointer`.

## Output Arguments

**`dt` — Pointer to data array**
`mwPointer | 0`

Pointer to the data array within an `mxArray`, specified as `mwPointer`. The data in the output argument depends on which version of the Fortran Matrix API you use:

- If you build with the separate complex API (`mex -R2017b` option), then the function returns a pointer to the first `mxDouble` element of the real part of the data.
- If you build with the interleaved complex API (`mex -R2018a` option) and `pm` is complex, then the function terminates the MEX file and returns control to the MATLAB prompt. In a non-MEX file application, the function returns `0`.

If `pm` is `0`, then the function returns `0`.

## Compatibility Considerations

**Do not use separate complex API**
*Not recommended starting in R2018b*

Use the `mxGetDoubles` function in the interleaved complex API for real input arrays of type `mxDOUBLE_CLASS`. Use `mxGetComplexDoubles` for complex input arrays of type `mxDOUBLE_CLASS`. These functions validate the type and complexity of the input.

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- "Typed Data Access in C MEX Files"
- "MATLAB Support for Interleaved Complex API in MEX Functions"
- "Upgrade MEX Files to Use Interleaved Complex API"
- Example `xtimesy.F`

To build the MEX file, call `mex` with the `-R2018a` option.

**Runtime error calling `mxGetPr` on complex `mxArrays` in applications built with interleaved complex API**
*Errors starting in R2018b*

Use the `mxGetComplexDoubles` function instead of `mxGetPr` and `mxGetPi`. For more information, see the `dt` output argument. For an example showing how to update code that uses `mxGetPr`, see `convec.F`.

## See Also

`mxGetComplexDoubles` | `mxGetDoubles`

**Topics**
`convec.F`
`xtimesy.F`
"Typed Data Access in C MEX Files"
"MATLAB Support for Interleaved Complex API in MEX Functions"

**Introduced before R2006a**

# mxGetProperty (C and Fortran)

Value of public property of MATLAB object

## C Syntax

```
#include "matrix.h"
mxArray *mxGetProperty(const mxArray *pa, mwIndex index,
         const char *propname);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetProperty(pa, index, propname)
mwPointer pa
mwIndex index
character*(*) propname
```

## Arguments

`pa`

Pointer to an `mxArray` which is an object.

`index`

Index of the desired element of the object array.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

`propname`

Name of the property whose value you want to extract.

## Returns

Pointer to the `mxArray` of the specified `propname` on success. Returns `NULL` in C (`0` in Fortran) if unsuccessful. Common causes of failure include:

- Specifying a nonexistent `propname`.
- Specifying a nonpublic `propname`.
- Specifying an `index` to an element outside the bounds of the `mxArray`. To test the index value, use `mxGetNumberOfElements` or `mxGetM` and `mxGetN`.
- Insufficient heap space.

## Description

Call `mxGetProperty` to get the value held in the specified element. In pseudo-C terminology, `mxGetProperty` returns the value at:

```
pa[index].propname
```

`mxGetProperty` makes a copy of the value. If the property uses a large amount of memory, then creating a copy might be a concern. There must be sufficient memory (in the heap) to hold the copy of the value.

## Examples

### Display Name Property of `timeseries` Object

Create a MEX file, `dispproperty.c`, in a folder on your MATLAB path.

```
/*=================================================================
 * dispproperty.c - Display timeseries Name property
 * This is a MEX file for MATLAB.
 * Copyright 2013 The MathWorks, Inc.
 * All rights reserved.
 *=================================================================*/

#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{
  /* Check for proper number of arguments. */
  if(nrhs!=1) {
    mexErrMsgIdAndTxt( "MATLAB:dispproperty:invalidNumInputs",
            "One input required.");
  } else if(nlhs>1) {
    mexErrMsgIdAndTxt( "MATLAB:dispproperty:maxlhs",
            "Too many output arguments.");
  }
  /* Check for timeseries object. */
  if (!mxIsClass(prhs[0], "timeseries")) {
    mexErrMsgIdAndTxt( "MATLAB:dispproperty:invalidClass",
            "Input must be timeseries object.");
  }
  plhs[0] = mxGetProperty(prhs[0],0,"Name");

}
```

Build the MEX file.

```
mex('-v','dispproperty.c')
```

Create a `timeseries` object.

```
ts = timeseries(rand(5, 4),'Name','LaunchData');
```

Display name.

```
tsname = dispproperty(ts)
```

```
tsname =
LaunchData
```

### Change Object Color

Open and build the `mxgetproperty.c` MEX file in the *`matlabroot`*`/extern/examples/mex` folder.

## Limitations

- `mxGetProperty` is not supported for standalone applications, such as applications built with the MATLAB engine API.
- Properties of type `datetime` are not supported.

## See Also

`mxGetM` | `mxGetN` | `mxGetNumberOfElements` | `mxSetProperty`

**Topics**

"matlab::engine::MATLABEngine::getProperty"

**Introduced in R2008a**

# mxGetScalar (C and Fortran)

Real component of first data element in array

## C Syntax

```
#include "matrix.h"
double mxGetScalar(const mxArray *pm);
```

## Fortran Syntax

```
#include "fintrf.h"
real*8 mxGetScalar(pm)
mwPointer pm
```

## Arguments

`pm`

Pointer to an `mxArray`; cannot be a cell `mxArray`, a structure `mxArray`, or an empty `mxArray`.

## Returns

The value of the first real (nonimaginary) element of the `mxArray`.

In C, `mxGetScalar` returns a `double`. If real elements in the `mxArray` are of a type other than `double`, then `mxGetScalar` automatically converts the scalar value into a `double`. To preserve the original data representation of the scalar, cast the return value to the desired data type.

If `pm` points to a sparse `mxArray`, then `mxGetScalar` returns the value of the first nonzero real element in the `mxArray`. If there are no nonzero elements, then the function returns `0`.

## Description

Call `mxGetScalar` to get the value of the first real (nonimaginary) element of the `mxArray`.

Usually you call `mxGetScalar` when `pm` points to an `mxArray` containing only one element (a scalar). However, `pm` can point to an `mxArray` containing many elements. If `pm` points to an `mxArray` containing multiple elements, then the function returns the value of the first real element. For example, if `pm` points to a two-dimensional `mxArray`, then `mxGetScalar` returns the value of the `(1,1)` element. If `pm` points to a three-dimensional `mxArray`, then the function returns the value of the `(1,1,1)` element; and so on.

Use `mxGetScalar` on a nonempty `mxArray` of type numeric, logical, or char only. To test for these conditions, use Matrix Library functions such as `mxIsEmpty`, `mxIsLogical`, `mxIsNumeric`, or `mxIsChar`.

If the input value to `mxGetScalar` is type `int64` or `uint64`, then the value might lose precision if it is greater than `flintmax`.

## Examples

See these examples in *matlabroot*/extern/examples/refbook:

- timestwoalt.c
- xtimesy.c

See these examples in *matlabroot*/extern/examples/mex:

- mexlock.c
- mexlockf.F

See these examples in *matlabroot*/extern/examples/mx:

- mxsetdimensions.c

## See Also

mxGetM, mxGetN, mxIsScalar

# mxGetString (C and Fortran)

`mxChar` array to C-style string or Fortran `character` array

## C Syntax

```
#include "matrix.h"
int mxGetString(const mxArray *pm, char *str, mwSize strlen);
```

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxGetString(pm, str, strlen)
mwPointer pm
character*(*) str
mwSize strlen
```

## Arguments

`pm`

Pointer to an `mxChar` array.

`str`

Starting location. `mxGetString` writes the character data into `str` and then, in C, terminates the string with a `NULL` character (in the manner of C strings). `str` can point to either dynamic or static memory.

`strlen`

Size in bytes of destination buffer pointed to by `str`. Typically, in C, you set `strlen` to `1` plus the number of elements in the `mxArray` to which `pm` points. To get the number of elements, use `mxGetM` or `mxGetN`.

Do not use with “Multibyte Encoded Characters” on page 1-445.

## Returns

`0` on success or if `strlen == 0`, and `1` on failure. Possible reasons for failure include:

- `mxArray` is not an `mxChar` array.
- `strlen` is not large enough to store the entire `mxArray`. If so, then the function returns `1` and truncates the string.

## Description

Call `mxGetString` to copy the character data of an `mxArray` into a C-style string in C or a `character` array in Fortran. The copied data starts at `str` and contains no more than `strlen-1` characters in C (no more than `strlen` characters in Fortran). In C, the C-style string is always terminated with a `NULL` character.

If the array contains multiple rows, then the function copies them into a single array, one column at a time.

#### Multibyte Encoded Characters

Use this function only with characters represented in single-byte encoding schemes. For characters represented in multibyte encoding schemes, use the C function `mxArrayToString`. Fortran applications must allocate sufficient space for the return string to avoid possible truncation.

## Examples

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxmalloc.c`

See these examples in *matlabroot*`/extern/examples/mex`:

- `explore.c`

See these examples in *matlabroot*`/extern/examples/refbook`:

- `revord.F`

## See Also

`mxArrayToString`, `mxCreateCharArray`, `mxCreateCharMatrixFromStrings`, `mxCreateString`, `mxGetChars`

**Introduced before R2006a**

# mxIsCell (C)

Determine whether mxArray is cell array

## C Syntax

```
#include "matrix.h"
bool mxIsCell(const mxArray *pm);
```

## Description

`mxIsCell` returns logical `1` (`true`) if the specified array is a cell array. Otherwise, it returns logical `0` (`false`).

In C, calling `mxIsCell` is equivalent to calling:

```
mxGetClassID(pm) == mxCELL_CLASS
```

---

**Note** `mxIsCell` does not answer the question “Is this `mxArray` a cell of a cell array?” An individual cell of a cell array can be of any type.

---

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## See Also

`mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxIsCell (Fortran)

Determine whether mxArray is cell array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsCell(pm)
mwPointer pm
```

## Description

`mxIsCell` returns `1` if the specified array is a cell array. Otherwise, it returns `0`.

In Fortran, calling `mxIsCell` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'cell'
```

---

**Note** `mxIsCell` does not answer the question "Is this `mxArray` a cell of a cell array?" An individual cell of a cell array can be of any type.

---

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxIsChar (C)

Determine whether input is mxChar array

## C Syntax

```
#include "matrix.h"
bool mxIsChar(const mxArray *pm);
```

## Description

mxIsChar returns logical 1 (true) if pm points to an mxChar array. Otherwise, it returns logical 0 (false).

In C, calling mxIsChar is equivalent to calling:

```
mxGetClassID(pm) == mxCHAR_CLASS
```

## Input Arguments

**pm — MATLAB array**
const mxArray*

Pointer to an mxArray array, specified as const mxArray*.

## Examples

See these examples in *matlabroot*/extern/examples/refbook:

- phonebook.c
- revord.c

See these examples in *matlabroot*/extern/examples/mx:

- mxcreatecharmatrixfromstr.c
- mxmalloc.c

## See Also

mxGetClassID | mxIsClass

**Introduced before R2006a**

# mxIsChar (Fortran)

Determine whether input is `mxChar` array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsChar(pm)
mwPointer pm
```

## Description

Use `mxIsChar` returns `1` if `pm` points to an `mxChar` array. Otherwise, it returns `0`.

In Fortran, calling `mxIsChar` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'char'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matdemo1.F`

## See Also

`mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxIsClass (C)

Determine whether mxArray is object of specified class

## C Syntax

```
#include "matrix.h"
bool mxIsClass(const mxArray *pm, const char *classname);
```

## Returns

Logical `1` (`true`) if `pm` points to an array having category `classname`, and logical `0` (`false`) otherwise.

## Description

Each `mxArray` is tagged as being a certain type. `mxIsClass` returns logical `1` (`true`) if the `mxArray` is of the specified type. Otherwise, the function returns logical `0` (`false`).

MATLAB does not check if the class is derived from a base class.

In C:

```
mxIsClass(pm, "double");
```

is equivalent to calling either of these forms:

```
mxIsDouble(pm);
strcmp(mxGetClassName(pm), "double")==0;
```

It is more efficient to use the `mxIsDouble` form.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

**`classname` — Array category to test**
`const char*`

Array category to test, specified as `const char*`. Use one of these predefined constants. Do not specify `classname` as an integer identifier.

| Value of `classname` | Corresponding Class |
|---|---|
| `cell` | `mxCELL_CLASS` |
| `char` | `mxCHAR_CLASS` |
| `double` | `mxDOUBLE_CLASS` |

| **Value of `classname`** | **Corresponding Class** |
|---|---|
| `function_handle` | `mxFUNCTION_CLASS` |
| `int8` | `mxINT8_CLASS` |
| `int16` | `mxINT16_CLASS` |
| `int32` | `mxINT32_CLASS` |
| `int64` | `mxINT64_CLASS` |
| `logical` | `mxLOGICAL_CLASS` |
| `single` | `mxSINGLE_CLASS` |
| `struct` | `mxSTRUCT_CLASS` |
| `uint8` | `mxUINT8_CLASS` |
| `uint16` | `mxUINT16_CLASS` |
| `uint32` | `mxUINT32_CLASS` |
| `uint64` | `mxUINT64_CLASS` |
| <*class_name*>, which represents the name of a specific MATLAB custom object. You can also specify one of your own class names. | <*class_id*> |
| `unknown` | `mxUNKNOWN_CLASS` |

## Examples

See these examples in *matlabroot*/extern/examples/mx:

- `mxisclass.c`

## See Also

`mxClassID` | `mxGetClassID` | `mxGetClassName` | `mxIsEmpty`

**Introduced before R2006a**

# mxIsClass (Fortran)

Determine whether mxArray is object of specified class

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsClass(pm, classname)
mwPointer pm
character*(*) classname
```

## Description

Each `mxArray` is tagged as being a certain type. `mxIsClass` returns `1` if the `mxArray` is of the specified type. Otherwise, the function returns `0`.

MATLAB does not check if the class is derived from a base class.

In Fortran:

```
mxIsClass(pm, 'double')
```

is equivalent to calling either one of the following:

```
mxIsDouble(pm)
mxGetClassName(pm) .eq. 'double'
```

It is more efficient to use the `mxIsDouble` form.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

**`classname` — Array category to test**
`character*(*)`

Array category to test, specified as `character*(*)`. Use one of these predefined constants. Do not specify `classname` as an integer identifier.

| Value of `classname` | Corresponding Class |
|---|---|
| `cell` | `mxCELL_CLASS` |
| `char` | `mxCHAR_CLASS` |
| `double` | `mxDOUBLE_CLASS` |
| `function_handle` | `mxFUNCTION_CLASS` |
| `int8` | `mxINT8_CLASS` |
| `int16` | `mxINT16_CLASS` |

| Value of classname | Corresponding Class |
|---|---|
| int32 | mxINT32_CLASS |
| int64 | mxINT64_CLASS |
| logical | mxLOGICAL_CLASS |
| single | mxSINGLE_CLASS |
| struct | mxSTRUCT_CLASS |
| uint8 | mxUINT8_CLASS |
| uint16 | mxUINT16_CLASS |
| uint32 | mxUINT32_CLASS |
| uint64 | mxUINT64_CLASS |
| <*class_name*>, which represents the name of a specific MATLAB custom object. You can also specify one of your own class names. | <*class_id*> |
| unknown | mxUNKNOWN_CLASS |

## See Also

mxClassIDFromClassName | mxGetClassID | mxIsEmpty

**Introduced before R2006a**

# mxIsComplex (C)

Determine whether data is complex

## C Syntax

```
#include "matrix.h"
bool mxIsComplex(const mxArray *pm);
```

## Description

Use `mxIsComplex` to determine whether an imaginary part is allocated for an `mxArray`. If an `mxArray` does not have any imaginary data, then the imaginary pointer `pi` is `NULL`. If an `mxArray` is complex, then `pi` points to an array of numbers.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxisfinite.c`
- `mxgetinf.c`

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `convec.c`
- `phonebook.c`

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`
- `yprime.c`
- `mexlock.c`

## See Also
`mxIsNumeric`

**Introduced before R2006a**

# mxIsComplex (Fortran)

Determine whether data is complex

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsComplex(pm)
mwPointer pm
```

## Description

`mxIsComplex` returns `1` if an imaginary part is allocated for an `mxArray`. If an `mxArray` does not have any imaginary data, then the function returns `0`. If an `mxArray` is complex, then `pi` points to an array of numbers.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `convec.F`
- `fulltosparse.F`

## See Also

`mxIsNumeric`

**Introduced before R2006a**

# mxIsDouble (C)

Determine whether mxArray represents data as double-precision, floating-point numbers

## C Syntax

```
#include "matrix.h"
bool mxIsDouble(const mxArray *pm);
```

## Description

`mxIsDouble` returns logical `1` (`true`) if the `mxArray` stores its real and imaginary data as double-precision, floating-point numbers. Otherwise, it returns logical `0` (`false`).

Older versions of MATLAB store all `mxArray` data as double-precision, floating-point numbers. However, starting with MATLAB Version 5 software, MATLAB can store real and imaginary data in other numerical formats.

In C, calling `mxIsDouble` is equivalent to calling:

```
mxGetClassID(pm) == mxDOUBLE_CLASS
```

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `fulltosparse.c`

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxgeteps.c`

## See Also
`mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxIsDouble (Fortran)

Determine whether mxArray represents data as double-precision, floating-point numbers

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsDouble(pm)
mwPointer pm
```

## Description

`mxIsDouble` returns `1` if the `mxArray` stores its real and imaginary data as double-precision, floating-point numbers. Otherwise, it returns `0`.

Older versions of MATLAB store all `mxArray` data as double-precision, floating-point numbers. However, starting with MATLAB Version 5 software, MATLAB can store real and imaginary data in other numerical formats.

In Fortran, calling `mxIsDouble` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'double'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `fulltosparse.F`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxgetepsf.F`

## See Also

`mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxIsEmpty (C)

Determine whether mxArray is empty

## C Syntax

```
#include "matrix.h"
bool mxIsEmpty(const mxArray *pm);
```

## Description

`mxIsEmpty` returns logical `1` (`true`) if the `mxArray` is empty. Otherwise, it returns logical `0` (`false`). An `mxArray` is empty if the size of any of its dimensions is 0.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxisfinite.c`

## See Also

`mxIsClass`

**Introduced before R2006a**

# mxIsEmpty (Fortran)

Determine whether mxArray is empty

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsEmpty(pm)
mwPointer pm
```

## Description

`mxIsEmpty` returns `1` if the `mxArray` is empty. Otherwise, it returns `0`. An `mxArray` is empty if the size of any of its dimensions is 0.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxIsClass`

**Introduced before R2006a**

# mxIsFinite (C and Fortran)

Determine whether input is finite

## C Syntax

```
#include "matrix.h"
bool mxIsFinite(double value);
```

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsFinite(value)
real*8 value
```

## Arguments

`value`

Double-precision, floating-point number to test

## Returns

Logical `1` (`true`) if `value` is finite, and logical `0` (`false`) otherwise.

## Description

Call `mxIsFinite` to determine whether `value` is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxisfinite.c`

## See Also

`mxIsInf`, `mxIsNan`

**Introduced before R2006a**

# mxIsFromGlobalWS (C)

Determine whether mxArray was copied from MATLAB global workspace

## C Syntax

```
#include "matrix.h"
bool mxIsFromGlobalWS(const mxArray *pm);
```

## Description

The function returns logical `1` (`true`) if the array was copied out of the global workspace. Otherwise, it returns logical `0` (`false`). Use `mxIsFromGlobalWS` for standalone MAT-file programs.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matcreat.c`
- `matdgns.c`

## See Also

**Introduced before R2006a**

# mxIsFromGlobalWS (Fortran)

Determine whether mxArray was copied from MATLAB global workspace

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsFromGlobalWS(pm)
mwPointer pm
```

## Description

The function returns `1` if the array was copied out of the global workspace. Otherwise, it returns `0`. Use `mxIsFromGlobalWS` for standalone MAT-file programs.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

**Introduced before R2006a**

# mxIsInf (C and Fortran)

Determine whether input is infinite

## C Syntax

```
#include "matrix.h"
bool mxIsInf(double value);
```

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsInf(value)
real*8 value
```

## Arguments

`value`

Double-precision, floating-point number to test

## Returns

Logical `1` (`true`) if value is infinite, and logical `0` (`false`) otherwise.

## Description

Call `mxIsInf` to determine whether `value` is equal to infinity or minus infinity. MATLAB software stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable `Inf` is built into the system; you cannot modify it.

Operations that return infinity include:

- Division by 0. For example, `5/0` returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine.

If `value` equals `NaN` (Not-a-Number), then `mxIsInf` returns `false`. In other words, `NaN` is not equal to infinity.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxisfinite.c`

## See Also

`mxIsFinite`, `mxIsNaN`

**Introduced before R2006a**

# mxIsInt16 (C)

Determine whether mxArray represents data as signed 16-bit integers

## C Syntax

```
#include "matrix.h"
bool mxIsInt16(const mxArray *pm);
```

## Description

`mxIsInt16` returns logical `1` (`true`) if the `mxArray` stores its real and imaginary data as 16-bit signed integers. Otherwise, it returns logical `0` (`false`).

In C, calling `mxIsInt16` is equivalent to calling:

```
mxGetClassID(pm) == mxINT16_CLASS
```

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## See Also

`mxGetClassID` | `mxIsClass` | `mxIsUint16`

**Introduced before R2006a**

# mxIsInt32 (C)

Determine whether mxArray represents data as signed 32-bit integers

## C Syntax

```
#include "matrix.h"
bool mxIsInt32(const mxArray *pm);
```

## Description

`mxIsInt32` returns logical `1` (`true`) if the `mxArray` stores its data as 32-bit integers. Otherwise, it returns logical `0` (`false`).

In C, calling `mxIsInt32` is equivalent to calling:

```
mxGetClassID(pm) == mxINT32_CLASS
```

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## See Also

`mxGetClassID` | `mxIsClass` | `mxIsUint32`

**Introduced before R2006a**

# mxIsInt64 (C)

Determine whether mxArray represents data as signed 64-bit integers

## C Syntax

```
#include "matrix.h"
bool mxIsInt64(const mxArray *pm);
```

## Description

`mxIsInt64` returns logical `1` (`true`) if the `mxArray` represents its real and imaginary data as 64-bit signed integers. Otherwise, it returns logical `0` (`false`).

In C, calling `mxIsInt64` is equivalent to calling:

```
mxGetClassID(pm) == mxINT64_CLASS
```

## See Also

`mxIsClass`, `mxGetClassID`, `mxIsUint64`

**Introduced before R2006a**

# mxIsInt8 (C)

Determine whether mxArray represents data as signed 8-bit integers

## C Syntax

```
#include "matrix.h"
bool mxIsInt8(const mxArray *pm);
```

## Description

Use `mxIsInt8` to determine whether the specified array represents its real and imaginary data as 8-bit signed integers.

In C, calling `mxIsInt8` is equivalent to calling:

```
mxGetClassID(pm) == mxINT8_CLASS
```

## See Also

`mxGetClassID` | `mxIsClass` | `mxIsUint8`

**Introduced before R2006a**

# mxIsInt16 (Fortran)

Determine whether mxArray represents data as signed 16-bit integers

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsInt16(pm)
mwPointer pm
```

## Description

`mxIsInt16` returns `1` if the specified array represents its real and imaginary data as 16-bit signed integers. Otherwise, it returns `0`.

In C, calling `mxIsInt16` is equivalent to calling:

```
mxGetClassID(pm) == mxINT16_CLASS
```

In Fortran, calling `mxIsInt16` is equivalent to calling:

```
mxGetClassName(pm) == 'int16'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxGetClassID` | `mxIsClass` | `mxIsUint16`

**Introduced before R2006a**

# mxIsInt32 (Fortran)

Determine whether mxArray represents data as signed 32-bit integers

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsInt32(pm)
mwPointer pm
```

## Description

`mxIsInt32` returns `1` if the `mxArray` stores its data as 32-bit integers. Otherwise, it returns `0`.

In Fortran, calling `mxIsInt32` is equivalent to calling:

```
mxGetClassName(pm) == 'int32'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also
`mxGetClassID` | `mxIsClass` | `mxIsUint32`

**Introduced before R2006a**

# mxIsInt64 (Fortran)

Determine whether mxArray represents data as signed 64-bit integers

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsInt64(pm)
mwPointer pm
```

## Description

`mxIsInt64` returns `1` if the `mxArray` stores its data as 64-bit signed integers. Otherwise, it returns `0`.

In Fortran, calling `mxIsInt64` is equivalent to calling:

```
mxGetClassName(pm) == 'int64'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also
`mxGetClassID` | `mxIsClass` | `mxIsUint64`

**Introduced before R2006a**

# mxIsInt8 (Fortran)

Determine whether mxArray represents data as signed 8-bit integers

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsInt8(pm)
mwPointer pm
```

## Description

`mxIsInt8` returns `1` if the `mxArray` stores its data as 8-bit signed integers. Otherwise, it returns `0`.

In Fortran, calling `mxIsInt8` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'int8'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also
`mxGetClassID` | `mxIsClass` | `mxIsUint8`

**Introduced before R2006a**

# mxIsLogical (C)

Determine whether mxArray is of type `mxLogical`

## C Syntax

```
#include "matrix.h"
bool mxIsLogical(const mxArray *pm);
```

## Description

`mxIsLogical` returns logical `1` (`true`) if the data in the `mxArray` is Boolean (logical). Otherwise, it returns logical `0` (`false`). If an `mxArray` is logical, then MATLAB treats all zeros as meaning `false` and all nonzero values as meaning `true`.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxislogical.c`

## See Also
`mxIsClass`

**Topics**
"Logical Operations"

**Introduced before R2006a**

# mxIsLogical (Fortran)

Determine whether mxArray is of type `mxLogical`

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsLogical(pm)
mwPointer pm
```

## Description

`mxIsLogical` returns `1` if the `mxArray` logical. Otherwise, it returns `0`. If an `mxArray` is logical, then MATLAB treats all zeros as meaning `false` and all nonzero values as meaning `true`.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxIsClass`

**Topics**
"Logical Operations"

**Introduced before R2006a**

# mxIsLogicalScalar (C)

Determine whether scalar array is of type mxLogical

## C Syntax

```
#include "matrix.h"
bool mxIsLogicalScalar(const mxArray *array_ptr);
```

## Arguments

`array_ptr`

Pointer to an `mxArray`

## Returns

Logical `1` (`true`) if the `mxArray` is of class `mxLogical` and has `1`-by-`1` dimensions. Otherwise, it returns logical `0` (`false`).

## Description

Use `mxIsLogicalScalar` to determine whether MATLAB treats the scalar data in the `mxArray` as logical or numerical.

## See Also

`mxGetLogicals` | `mxGetScalar` | `mxIsLogical` | `mxIsLogicalScalarTrue`

**Topics**

"Logical Operations"

**Introduced before R2006a**

# mxIsLogicalScalarTrue (C)

Determine whether scalar array of type mxLogical is true

## C Syntax

```
#include "matrix.h"
bool mxIsLogicalScalarTrue(const mxArray *array_ptr);
```

## Arguments

`array_ptr`

Pointer to an `mxArray`

## Returns

Logical `1` (`true`) if the value of the `mxArray` logical, scalar element is `true`. Otherwise, it returns logical `0` (`false`).

## Description

Use `mxIsLogicalScalarTrue` to determine whether the value of a scalar `mxArray` is true or false.

## See Also

`mxGetLogicals` | `mxGetScalar` | `mxIsLogical` | `mxIsLogicalScalar`

**Topics**

"Logical Operations"

**Introduced before R2006a**

# mxIsNaN (C and Fortran)

Determine whether input is NaN (Not-a-Number)

## C Syntax

```
#include "matrix.h"
bool mxIsNaN(double value);
```

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsNaN(value)
real*8 value
```

## Arguments

`value`

Double-precision, floating-point number to test

## Returns

Logical `1` (`true`) if `value` is `NaN` (Not-a-Number), and logical `0` (`false`) otherwise.

## Description

Call `mxIsNaN` to determine whether `value` is `NaN`. `NaN` is the IEEE arithmetic representation for Not-a-Number. A `NaN` is obtained as a result of mathematically undefined operations such as

- `0.0/0.0`
- `Inf-Inf`

The system understands a family of bit patterns as representing `NaN`. `NaN` is not a single value; it is a family of numbers that MATLAB (and other IEEE-compliant applications) uses to represent an error condition or missing data.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxisfinite.c`

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `findnz.c`
- `fulltosparse.c`

## See Also

`mxIsFinite`, `mxIsInf`

**Introduced before R2006a**

# mxIsNumeric (C)

Determine whether mxArray is numeric

## C Syntax

```
#include "matrix.h"
bool mxIsNumeric(const mxArray *pm);
```

## Description

Call `mxIsNumeric` to determine whether the specified array contains numeric data. If the array has a storage type that represents numeric data, then `mxIsNumeric` returns logical `1` (`true`). Call `mxGetClassID` to determine the storage type. These class IDs represent storage types for arrays that can contain numeric data:

- `mxDOUBLE_CLASS`
- `mxSINGLE_CLASS`
- `mxINT8_CLASS`
- `mxUINT8_CLASS`
- `mxINT16_CLASS`
- `mxUINT16_CLASS`
- `mxINT32_CLASS`
- `mxUINT32_CLASS`
- `mxINT64_CLASS`
- `mxUINT64_CLASS`

Otherwise, `mxIsNumeric` returns logical `0` (`false`).

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `phonebook.c`

## See Also

`mxGetClassID`

**Introduced before R2006a**

# mxIsNumeric (Fortran)

Determine whether mxArray is numeric

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsNumeric(pm)
mwPointer pm
```

## Description

Call `mxIsNumeric` to determine whether the specified array contains numeric data. If the array has a storage type that represents numeric data, then `mxIsNumeric` returns `1`. Call `mxGetClassID` to determine the storage type. These class IDs represent storage types for arrays that can contain numeric data:

- `mxDOUBLE_CLASS`
- `mxSINGLE_CLASS`
- `mxINT8_CLASS`
- `mxUINT8_CLASS`
- `mxINT16_CLASS`
- `mxUINT16_CLASS`
- `mxINT32_CLASS`
- `mxUINT32_CLASS`
- `mxINT64_CLASS`
- `mxUINT64_CLASS`

Otherwise, `mxIsNumeric` returns `0`.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/eng_mat`:

- `matdemo1.F`

## See Also

`mxGetClassID`

**Introduced before R2006a**

# mxIsScalar (C)

Determine whether array is scalar array

## C Syntax

```
#include "matrix.h"
bool mxIsScalar(const mxArray *array_ptr);
```

## Arguments

`array_ptr`

Pointer to an `mxArray`

## Returns

Logical `1` (`true`) if the `mxArray` has `1`-by-`1` dimensions. Otherwise, it returns logical `0` (`false`).

**Note** Only use `mxIsScalar` for `mxArray` classes with IDs documented by `mxClassID`.

## Example

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxisscalar.c`

## See Also

`mxClassID` | `mxGetScalar`

**Introduced in R2015a**

# mxIsSingle (C)

Determine whether mxArray represents data as single-precision, floating-point numbers

## C Syntax

```
#include "matrix.h"
bool mxIsSingle(const mxArray *pm);
```

## Description

`mxIsSingle` returns logical `1` (`true`) if the `mxArray` stores its real and imaginary data as single-precision, floating-point numbers. Otherwise, it returns logical `0` (`false`).

In C, calling `mxIsSingle` is equivalent to calling:

```
mxGetClassID(pm) == mxSINGLE_CLASS
```

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## See Also
`mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxIsSingle (Fortran)

Determine whether mxArray represents data as single-precision, floating-point numbers

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsSingle(pm)
mwPointer pm
```

## Description

`mxIsSingle` returns `1` if the `mxArray` stores its real and imaginary data as single-precision, floating-point numbers. Otherwise, it returns `0`.

In Fortran, calling `mxIsSingle` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'single'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxIsSparse (C)

Determine whether input is sparse mxArray

## C Syntax

```
#include "matrix.h"
bool mxIsSparse(const mxArray *pm);
```

## Description

`mxIsSparse` returns logical `1` (`true`) if `pm` points to a sparse `mxArray`. Otherwise, it returns logical `0` (`false`). Many routines (for example, `mxGetIr` and `mxGetJc`) require a sparse `mxArray` as input.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `phonebook.c`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxgetnzmax.c`
- `mxsetdimensions.c`
- `mxsetnzmax.c`

## See Also
`mxCreateSparse` | `mxGetClassID` | `mxIsClass` | `sparse`

**Introduced before R2006a**

# mxIsSparse (Fortran)

Determine whether input is sparse mxArray

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsSparse(pm)
mwPointer pm
```

## Description

`mxIsSparse` returns `1` if `pm` points to a sparse `mxArray`. Otherwise, it returns `0`. Many routines (for example, `mxGetIr` and `mxGetJc`) require a sparse `mxArray` as input.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxsetdimensionsf.F`

## See Also

`mxCreateSparse` | `mxGetClassID` | `mxIsClass` | `sparse`

**Introduced before R2006a**

# mxIsStruct (C)

Determine whether mxArray is structure

## C Syntax

```
#include "matrix.h"
bool mxIsStruct(const mxArray *pm);
```

## Description

`mxIsStruct` returns logical `1` (`true`) if `pm` points to a structure `mxArray`. Otherwise, it returns logical `0` (`false`). Many routines (for example, `mxGetFieldNameByNumber` and `mxSetField`) require a structure `mxArray` as an argument.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `phonebook.c`

## See Also
`mxCreateStructArray` | `mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxIsStruct (Fortran)

Determine whether mxArray is structure

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsStruct(pm)
mwPointer pm
```

## Description

`mxIsStruct` returns `1` if `pm` points to a structure `mxArray`. Otherwise, it returns `0`. Many routines (for example, `mxGetFieldNameByNumber` and `mxSetField`) require a structure `mxArray` as an argument.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxCreateStructArray` | `mxGetClassID` | `mxIsClass`

**Introduced before R2006a**

# mxIsUint16 (C)

Determine whether mxArray represents data as unsigned 16-bit integers

## C Syntax

```
#include "matrix.h"
bool mxIsUint16(const mxArray *pm);
```

## Description

`mxIsUint16` returns logical `1` (`true`) if the `mxArray` stores its data as 64-bit unsigned integers. Otherwise, it returns logical `0` (`false`).

In C, calling `mxIsUint16` is equivalent to calling:

```
mxGetClassID(pm) == mxUINT16_CLASS
```

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## See Also
`mxGetClassID` | `mxIsClass` | `mxIsInt16`

**Introduced before R2006a**

# mxIsUint32 (C)

Determine whether mxArray represents data as unsigned 32-bit integers

## C Syntax

```
#include "matrix.h"
bool mxIsUint32(const mxArray *pm);
```

## Description

`mxIsUint32` returns logical `1` (`true`) if the `mxArray` stores its data as 32-bit unsigned integers. Otherwise, it returns logical `0` (`false`).

In C, calling `mxIsUint32` is equivalent to calling:

```
mxGetClassID(pm) == mxUINT32_CLASS
```

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## See Also

`mxGetClassID` | `mxIsClass` | `mxIsInt32`

**Introduced before R2006a**

# mxIsUint64 (C)

Determine whether mxArray represents data as unsigned 64-bit integers

## C Syntax

```
#include "matrix.h"
bool mxIsUint64(const mxArray *pm);
```

## Description

`mxIsUint64` returns logical `1` (`true`) if the `mxArray` stores its data as 64-bit unsigned integers. Otherwise, it returns logical `0` (`false`).

In C, calling `mxIsUint64` is equivalent to calling:

```
mxGetClassID(pm) == mxUINT64_CLASS
```

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## See Also

`mxGetClassID` | `mxIsClass` | `mxIsInt64`

**Introduced before R2006a**

# mxIsUint8 (C)

Determine whether mxArray represents data as unsigned 8-bit integers

## C Syntax

```
#include "matrix.h"
bool mxIsUint8(const mxArray *pm);
```

## Description

`mxIsUint8` returns logical `1` (`true`) if the `mxArray` stores its data as unsigned 8-bit integers. Otherwise, it returns logical `0` (`false`).

In C, calling `mxIsUint8` is equivalent to calling:

```
mxGetClassID(pm) == mxUINT8_CLASS
```

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

## See Also

`mxGetClassID` | `mxIsClass` | `mxIsInt8`

**Introduced before R2006a**

# mxIsUint16 (Fortran)

Determine whether mxArray represents data as unsigned 16-bit integers

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsUint16(pm)
mwPointer pm
```

## Returns

Logical `1` (`true`) if the `mxArray` stores its data as unsigned 16-bit integers, and logical `0` (`false`) otherwise.

## Description

`mxIsUint16` returns `1` if the `mxArray` stores its data as 64-bit unsigned integers. Otherwise, it returns `0`.

In Fortran, calling `mxIsUint16` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'uint16'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also
`mxGetClassID` | `mxIsClass` | `mxIsInt16`

**Introduced before R2006a**

# mxIsUint32 (Fortran)

Determine whether mxArray represents data as unsigned 32-bit integers

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsUint32(pm)
mwPointer pm
```

## Description

`mxIsUint32` returns `1` if the `mxArray` stores its data as 32-bit unsigned integers. Otherwise, it returns `0`.

In Fortran, calling `mxIsUint32` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'uint32'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxGetClassID` | `mxIsClass` | `mxIsInt32`

**Introduced before R2006a**

# mxIsUint64 (Fortran)

Determine whether mxArray represents data as unsigned 64-bit integers

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsUint64(pm)
mwPointer pm
```

## Description

`mxIsUint64` returns `1` if the `mxArray` stores its data as 64-bit unsigned integers. Otherwise, it returns `0`.

In Fortran, calling `mxIsUint64` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'uint64'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxGetClassID` | `mxIsClass` | `mxIsInt64`

**Introduced before R2006a**

# mxIsUint8 (Fortran)

Determine whether mxArray represents data as unsigned 8-bit integers

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxIsUint8(pm)
mwPointer pm
```

## Description

`mxIsUint8` returns `1` if the `mxArray` stores its data as 8-bit unsigned integers. Otherwise, it returns `0`.

In Fortran, calling `mxIsUint8` is equivalent to calling:

```
mxGetClassName(pm) .eq. 'uint8'
```

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

## See Also

`mxGetClassID` | `mxIsClass` | `mxIsInt8`

**Introduced before R2006a**

# mxLogical (C)

Type for logical array

## Description

All logical mxArrays store their data elements as mxLogical rather than as bool.

The header file containing this type is:

```
#include "matrix.h"
```

## Examples

See these examples in *matlabroot*/extern/examples/mx:

- mxislogical.c

## See Also

mxCreateLogicalArray

## Tips

- For information about data in MATLAB language scripts and functions, see “Data Types”.

**Introduced before R2006a**

# mxMakeArrayComplex (C)

Convert real mxArray to complex, preserving real data

## C Syntax

```
#include "matrix.h"
int mxMakeArrayComplex(mxArray *pa);
```

## Description

Use mxMakeArrayComplex to convert a real mxArray to a complex mxArray. The real part of the updated array contains the real data from the original array.

If pa is empty, then the function returns a complex empty mxArray.

If pa is complex, then the function does nothing.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to a numeric mxArray array.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

Returns 0 if unsuccessful. The function is unsuccessful if pa is NULL, nonnumeric, or read-only.

## Examples

Suppose that your application processes complex data and you create complex mxArrays to handle the data. If you pass a complex array containing only real data to a MATLAB function, then the returned value is a real array. For example, call the MATLAB sqrt function with the following input.

```
a = complex([2,4])
```

```
a =

   2.0000 + 0.0000i   4.0000 + 0.0000i
```

Although the input argument is complex, the data is real-only, and the output of the function is no longer complex.

```
a1 = sqrt(a)
```

```
a1 =

    1.4142    2.0000
```

To maintain the complexity of the data, use the `mxMakeArrayComplex` function to wrap the result. To build the MEX file `complexFnc.c`:

```
mex -R2018a complexFnc.c
```

```
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
    mxArray *rhs[1], *lhs[1];

    /* check for the proper number of arguments */
    if(nrhs != 1) {
        mexErrMsgIdAndTxt("MATLAB:complexFnc:checkrhs","1 input required.");
    }

    if(nlhs > 1) {
        mexErrMsgIdAndTxt("MATLAB:complexFnc:checklhs","Too many output arguments.");
    }

#if MX_HAS_INTERLEAVED_COMPLEX

    /* get the square root */
    rhs[0] = mxDuplicateArray(prhs[0]);
    mexCallMATLAB(1, lhs, 1, rhs, "sqrt");
    if(!mxIsComplex(lhs[0])) {
        /* preserve complexity of data */
        mxMakeArrayComplex(lhs[0]);
    }

    plhs[0] = mxDuplicateArray(lhs[0]);
#endif
}
```

## See Also

`mxMakeArrayReal`

**Introduced in R2018a**

# mxMakeArrayComplex (Fortran)

Convert real mxArray to complex, preserving real data

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxMakeArrayComplex(pa)
mwPointer pa
```

## Description

Use mxMakeArrayComplex to convert a real mxArray to a complex mxArray. The real part of the updated array contains the real data from the original array.

If pa is empty, then the function returns a complex empty mxArray.

If pa is complex, then the function does nothing.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to a numeric mxArray array.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

Returns 0 if unsuccessful. The function is unsuccessful if pa is NULL, nonnumeric, or read-only.

## See Also

mxMakeArrayReal

**Introduced in R2018b**

# mxMakeArrayReal (C)

Convert complex `mxArray` to real, preserving real data

## C Syntax

```
#include "matrix.h"
int mxMakeArrayReal(mxArray *pa);
```

## Description

Use `mxMakeArrayReal` to convert a complex `mxArray` to a real `mxArray`. The array contains the data from the real part of the original array. If the original `mxArray` is real, then the function does nothing.

## Input Arguments

**pa — MATLAB array**
`mxArray *`

Pointer to a numeric `mxArray` array.

## Output Arguments

**`status` — Function status**
`int`

Function status, returned as `int`. If successful, then the function returns `1`.

Returns `0` if unsuccessful. The function is unsuccessful if `pa` is `NULL`, nonnumeric, or read-only.

## Examples

Suppose that your application determines that real numbers are the only meaningful result. If complex results occur because of noise in the data, then the program drops small imaginary parts. However, if the imaginary part exceeds a threshold, then the program throws an error.

In the following example `dropComplexIfUnderThreshold.c`, the threshold limit is set to `.2`.

```
#include "mex.h"

/* dropComplexIfUnderThreshold converts input to a real double scalar
 * with eihter no imaginary data or imaginary data less than
 * the value of LIMIT.
 *
 * Use this function for data with imaginary values less than some LIMIT
 * that can be dropped, and then revert the results to a real array.
 *
 * Usage: B = dropComplexIfUnderThreshold(A);
 * Where:
 *    A is a mxDOUBLE_CLASS scalar complex or real.
 *    B is a real scalar which is a copy of the real value of A.
 *
 * Errors if:
 *    nlhs != 1
 *    nrhs != 1
```

```
 *    prhs[0] is not a mxDOUBLE_CLASS scalar
 *    imaginary data value is equal or greater than LIMIT
 *
 * Build:
 *   mex -R2018a dropComplexIfUnderThreshold.c     - interleaved complex API
 *   mex [-R2017b] dropComplexIfUnderThreshold.c   - separate complex API
 *
 * Run:
 * >> dropComplexIfUnderThreshold(3)
 * ans = 3
 *
 * >> dropComplexIfUnderThreshold(complex(3,.1))
 * ans = 3
 *
 * >> dropComplexIfUnderThreshold(complex(1,.2))
 * Error using dropComplexIfUnderThreshold
 * Data error.
 * >>
 */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
#define LIMIT .2

    /* check for the proper number of arguments */
    if(nrhs != 1) {
        mexErrMsgIdAndTxt("MATLAB:dropComplexIfUnderThreshold:checkrhs","1 input required.");
    }

    if(nlhs > 1) {
        mexErrMsgIdAndTxt("MATLAB:dropComplexIfUnderThreshold:checklhs","Too many output arguments.");
    }

    if( !(mxIsDouble(prhs[0]) && mxIsScalar(prhs[0])) ) {
        mexErrMsgIdAndTxt("MATLAB:dropComplexIfUnderThreshold:checkdouble","rhs[0] must be double scalar.");
    }

    plhs[0] = mxDuplicateArray(prhs[0]);

    if(mxIsComplex(prhs[0])) {
#if MX_HAS_INTERLEAVED_COMPLEX
        mxComplexDouble *dt = mxGetComplexDoubles(prhs[0]);

        /* test imaginary data for significance */
        if( dt[0].imag < LIMIT) {
            mxMakeArrayReal(plhs[0]);
        }
        else {
            mexErrMsgIdAndTxt("MATLAB:dropComplexIfUnderThreshold:outOfBounds","Data error.");
        }
#else
        mxDouble *dt = mxGetPi(plhs[0]);

        /* test imaginary data for significance */
        if (dt[0] < LIMIT) {
            mxFree(mxGetPi(plhs[0]));
            mxSetPi(plhs[0], 0);
        } else {
            mexErrMsgIdAndTxt("MATLAB:dropComplexIfUnderThreshold:outOfBounds","Data error.");
        }
#endif
    }
}
```

To build the MEX file, type:

```
mex -R2018a dropComplexIfUnderThreshold.c
```

To test the function, type:

```
dropComplexIfUnderThreshold(3)
```

```
ans = 3
```

```
dropComplexIfUnderThreshold(complex(3,.1))
```

```
ans = 3
```

```
dropComplexIfUnderThreshold(complex(1,.2))
```

```
Error using dropComplexIfUnderThreshold
Data error.
```

## See Also

mxMakeArrayComplex

**Introduced in R2018a**

# mxMakeArrayReal (Fortran)

Convert complex `mxArray` to real, preserving real data

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxMakeArrayReal(pa)
mwPointer pa
```

## Description

Use `mxMakeArrayReal` to convert a complex `mxArray` to a real `mxArray`. The array contains the data from the real part of the original array. If the original `mxArray` is real, then the function does nothing.

## Input Arguments

**pa — MATLAB array**
`mwPointer`

Pointer to a numeric `mxArray` array.

## Output Arguments

**`status` — Function status**
`integer*4`

Function status, returned as `integer*4`. If successful, then the function returns `1`.

Returns `0` if unsuccessful. The function is unsuccessful if `pa` is `NULL`, nonnumeric, or read-only.

## See Also
`mxMakeArrayComplex`

**Introduced in R2018b**

# mxMalloc (C and Fortran)

Allocate uninitialized dynamic memory using MATLAB memory manager

## C Syntax

```
#include "matrix.h"
#include <stdlib.h>
void *mxMalloc(mwSize n);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxMalloc(n)
mwSize n
```

## Arguments

`n`

Number of bytes to allocate for `n` greater than `0`

## Returns

Pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a MAT or engine standalone application, then `mxMalloc` returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX file, then the MEX file terminates and control returns to the MATLAB prompt.

`mxMalloc` is unsuccessful when there is insufficient free heap space.

If you call `mxMalloc` in C with value `n = 0`, then MATLAB returns either `NULL` or a valid pointer.

## Description

`mxMalloc` allocates contiguous heap space sufficient to hold `n` bytes. To allocate memory in MATLAB applications, use `mxMalloc` instead of the ANSI C `malloc` function.

In MEX files, but not MAT or engine applications, `mxMalloc` registers the allocated memory with the MATLAB memory manager. When control returns to the MATLAB prompt, the memory manager then automatically frees, or deallocates, this memory.

How you manage the memory created by this function depends on the purpose of the data assigned to it. If you assign it to an output argument in `plhs[]` using a function such as `mxSetDoubles`, then MATLAB is responsible for freeing the memory.

If you use the data internally, then the MATLAB memory manager maintains a list of all memory allocated by the function and automatically frees (deallocates) the memory when control returns to the MATLAB prompt. In general, we recommend that MEX file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX file than to rely on the automatic mechanism. Therefore, when you finish using the memory allocated by this function, call `mxFree` to deallocate the memory.

If you do not assign this data to an output argument, and you want it to persist after the MEX file completes, then call `mexMakeMemoryPersistent` after calling this function. If you write a MEX file with persistent memory, then be sure to register a `mexAtExit` function to free allocated memory in the event your MEX file is cleared.

## Examples

See these examples in *matlabroot*/extern/examples/mx:

- `mxmalloc.c`
- `mxsetdimensions.c`

See these examples in *matlabroot*/extern/examples/refbook:

- `arrayFillSetPr.c`

## See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxCalloc`, `mxDestroyArray`, `mxFree`, `mxRealloc`

**Introduced before R2006a**

# mxRealloc (C and Fortran)

Reallocate dynamic memory using MATLAB memory manager

## C Syntax

```
#include "matrix.h"
#include <stdlib.h>
void *mxRealloc(void *ptr, mwSize size);
```

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxRealloc(ptr, size)
mwPointer ptr
mwSize size
```

## Arguments

`ptr`

Pointer to a block of memory allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`.

`size`

New size of allocated memory, in bytes.

## Returns

Pointer to the start of the reallocated block of memory, if successful. If unsuccessful in a MAT or engine standalone application, then `mxRealloc` returns `NULL` in C (`0` in Fortran) and leaves the original memory block unchanged. (Use `mxFree` to free the original memory block). If unsuccessful in a MEX file, then the MEX file terminates and control returns to the MATLAB prompt.

`mxRealloc` is unsuccessful when there is insufficient free heap space.

## Description

`mxRealloc` changes the size of a memory block that has been allocated with `mxCalloc`, `mxMalloc`, or `mxRealloc`. To allocate memory in MATLAB applications, use `mxRealloc` instead of the ANSI C `realloc` function.

`mxRealloc` changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents of the reallocated memory are unchanged up to the smaller of the new and old sizes. The reallocated memory might be in a different location from the original memory, so the returned pointer can be different from `ptr`. If the memory location changes, then `mxRealloc` frees the original memory block pointed to by `ptr`.

If `size` is greater than `0` and `ptr` is `NULL` in C (`0` in Fortran), then `mxRealloc` behaves like `mxMalloc`. `mxRealloc` allocates a new block of memory of `size` bytes and returns a pointer to the new block.

If `size` is `0` and `ptr` is not `NULL` in C (`0` in Fortran), then `mxRealloc` frees the memory pointed to by `ptr` and returns `NULL` in C (`0` in Fortran).

In MEX files, but not MAT or engine applications, `mxRealloc` registers the allocated memory with the MATLAB memory manager. When control returns to the MATLAB prompt, the memory manager then automatically frees, or deallocates, this memory.

How you manage the memory created by this function depends on the purpose of the data assigned to it. If you assign it to an output argument in `plhs[]` using a function such as `mxSetDoubles`, then MATLAB is responsible for freeing the memory.

If you use the data internally, then the MATLAB memory manager maintains a list of all memory allocated by the function and automatically frees (deallocates) the memory when control returns to the MATLAB prompt. In general, we recommend that MEX file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX file than to rely on the automatic mechanism. Therefore, when you finish using the memory allocated by this function, call `mxFree` to deallocate the memory.

If you do not assign this data to an output argument, and you want it to persist after the MEX file completes, then call `mexMakeMemoryPersistent` after calling this function. If you write a MEX file with persistent memory, then be sure to register a `mexAtExit` function to free allocated memory in the event your MEX file is cleared.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxsetnzmax.c`

## See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxCalloc`, `mxDestroyArray`, `mxFree`, `mxMalloc`

**Introduced before R2006a**

# mxRemoveField (C and Fortran)

Remove field from structure array

## C Syntax

```
#include "matrix.h"
void mxRemoveField(mxArray *pm, int fieldnumber);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxRemoveField(pm, fieldnumber)
mwPointer pm
integer*4 fieldnumber
```

## Arguments

`pm`

Pointer to a structure `mxArray`

`fieldnumber`

Number of the field you want to remove. In C, to remove the first field, set `fieldnumber` to `0`; to remove the second field, set `fieldnumber` to `1`; and so on. In Fortran, to remove the first field, set `fieldnumber` to `1`; to remove the second field, set `fieldnumber` to `2`; and so on.

## Description

Call `mxRemoveField` to remove a field from a structure array. If the field does not exist, then nothing happens. This function does not destroy the field values. To destroy the actual field values, call `mxRemoveField` and then call `mxDestroyArray`.

Consider a MATLAB structure initialized to:

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

In C, the field number `0` represents the field `name`; field number `1` represents field `billing`; field number `2` represents field `test`. In Fortran, the field number `1` represents the field `name`; field number `2` represents field `billing`; field number `3` represents field `test`.

## See Also

`mxAddField`, `mxDestroyArray`, `mxGetFieldByNumber`

**Introduced before R2006a**

# mxSetCell (C and Fortran)

Set contents of cell array

## C Syntax

```
#include "matrix.h"
void mxSetCell(mxArray *pm, mwIndex index, mxArray *value);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetCell(pm, index, value)
mwPointer pm, value
mwIndex index
```

## Arguments

`pm`

Pointer to a cell `mxArray`

`index`

Index from the beginning of the `mxArray`. Specify the number of elements between the first cell of the `mxArray` and the cell you want to set. The easiest way to calculate `index` in a multidimensional cell array is to call `mxCalcSingleSubscript`.

`value`

Pointer to new value for the cell. You can put an `mxArray` of any type into a cell. You can even put another cell `mxArray` into a cell.

## Description

Call `mxSetCell` to put the designated `value` into a particular cell of a cell `mxArray`.

**Note** Inputs to a MEX-file are constant read-only `mxArrays`. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxDestroyArray` on the pointer returned by `mxGetCell` before you call `mxSetCell`.

## Examples

See these examples in *matlabroot*`/extern/examples/refbook`:

- `phonebook.c`

See these examples in *matlabroot*`/extern/examples/mx`:

- `mxcreatecellmatrix.c`
- `mxcreatecellmatrixf.F`

## See Also

`mxCreateCellArray`, `mxCreateCellMatrix`, `mxGetCell`, `mxIsCell`, `mxDestroyArray`

**Introduced before R2006a**

# mxSetClassName (C)

Structure array to MATLAB object array

**Note** Use mxSetClassName for classes defined without a classdef statement.

## C Syntax

```
#include "matrix.h"
int mxSetClassName(mxArray *array_ptr, const char *classname);
```

## Arguments

array_ptr

Pointer to an mxArray of class mxSTRUCT_CLASS

classname

Object class to which to convert array_ptr

## Returns

0 if successful, and nonzero otherwise. One cause of failure is that array_ptr is not a structure mxArray. Call mxIsStruct to determine whether array_ptr is a structure.

## Description

mxSetClassName converts a structure array to an object array, to be saved later to a MAT-file. MATLAB does not register or validate the object until it is loaded by the LOAD command. If the specified classname is an undefined class within MATLAB, then LOAD converts the object back to a simple structure array.

## See Also

mxIsClass, mxGetClassID, mxIsStruct

**Introduced before R2006a**

# mxSetData (C)

Set pointer to data elements in nonnumeric `mxArray`

**Note** `mxSetData` is not recommended for numeric arrays. Use typed, data-access functions instead. For more information, see “Compatibility Considerations”.

## C Syntax

```
#include "matrix.h"
void mxSetData(mxArray *pm, void *pa);
```

## Description

Use `mxSetData` to set data elements for nonnumeric arrays only.

For numeric arrays, MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- “Typed Data Access in C MEX Files”
- “MATLAB Support for Interleaved Complex API in MEX Functions”
- “Upgrade MEX Files to Use Interleaved Complex API”
- Example `arrayFillSetData.c`

To build the MEX file, call `mex` with the `-R2018a` option.

The `mxSetData` function does not free memory allocated for existing data. To free existing memory, call `mxFree` on the pointer returned by `mxGetData`.

## Input Arguments

**pm — Pointer to nonnumeric MATLAB array**
`mxArray *`

Pointer to a nonnumeric MATLAB array, specified as `mxArray *`.

**pa — Pointer to data array**
`void *`

Pointer to the data array within an `mxArray`, specified as `void *`

The array must be in dynamic memory. Call `mxCalloc` to allocate this memory. Do not use the ANSI C `calloc` function, which can cause memory alignment issues leading to program termination.

## Compatibility Considerations

**Results of `mxSetData` are different based on build option**
*Behavior changed in R2018a*

For a complex numeric `mxArray`, the `mxSetData` function sets different values based on the `mex` build option.

If you build the MEX file with the default release-specific option (`-R2017b`), then the function sets the elements of the array to the real-only values.

If you build the MEX file with the `-R2018a` option, then:

- When `pm` is a real array, `pa` becomes the real component of `pm`.
- When `pm` is complex array, `pa` also must be complex. Otherwise, the elements of `pa` become interleaved real and imaginary values, not real-only values.

## See Also

**Topics**
`arrayFillSetData.c`
“Typed Data Access in C MEX Files”
“MATLAB Support for Interleaved Complex API in MEX Functions”

**Introduced before R2006a**

# mxSetData (Fortran)

Set pointer to data elements in nonnumeric mxArray

---

**Note** mxSetData is not recommended for numeric arrays. Use typed, data-access functions instead. For more information, see "Compatibility Considerations".

---

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetData(pm, pr)
mwPointer pm, pr
```

## Description

Use mxSetData to set data elements for nonnumeric arrays only.

For numeric arrays, MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- "Typed Data Access in C MEX Files"
- "MATLAB Support for Interleaved Complex API in MEX Functions"
- "Upgrade MEX Files to Use Interleaved Complex API"

To build the MEX file, call mex with the -R2018a option.

The mxSetData function does not free memory allocated for existing data. To free existing memory, call mxFree on the pointer returned by mxGetData.

## Input Arguments

**pm — Pointer to nonnumeric MATLAB array**
mwPointer

Pointer to a nonnumeric MATLAB array, specified as mwPointer.

**pa — Pointer to data array**
mwPointer

Pointer to the data array within an mxArray, specified as mwPointer.

The array must be in dynamic memory. Call mxCalloc to allocate this memory.

## Compatibility Considerations

**Results of mxSetData are different based on build option**
*Behavior changed in R2018b*

For a complex numeric mxArray, the mxSetData function sets different values based on the mex build option.

If you build the MEX file with the default release-specific option (`-R2017b`), then the function sets the elements of the array to the real-only values.

If you build the MEX file with the `-R2018a` option, then:

- When `pm` is a real array, `pa` becomes the real component of `pm`.
- When `pm` is complex array, `pa` also must be complex. Otherwise, the elements of `pa` become interleaved real and imaginary values, not real-only values.

## See Also

**Topics**
“Typed Data Access in C MEX Files”
“MATLAB Support for Interleaved Complex API in MEX Functions”

**Introduced before R2006a**

# mxSetDimensions (C)

Modify number of dimensions and size of each dimension

## C Syntax

```
#include "matrix.h"
int mxSetDimensions(mxArray *pm, const mwSize *dims, mwSize ndim);
```

## Description

`mxSetDimensions` returns `0` on success, and `1` on failure. `mxSetDimensions` allocates heap space to hold the input size array. So it is possible (though unlikely) that increasing the number of dimensions can cause the system to run out of heap space.

Call `mxSetDimensions` to reshape an existing `mxArray`. `mxSetDimensions` is like `mxSetM` and `mxSetN`; however, `mxSetDimensions` provides greater control for reshaping an `mxArray` that has more than two dimensions.

`mxSetDimensions` does not allocate or deallocate any space for the `pr` or `pi` arrays. So, if your call to `mxSetDimensions` increases the number of elements in the `mxArray`, then enlarge the `pr` (and `pi`, if it exists) arrays accordingly.

If your call to `mxSetDimensions` reduces the number of elements in the `mxArray`, then you can optionally reduce the size of the `pr` and `pi` arrays using `mxRealloc`.

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, then the resulting array has the dimensions 4-by-1-by-7.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

**`dims` — Dimensions array**
`mwSize`

Dimensions array. Each element in the dimensions array contains the size of the array in that dimension, specified as `mwSize`. For example, in Fortran, setting `dims(1)` to `5` and `dims(2)` to `7` establishes a `5`-by-`7` `mxArray`. In most cases, there are `ndim` elements in the `dims` array.

**`ndim` — Number of dimensions**
`mwSize`

Number of dimensions, specified as `mwSize`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxsetdimensions.c`

## See Also

`mxGetNumberOfDimensions` | `mxRealloc` | `mxSetM` | `mxSetN`

**Introduced before R2006a**

# mxSetDimensions (Fortran)

Modify number of dimensions and size of each dimension

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetDimensions(pm, dims, ndim)
mwPointer pm
mwSize ndim
mwSize dims(ndim)
```

## Description

`mxSetDimensions` returns `0` on success, and `1` on failure. `mxSetDimensions` allocates heap space to hold the input size array. So it is possible (though unlikely) that increasing the number of dimensions can cause the system to run out of heap space.

Call `mxSetDimensions` to reshape an existing `mxArray`. `mxSetDimensions` is like `mxSetM` and `mxSetN`; however, `mxSetDimensions` provides greater control for reshaping an `mxArray` that has more than two dimensions.

`mxSetDimensions` does not allocate or deallocate any space for the `pr` or `pi` arrays. So, if your call to `mxSetDimensions` increases the number of elements in the `mxArray`, then enlarge the `pr` (and `pi`, if it exists) arrays accordingly.

If your call to `mxSetDimensions` reduces the number of elements in the `mxArray`, then you can optionally reduce the size of the `pr` and `pi` arrays using `mxRealloc`.

MATLAB automatically removes any trailing singleton dimensions specified in the `dims` argument. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, then the resulting array has the dimensions 4-by-1-by-7.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

**`dims` — Dimensions array**
`mwSize`

Dimensions array. Each element in the dimensions array contains the size of the array in that dimension, specified as `mwSize`. For example, in Fortran, setting `dims(1)` to `5` and `dims(2)` to `7` establishes a `5`-by-`7` `mxArray`. In most cases, there are `ndim` elements in the `dims` array.

**`ndim` — Number of dimensions**
`mwSize`

Number of dimensions, specified as `mwSize`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxsetdimensionsf.F`

## See Also

`mxGetNumberOfDimensions` | `mxRealloc` | `mxSetM` | `mxSetN`

**Introduced before R2006a**

# mxSetField (C and Fortran)

Set field value in structure array, given index and field name

## C Syntax

```
#include "matrix.h"
void mxSetField(mxArray *pm, mwIndex index,
  const char *fieldname, mxArray *pvalue);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetField(pm, index, fieldname, pvalue)
mwPointer pm, pvalue
mwIndex index
character*(*) fieldname
```

## Arguments

`pm`

Pointer to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.

`index`

Index of an element in the array.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

See `mxCalcSingleSubscript` for details on calculating an index.

`fieldname`

Name of a field in the structure. The field must exist in the structure. Call `mxGetFieldNameByNumber` or `mxGetFieldNumber` to determine existing field names.

`pvalue`

Pointer to an `mxArray` containing the data you want to assign to `fieldname`.

## Description

Use `mxSetField` to assign the contents of `pvalue` to the field `fieldname` of element `index`.

If you want to replace the contents of `fieldname`, then first free the memory of the existing data. Use the `mxGetField` function to get a pointer to the field, call `mxDestroyArray` on the pointer, then call `mxSetField` to assign the new value.

You cannot assign `pvalue` to more than one field in a structure or to more than one element in the `mxArray`. If you want to assign the contents of `pvalue` to multiple fields, then use the `mxDuplicateArray` function to make copies of the data then call `mxSetField` on each copy.

To free memory for structures created using this function, call `mxDestroyArray` only on the structure array. Do not call `mxDestroyArray` on the array `pvalue` points to. If you do, then MATLAB attempts to free the same memory twice, which can corrupt memory.

---

**Note** Inputs to a MEX-file are constant read-only `mxArrays`. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

---

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreatestructarray.c`

## See Also

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetField`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetFieldByNumber`, `mxDestroyArray`, `mxCalcSingleSubscript`

## Alternatives

### C Language

In C, you can replace the statements:

```
field_num = mxGetFieldNumber(pa, "fieldname");
mxSetFieldByNumber(pa, index, field_num, new_value_pa);
```

with a call to `mxSetField`:

```
mxSetField(pa, index, "fieldname", new_value_pa);
```

### Fortran Language

In Fortran, you can replace the statements:

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxSetFieldByNumber(pm, index, fieldnum, newvalue)
```

with a call to `mxSetField`:

```
mxSetField(pm, index, 'fieldname', newvalue)
```

**Introduced before R2006a**

# mxSetFieldByNumber (C and Fortran)

Set field value in structure array, given index and field number

## C Syntax

```
#include "matrix.h"
void mxSetFieldByNumber(mxArray *pm, mwIndex index,
  int fieldnumber, mxArray *pvalue);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetFieldByNumber(pm, index, fieldnumber, pvalue)
mwPointer pm, pvalue
mwIndex index
integer*4 fieldnumber
```

## Arguments

`pm`

Pointer to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.

`index`

Index of the desired element.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

See `mxCalcSingleSubscript` for details on calculating an index.

`fieldnumber`

Position of the field in the structure. The field must exist in the structure.

In C, the first field within each element has a `fieldnumber` of `0`. The `fieldnumber` of the last is `N-1`, where `N` is the number of fields.

In Fortran, the first field within each element has a `fieldnumber` of `1`. The `fieldnumber` of the last is `N`, where `N` is the number of fields.

`pvalue`

Pointer to the `mxArray` containing the data you want to assign.

## Description

Use `mxSetFieldByNumber` to assign the contents of `pvalue` to the field specified by `fieldnumber` of element `index`. `mxSetFieldByNumber` is like `mxSetField`; however, the function identifies the field by position number, not by name.

If you want to replace the contents at `fieldnumber`, then first free the memory of the existing data. Use the `mxGetFieldByNumber` function to get a pointer to the field, call `mxDestroyArray` on the pointer, then call `mxSetFieldByNumber` to assign the new value.

You cannot assign `pvalue` to more than one field in a structure or to more than one element in the `mxArray`. If you want to assign the contents of `pvalue` to multiple fields, then use the `mxDuplicateArray` function to make copies of the data then call `mxSetFieldByNumber` on each copy.

To free memory for structures created using this function, call `mxDestroyArray` only on the structure array. Do not call `mxDestroyArray` on the array `pvalue` points to. If you do, then MATLAB attempts to free the same memory twice, which can corrupt memory.

---

**Note** Inputs to a MEX-file are constant read-only `mxArray`s. Do not modify the inputs. Using `mxSetCell*` or `mxSetField*` functions to modify the cells or fields of a MATLAB argument causes unpredictable results.

---

## Alternatives

### C Language

In C, calling:

```
mxSetField(pa, index, "field_name", new_value_pa);
```

is equivalent to calling:

```
field_num = mxGetFieldNumber(pa, "field_name");
mxSetFieldByNumber(pa, index, field_num, new_value_pa);
```

### Fortran Language

In Fortran, calling:

```
mxSetField(pm, index, 'fieldname', newvalue)
```

is equivalent to calling:

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxSetFieldByNumber(pm, index, fieldnum, newvalue)
```

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxcreatestructarray.c`

## See Also

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxDestroyArray`, `mxCalcSingleSubscript`

**Introduced before R2006a**

# mxSetImagData (C)

Set imaginary data elements in numeric mxArray

**Note** mxSetImagData is not available in the interleaved complex API. Use typed, data-access functions instead. For more information, see "Compatibility Considerations".

## C Syntax

```
#include "matrix.h"
void mxSetImagData(mxArray *pm, void *pi);
```

## Description

The mxSetImagData function is similar to mxSetPi, except that in C, its pi argument is a void *. Use this function on numeric arrays with contents other than double.

The mxSetImagData function does not free memory allocated for existing data. To free existing memory, call mxFree on the pointer returned by mxGetImagData.

## Input Arguments

**pm — Pointer to MATLAB array**
mxArray*

Pointer to a MATLAB array, specified as mxArray *.

**pi — Pointer to complex data array**
void*

Pointer to the complex data array within an mxArray, specified as void *. Each element in the array contains the imaginary component of a value.

The array must be in dynamic memory. Call mxCalloc to allocate this memory. Do not use the ANSI C calloc function, which can cause memory alignment issues leading to program termination. If pi points to static memory, then memory errors result when the array is destroyed.

## Compatibility Considerations

**Do not use separate complex API**
*Not recommended starting in R2018a*

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. These functions verify that the input array is complex and of the correct type for the function. For more information, see:

- "Typed Data Access in C MEX Files"
- "MATLAB Support for Interleaved Complex API in MEX Functions"

- “Upgrade MEX Files to Use Interleaved Complex API”

To build the MEX file, call `mex` with the `-R2018a` option.

**Error building `mxSetImagData` with interleaved complex API**
*Errors starting in R2018a*

The `mxSetImagData` function is only available in the separate complex API. To build `myMexFile.c` using this function, type:

```
mex -R2017b myMexFile.c
```

Existing MEX files built with this function continue to run.

## See Also

**Topics**
“Typed Data Access in C MEX Files”
“MATLAB Support for Interleaved Complex API in MEX Functions”

**Introduced before R2006a**

# mxSetImagData (Fortran)

Set imaginary data elements in numeric `mxArray`

---

**Note** `mxSetImagData` is not available in the interleaved complex API. Use typed, data-access functions instead. For more information, see “Compatibility Considerations”.

---

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetImagData(pm, pi)
mwPointer pm, pi
```

## Description

The `mxSetImagData` function is similar to `mxSetPi`. Use this function on numeric arrays with contents other than `double`.

The `mxSetImagData` function does not free memory allocated for existing data. To free existing memory, call `mxFree` on the pointer returned by `mxGetImagData`.

## Input Arguments

**pm — Pointer to MATLAB array**
`mwPointer`

Pointer to a MATLAB array, specified as `mwPointer`.

**pi — Pointer to complex data array**
`mwPointer`

Pointer to the complex data array within an `mxArray`, specified as `mwPointer`. Each element in the array contains the imaginary component of a value.

The array must be in dynamic memory; call `mxCalloc` to allocate this memory. If `pi` points to static memory, then memory errors result when the array is destroyed.

## Compatibility Considerations

**Do not use separate complex API**
*Not recommended starting in R2018b*

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. These functions verify that the input array is complex and of the correct type for the function. For more information, see:

- “Typed Data Access in C MEX Files”
- “MATLAB Support for Interleaved Complex API in MEX Functions”

- “Upgrade MEX Files to Use Interleaved Complex API”

To build the MEX file, call `mex` with the `-R2018a` option.

**Error building `mxSetImagData` with interleaved complex API**
*Errors starting in R2018b*

The `mxSetImagData` function is only available in the separate complex API. To build `myMexFile.F` using this function, type:

```
mex -R2017b myMexFile.F
```

Existing MEX files built with this function continue to run.

## See Also

**Topics**
“Typed Data Access in C MEX Files”
“MATLAB Support for Interleaved Complex API in MEX Functions”

**Introduced before R2006a**

# mxSetIr (C and Fortran)

IR array of sparse array

## C Syntax

```
#include "matrix.h"
void mxSetIr(mxArray *pm, mwIndex *ir);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetIr(pm, ir)
mwPointer pm, ir
```

## Arguments

`pm`

Pointer to a sparse `mxArray`

`ir`

Pointer to the `ir` array. The `ir` array must be sorted in column-major order.

## Description

Use `mxSetIr` to specify the `ir` array of a sparse `mxArray`. The `ir` array is an array of integers; the length of the `ir` array equals the value of `nzmax`, the storage allocated for the sparse array, or `nnz`, the number of nonzero matrix elements.

Each element in the `ir` array indicates a row (offset by 1) at which a nonzero element can be found. (The `jc` array is an index that indirectly specifies a column where nonzero elements can be found. See `mxSetJc` for more details on `jc`.)

For example, suppose that you create a `7`-by-`3` sparse `mxArray` named `Sparrow` containing six nonzero elements by typing:

```
Sparrow = zeros(7,3);
Sparrow(2,1) = 1;
Sparrow(5,1) = 1;
Sparrow(3,2) = 1;
Sparrow(2,3) = 2;
Sparrow(5,3) = 1;
Sparrow(6,3) = 1;
Sparrow = sparse(Sparrow);
```

The `pr` array holds the real data for the sparse matrix, which in `Sparrow` is the five 1s and the one 2. If there is any nonzero imaginary data, then it is in a `pi` array.

| Subscript | ir | pr | jc | Comments |
|---|---|---|---|---|
| `(2,1)` | `1` | `1` | `0` | Column 1; `ir` is 1 because row is 2. |

| Subscript | ir | pr | jc | Comments |
|---|---|---|---|---|
| `(5,1)` | 4 | 1 | 2 | Column 1; `ir` is 4 because row is 5. |
| `(3,2)` | 2 | 1 | 3 | Column 2; `ir` is 2 because row is 3. |
| `(2,3)` | 1 | 2 | 6 | Column 3; `ir` is 1 because row is 2. |
| `(5,3)` | 4 | 1 | | Column 3; `ir` is 4 because row is 5. |
| `(6,3)` | 5 | 1 | | Column 3; `ir` is 5 because row is 6. |

Notice how each element of the `ir` array is always 1 less than the row of the corresponding nonzero element. For instance, the first nonzero element is in row 2; therefore, the first element in `ir` is 1 (that is, 2 – 1). The second nonzero element is in row 5; therefore, the second element in `ir` is 4 (5 – 1).

The `ir` array must be in column-major order. The `ir` array must define the row positions in column 1 (if any) first, then the row positions in column 2 (if any) second, and so on, through column N. Within each column, row position 1 must appear before row position 2, and so on.

`mxSetIr` does not sort the `ir` array for you; you must specify an `ir` array that is already sorted.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetIr` before you call `mxSetIr`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxsetnzmax.c`

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## See Also

`mxCreateSparse`, `mxGetIr`, `mxGetJc`, `mxSetJc`, `mxFree`, `nzmax`, `nnz`

**Introduced before R2006a**

# mxSetJc (C and Fortran)

JC array of sparse array

## C Syntax

```
#include "matrix.h"
void mxSetJc(mxArray *pm, mwIndex *jc);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetJc(pm, jc)
mwPointer pm, jc
```

## Arguments

`pm`

Pointer to a sparse `mxArray`

`jc`

Pointer to the `jc` array

## Description

Use `mxSetJc` to specify a new `jc` array for a sparse `mxArray`. The `jc` array is an integer array having `n+1` elements, where `n` is the number of columns in the sparse `mxArray`.

If the `j`th column of the sparse `mxArray` has any nonzero elements, then:

- `jc[j]` is the index in `ir`, `pr`, and `pi` (if it exists) of the first nonzero element in the `j`th column.
- `jc[j+1]-1` is the index of the last nonzero element in the `j`th column.
- For the `j`th column of the sparse matrix, `jc[j]` is the total number of nonzero elements in all preceding columns.

The number of nonzero elements in the `j`th column of the sparse `mxArray` is:

```
jc[j+1] - jc[j];
```

For the `j`th column of the sparse `mxArray`, `jc[j]` is the total number of nonzero elements in all preceding columns. The last element of the `jc` array, `jc[number of columns]`, is equal to `nnz`, which is the number of nonzero elements in the entire sparse `mxArray`.

For example, consider a `7`-by-`3` sparse `mxArray` named `Sparrow` containing six nonzero elements, created by typing:

```
Sparrow = zeros(7,3);
Sparrow(2,1) = 1;
Sparrow(5,1) = 1;
Sparrow(3,2) = 1;
Sparrow(2,3) = 2;
```

```
Sparrow(5,3) = 1;
Sparrow(6,3) = 1;
Sparrow = sparse(Sparrow);
```

The following table lists the contents of the `ir`, `jc`, and `pr` arrays.

| Subscript | ir | pr | jc | Comment |
|---|---|---|---|---|
| (2,1) | 1 | 1 | 0 | Column 1 contains two nonzero elements, with rows designated by `ir[0]` and `ir[1]` |
| (5,1) | 4 | 1 | 2 | Column 2 contains one nonzero element, with row designated by `ir[2]` |
| (3,2) | 2 | 1 | 3 | Column 3 contains three nonzero elements, with rows designated by `ir[3]`,`ir[4]`, and `ir[5]` |
| (2,3) | 1 | 2 | 6 | There are six nonzero elements in all. |
| (5,3) | 4 | 1 | | |
| (6,3) | 5 | 1 | | |

As an example of a much sparser `mxArray`, consider a `1000`-by-`8` sparse `mxArray` named `Spacious` containing only three nonzero elements. The `ir`, `pr`, and `jc` arrays contain the values listed in this table.

| Subscript | ir | pr | jc | Comment |
|---|---|---|---|---|
| (73,2) | 72 | 1 | 0 | Column 1 contains no nonzero elements. |
| (50,3) | 49 | 1 | 0 | Column 2 contains one nonzero element, with row designated by `ir[0]`. |
| (64,5) | 63 | 1 | 1 | Column 3 contains one nonzero element, with row designated by `ir[1]`. |
| | | | 2 | Column 4 contains no nonzero elements. |
| | | | 2 | Column 5 contains one nonzero element, with row designated by `ir[2]`. |
| | | | 3 | Column 6 contains no nonzero elements. |
| | | | 3 | Column 7 contains no nonzero elements. |
| | | | 3 | Column 8 contains no nonzero elements. |
| | | | 3 | There are three nonzero elements in all. |

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetJc` before you call `mxSetJc`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxsetdimensions.c`

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## See Also

`mxCreateSparse`, `mxGetIr`, `mxGetJc`, `mxSetIr`, `mxFree`

**Introduced before R2006a**

# mxSetM (C)

Set number of rows in mxArray

## C Syntax

```
#include "matrix.h"
void mxSetM(mxArray *pm, mwSize m);
```

## Description

`mxSetM` sets the number of rows in the specified `mxArray`. The term *rows* means the first dimension of an `mxArray`, regardless of the number of dimensions. Call `mxSetN` to set the number of columns.

You typically use `mxSetM` to change the shape of an existing `mxArray`. The `mxSetM` function does not allocate or deallocate any space for the `pr`, `pi`, `ir`, or `jc` arrays. So, if your calls to `mxSetM` and `mxSetN` increase the number of elements in the `mxArray`, then enlarge the `pr`, `pi`, `ir`, and/or `jc` arrays. Call `mxRealloc` to enlarge them.

If calling `mxSetM` and `mxSetN` reduces the number of elements in the `mxArray`, then you might want to reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays to use heap space more efficiently. However, reducing the size is not mandatory.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

**m — Number of rows**
`mwSize`

Number of rows, specified as `mwSize`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxsetdimensions.c`

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `sincall.c`

## See Also

`mxGetM` | `mxGetN` | `mxRealloc` | `mxSetN`

**Introduced before R2006a**

# mxSetM (Fortran)

Set number of rows in mxArray

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetM(pm, m)
mwPointer pm
mwSize m
```

## Description

`mxSetM` sets the number of rows in the specified `mxArray`. The term *rows* means the first dimension of an `mxArray`, regardless of the number of dimensions. Call `mxSetN` to set the number of columns.

You typically use `mxSetM` to change the shape of an existing `mxArray`. The `mxSetM` function does not allocate or deallocate any space for the `pr`, `pi`, `ir`, or `jc` arrays. So, if your calls to `mxSetM` and `mxSetN` increase the number of elements in the `mxArray`, then enlarge the `pr`, `pi`, `ir`, and/or `jc` arrays. Call `mxRealloc` to enlarge them.

If calling `mxSetM` and `mxSetN` reduces the number of elements in the `mxArray`, then you might want to reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays to use heap space more efficiently. However, reducing the size is not mandatory.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

**m — Number of rows**
`mwSize`

Number of rows, specified as `mwSize`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `sincall.F`

## See Also

`mxGetM` | `mxGetN` | `mxRealloc` | `mxSetN`

**Introduced before R2006a**

# mxSetN (C)

Set number of columns in mxArray

## C Syntax

```
#include "matrix.h"
void mxSetN(mxArray *pm, mwSize n);
```

## Description

`mxSetN` sets the number of columns in the specified `mxArray`. The term *columns* always means the second dimension of a matrix. Calling `mxSetN` forces an `mxArray` to have two dimensions. For example, if `pm` points to an `mxArray` having three dimensions, then calling `mxSetN` reduces the `mxArray` to two dimensions.

You typically use `mxSetN` to change the shape of an existing `mxArray`. The `mxSetN` function does not allocate or deallocate any space for the `pr`, `pi`, `ir`, or `jc` arrays. So, if your calls to `mxSetN` and `mxSetM` increase the number of elements in the `mxArray`, then enlarge the `pr`, `pi`, `ir`, and/or `jc` arrays.

If calling `mxSetM` and `mxSetN` reduces the number of elements in the `mxArray`, then you might want to reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays to use heap space more efficiently. However, reducing the size is not mandatory.

## Input Arguments

**pm — MATLAB array**
`const mxArray*`

Pointer to an `mxArray` array, specified as `const mxArray*`.

**n — Number of columns**
`mwSize`

Number of columns, specified as `mwSize`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxsetdimensions.c`

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `sincall.c`

## See Also

`mxGetM` | `mxGetN` | `mxSetM`

**Introduced before R2006a**

# mxSetN (Fortran)

Set number of columns in mxArray

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetN(pm, n)
mwPointer pm
mwSize n
```

## Description

`mxSetN` sets the number of columns in the specified `mxArray`. The term *columns* always means the second dimension of a matrix. Calling `mxSetN` forces an `mxArray` to have two dimensions. For example, if `pm` points to an `mxArray` having three dimensions, then calling `mxSetN` reduces the `mxArray` to two dimensions.

You typically use `mxSetN` to change the shape of an existing `mxArray`. The `mxSetN` function does not allocate or deallocate any space for the `pr`, `pi`, `ir`, or `jc` arrays. So, if your calls to `mxSetN` and `mxSetM` increase the number of elements in the `mxArray`, then enlarge the `pr`, `pi`, `ir`, and/or `jc` arrays.

If calling `mxSetM` and `mxSetN` reduces the number of elements in the `mxArray`, then you might want to reduce the sizes of the `pr`, `pi`, `ir`, and/or `jc` arrays to use heap space more efficiently. However, reducing the size is not mandatory.

## Input Arguments

**pm — MATLAB array**
`mwPointer`

Pointer to an `mxArray` array, specified as `mwPointer`.

**n — Number of columns**
`mwSize`

Number of columns, specified as `mwSize`.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `sincall.F`

## See Also

`mxGetM` | `mxGetN` | `mxSetM`

**Introduced before R2006a**

# mxSetNzmax (C and Fortran)

Set storage space for nonzero elements

## C Syntax

```
#include "matrix.h"
void mxSetNzmax(mxArray *pm, mwSize nzmax);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetNzmax(pm, nzmax)
mwPointer pm
mwSize nzmax
```

## Arguments

`pm`

Pointer to a sparse `mxArray`.

`nzmax`

Number of elements for `mxCreateSparse` to allocate to hold the arrays pointed to by `ir`, `pr`, and `pi` (if it exists). Set `nzmax` greater than or equal to the number of nonzero elements in the `mxArray`, but set it to be less than or equal to the number of rows times the number of columns. If you specify an `nzmax` value of `0`, then `mxSetNzmax` sets the value of `nzmax` to `1`.

## Description

Use `mxSetNzmax` to assign a new value to the `nzmax` field of the specified sparse `mxArray`. The `nzmax` field holds the maximum number of nonzero elements in the sparse `mxArray`.

The number of elements in the `ir`, `pr`, and `pi` (if it exists) arrays must be equal to `nzmax`. Therefore, after calling `mxSetNzmax`, you must change the size of the `ir`, `pr`, and `pi` arrays. To change the size of one of these arrays:

1. Call `mxRealloc` with a pointer to the array, setting the size to the new value of `nzmax`.
2. Call the appropriate `mxSet` routine (`mxSetIr`, `mxSetDoubles`, or `mxSetComplexDoubles`) to establish the new memory area as the current one.

Ways to determine how large to make `nzmax` are:

- Set `nzmax` equal to or slightly greater than the number of nonzero elements in a sparse `mxArray`. This approach conserves precious heap space.
- Make `nzmax` equal to the total number of elements in an `mxArray`. This approach eliminates (or, at least reduces) expensive reallocations.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mx`:

- `mxsetnzmax.c`

## See Also
`mxGetNzmax` | `mxRealloc`

**Introduced before R2006a**

# mxSetPi (C)

(Not recommended) Set imaginary data elements in `mxDOUBLE_CLASS` array

**Note** `mxSetPi` is not available in the interleaved complex API. Use `mxSetComplexDoubles` instead. For more information, see "Compatibility Considerations".

## C Syntax

```
#include "matrix.h"
void mxSetPi(mxArray *pm, double *pi);
```

## Description

Use `mxSetPi` to set the imaginary data of the specified `mxArray`.

Most `mxCreate*` functions optionally allocate heap space to hold imaginary data. If you allocate heap space when calling an `mxCreate*` function, then do not use `mxSetPi` to initialize the imaginary elements of the array. Instead, call this function to replace existing values with new values. Examples of allocating heap space include setting the `ComplexFlag` to `mxCOMPLEX` or setting `pi` to a non-`NULL` value.

The `mxSetPi` function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetPi`.

## Input Arguments

**pm — Pointer to MATLAB array**
`mxArray *`

Pointer to a MATLAB array of type `mxDOUBLE_CLASS`, specified as `mxArray *`.

**pi — Pointer to data array**
`double *`

Pointer to the first `mxDouble` element of the imaginary part of the data array within an `mxArray`, specified as `double *`. Each element in the array contains the imaginary component of a value.

The array must be in dynamic memory. Call `mxCalloc` to allocate this memory. Do not use the ANSI C `calloc` function, which can cause memory alignment issues leading to program termination. If `pi` points to static memory, then memory leaks and other memory errors can result.

## Compatibility Considerations

**Do not use separate complex API**
*Not recommended starting in R2018a*

Use the `mxSetComplexDoubles` function in the interleaved complex API instead of `mxSetPr` and `mxSetPi`. This function verifies that the input array is complex and of type `mxDOUBLE_CLASS`.

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- “Typed Data Access in C MEX Files”
- “MATLAB Support for Interleaved Complex API in MEX Functions”
- “Upgrade MEX Files to Use Interleaved Complex API”
- Example `arrayFillSetPr.c`

To build the MEX file, call `mex` with the `-R2018a` option.

**Error building `mxSetPi` with interleaved complex API**
*Errors starting in R2018a*

The `mxSetPi` function is only available in the separate complex API. To build `myMexFile.c` using this function, type:

```
mex -R2017b myMexFile.c
```

Existing MEX files built with this function continue to run.

## See Also

`mxSetComplexDoubles`

### Topics

`arrayFillSetPr.c`
“Typed Data Access in C MEX Files”
“MATLAB Support for Interleaved Complex API in MEX Functions”

**Introduced before R2006a**

# mxSetPi (Fortran)

(Not recommended) Set imaginary data elements in mxDOUBLE_CLASS array

**Note** mxSetPi is not available in the interleaved complex API. Use mxSetComplexDoubles instead. For more information, see "Compatibility Considerations".

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetPi(pm, pi)
mwPointer pm, pi
```

## Description

Use mxSetPi to set the imaginary data of the specified mxArray.

Most mxCreate* functions optionally allocate heap space to hold imaginary data. If you allocate heap space when calling an mxCreate* function, then do not use mxSetPi to initialize the imaginary elements of the array. Rather, call this function to replace existing values with new values. Examples of allocating heap space include setting the ComplexFlag to mxCOMPLEX or setting pi to a non-0 value.

The mxSetPi function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetPi.

## Input Arguments

**pm — Pointer to MATLAB array**
mwPointer

Pointer to a MATLAB array of type mxDOUBLE_CLASS, specified as mwPointer.

**pi — Pointer to data array**
mwPointer

Pointer to the first mxDouble element of the imaginary part of the data array within an mxArray, specified as mwPointer. Each element in the array contains the imaginary component of a value.

The array must be in dynamic memory; call mxCalloc to allocate this memory. If pi points to static memory, then memory leaks and other memory errors can result.

## Compatibility Considerations

**Do not use separate complex API**
*Not recommended starting in R2018b*

Use the mxSetComplexDoubles function in the interleaved complex API instead of mxSetPr and mxSetPi. This function verifies that the input array is complex and of type mxDOUBLE_CLASS.

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- “Typed Data Access in C MEX Files”
- “MATLAB Support for Interleaved Complex API in MEX Functions”
- “Upgrade MEX Files to Use Interleaved Complex API”

To build the MEX file, call `mex` with the `-R2018a` option.

**Error building `mxSetPi` with interleaved complex API**
*Errors starting in R2018b*

The `mxSetPi` function is only available in the separate complex API. To build `myMexFile.F` using this function, type:

```
mex -R2017b myMexFile.F
```

Existing MEX files built with this function continue to run.

## See Also

`mxSetComplexDoubles`

**Topics**
“Typed Data Access in C MEX Files”
“MATLAB Support for Interleaved Complex API in MEX Functions”

**Introduced before R2006a**

# mxSetPr (C)

(Not recommended) Set real data elements in mxDOUBLE_CLASS array

**Note** mxSetPr is not available in the interleaved complex API. Use mxSetDoubles or mxSetComplexDoubles instead. For more information, see “Compatibility Considerations”.

## C Syntax

```
#include "matrix.h"
void mxSetPr(mxArray *pm, double *pr);
```

## Description

The mxSetPr function sets the real data of a real mxDOUBLE_CLASS array pm. If you build with the interleaved complex API (mex -R2018a option) and pm is complex, then the function terminates the MEX file and returns control to the MATLAB prompt. In a non-MEX file application, the function returns NULL.

Call mxIsDouble to validate the mxArray type. Call mxIsComplex to determine whether the data is real.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the real elements of an array. Instead, call this function to replace the existing values with new values.

The mxSetPr function does not free memory allocated for existing data. To free existing memory, call mxFree on the pointer returned by mxGetPr.

## Input Arguments

**pm — Pointer to MATLAB array**
mxArray *

Pointer to a MATLAB array of type mxDOUBLE_CLASS, specified as mxArray *.

**pr — Pointer to data array**
double *

Pointer to the first mxDouble element of the real part of the data array within an mxArray, specified as double *. Each element in the array contains the real component of a value.

The array must be in dynamic memory. Call mxCalloc to allocate this memory. Do not use the ANSI C calloc function, which can cause memory alignment issues leading to program termination. If pr points to static memory, then memory leaks and other memory errors can result.

## Compatibility Considerations

**Do not use separate complex API**
*Not recommended starting in R2018a*

Use the `mxSetDoubles` function in the interleaved complex API for real arrays of type `mxDOUBLE_CLASS`. Use `mxSetComplexDoubles` for complex arrays of type `mxDOUBLE_CLASS`. These functions validate the type and complexity of the input.

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- "Typed Data Access in C MEX Files"
- "MATLAB Support for Interleaved Complex API in MEX Functions"
- "Upgrade MEX Files to Use Interleaved Complex API"
- Example `fulltosparse.c`

To build the MEX file, call `mex` with the `-R2018a` option.

**Runtime error calling `mxSetPr` on complex `mxArrays` in applications built with interleaved complex API**
*Errors starting in R2018a*

Use the `mxSetComplexDoubles` function instead of `mxSetPr` and `mxGetPi`. For an example showing how to update code that uses `mxSetPr`, see `mxsetnzmax.c`.

## See Also

`mxSetComplexDoubles` | `mxSetDoubles`

### Topics

`fulltosparse.c`
`mxsetnzmax.c`
"Typed Data Access in C MEX Files"
"MATLAB Support for Interleaved Complex API in MEX Functions"

**Introduced before R2006a**

# mxSetPr (Fortran)

(Not recommended) Set real data elements in `mxDOUBLE_CLASS` array

---

**Note** `mxSetPr` is not available in the interleaved complex API. Use `mxSetDoubles` or `mxSetComplexDoubles` instead. For more information, see "Compatibility Considerations".

---

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetPr(pm, pr)
mwPointer pm, pr
```

## Description

The `mxSetPr` function sets the real data of a real `mxDOUBLE_CLASS` array `pm`. If you build with the interleaved complex API (`mex -R2018a` option) and `pm` is complex, then the function terminates the MEX file and returns control to the MATLAB prompt. In a non-MEX file application, the function returns `0`.

Call `mxIsDouble` to validate the `mxArray` type. Call `mxIsComplex` to determine whether the data is real.

All `mxCreate*` functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the real elements of an array. Instead, call this function to replace the existing values with new values.

The `mxSetPr` function does not free memory allocated for existing data. To free existing memory, call `mxFree` on the pointer returned by `mxGetPr`.

## Input Arguments

**pm — Pointer to MATLAB array**
`mwPointer`

Pointer to a MATLAB array of type `mxDOUBLE_CLASS`, specified as `mwPointer`.

**pr — Pointer to data array**
`mwPointer`

Pointer to the first `mxDouble` element of the real part of the data array within an `mxArray`, specified as `mwPointer`. Each element in the array contains the real component of a value.

The array must be in dynamic memory. Call `mxCalloc` to allocate this memory. If `pr` points to static memory, then memory leaks and other memory errors can result.

## Compatibility Considerations

**Do not use separate complex API**
*Not recommended starting in R2018b*

Use the `mxSetDoubles` function in the interleaved complex API for real arrays of type `mxDOUBLE_CLASS`. Use `mxSetComplexDoubles` for complex arrays of type `mxDOUBLE_CLASS`. These functions validate the type and complexity of the input.

MathWorks recommends that you create MEX files and update existing MEX files to use the typed, data-access functions in the interleaved complex API. For more information, see:

- "Typed Data Access in C MEX Files"
- "MATLAB Support for Interleaved Complex API in MEX Functions"
- "Upgrade MEX Files to Use Interleaved Complex API"

To build the MEX file, call `mex` with the `-R2018a` option.

**Runtime error calling `mxSetPr` on complex `mxArrays` in applications built with interleaved complex API**
*Errors starting in R2018b*

Use the `mxSetComplexDoubles` function instead of `mxSetPr` and `mxSetPi`.

## See Also

`mxSetComplexDoubles` | `mxSetDoubles`

### Topics

"Typed Data Access in C MEX Files"
"MATLAB Support for Interleaved Complex API in MEX Functions"

**Introduced before R2006a**

# mxSetProperty (C and Fortran)

Set value of public property of MATLAB object

## C Syntax

```
#include "matrix.h"
void mxSetProperty(mxArray *pa, mwIndex index,
  const char *propname, const mxArray *value);
```

## Fortran Syntax

```
#include "fintrf.h"
subroutine mxSetProperty(pa, index, propname, value)
mwPointer pa, value
mwIndex index
character*(*) propname
```

## Arguments

`pa`

Pointer to an `mxArray` which is an object.

`index`

Index of the desired element of the object array.

In C, the first element of an `mxArray` has an `index` of `0`. The `index` of the last element is `N-1`, where `N` is the number of elements in the array. In Fortran, the first element of an `mxArray` has an `index` of `1`. The `index` of the last element is `N`, where `N` is the number of elements in the array.

`propname`

Name of the property whose value you are assigning.

value

Pointer to the `mxArray` you are assigning.

## Description

Use `mxSetProperty` to assign a `value` to the specified property. In pseudo-C terminology, `mxSetProperty` performs the assignment:

```
pa[index].propname = value;
```

Property `propname` must be an existing, public property and `index` must be within the bounds of the `mxArray`. To test the index value, use `mxGetNumberOfElements` or `mxGetM` and `mxGetN` functions.

`mxSetProperty` makes a copy of the value before assigning it as the new property value. If the property uses a large amount of memory, then making a copy might be a concern. There must be sufficient memory in the heap to hold the copy of the value.

## Limitations

- `mxSetProperty` is not supported for standalone applications, such as applications built with the MATLAB engine API.
- Properties of type `datetime` are not supported.

## See Also

`mxGetProperty`

### Topics

"matlab::engine::MATLABEngine::setProperty"

**Introduced in R2008a**

# mxGetDoubles (C)

Real data elements in mxDOUBLE_CLASS array

## C Syntax

```
#include "matrix.h"
mxDouble *mxGetDoubles(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxDOUBLE_CLASS array.

## Output Arguments

**dt — Data array**
mxDouble * | NULL

Pointer to the first mxDouble element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxDOUBLE_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxDOUBLE_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetComplexDoubles | mxSetDoubles

**Introduced in R2018a**

# mxGetComplexDoubles (C)

Complex data elements in mxDOUBLE_CLASS array

## C Syntax

```
#include "matrix.h"
mxComplexDouble *mxGetComplexDoubles(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxDOUBLE_CLASS array.

## Output Arguments

**dt — Data array**
mxComplexDouble *

Pointer to the first mxComplexDouble element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxDOUBLE_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxDOUBLE_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetDoubles | mxSetComplexDoubles

**Introduced in R2018a**

# mxSetDoubles (C)

Set real data elements in mxDOUBLE_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetDoubles(mxArray *pa, mxDouble *dt);
```

## Description

Use mxSetDoubles to set mxDouble data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxDOUBLE_CLASS array.

**dt — Data array**
mxDouble *

Pointer to the first mxDouble element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxDOUBLE_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

See the arrayFillSetPr.c example in the *matlabroot*/extern/examples/refbook folder.

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetDoubles` | `mxSetComplexDoubles`

**Introduced in R2018a**

# mxSetComplexDoubles (C)

Set complex data elements in mxDOUBLE_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetComplexDoubles(mxArray *pa, mxComplexDouble *dt);
```

## Description

Use mxSetComplexDoubles to set mxComplexDouble data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxDOUBLE_CLASS array.

**dt — Data array**
mxComplexDouble *

Pointer to the first mxComplexDouble element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxDOUBLE_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing data into an mxArray. The data in the example is defined as mxDouble. You can use this example as a pattern for any C numeric type, including complex. Suppose that you have an array with these values.

```
2.0 + 3.0i
3.0 + 4.0i
```

To modify this example for complex `mxDouble` data:

- Declare data variables

  ```
  mxComplexDouble *dynamicData;
  const mxComplexDouble data[] = {{2.0, 3.0}, {3.0, 4.0}};
  ```
- Call `mxCreateNumericMatrix` with the `mxCOMPLEX` argument
- Replace `mxSetDoubles` with `mxSetComplexDoubles` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetComplexDoubles` | `mxSetDoubles`

**Introduced in R2018a**

# mxGetInt16s (C)

Real data elements in mxINT16_CLASS array

## C Syntax

```
#include "matrix.h"
mxInt16 *mxGetInt16s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxINT16_CLASS array.

## Output Arguments

**dt — Data array**
mxInt16 *

Pointer to the first mxInt16 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxINT16_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxINT16_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetComplexInt16s | mxGetUint16s | mxSetInt16s

**Introduced in R2018a**

# mxGetComplexInt16s (C)

Complex data elements in `mxINT16_CLASS` array

## C Syntax

```
#include "matrix.h"
mxComplexInt16 *mxGetComplexInt16s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
`const mxArray *`

Pointer to an `mxINT16_CLASS` array.

## Output Arguments

**dt — Data array**
`mxComplexInt16 *`

Pointer to the first `mxComplexInt16` element of the data. If `pa` is `NULL`, then the function returns `NULL`.

If `mxArray` is not an `mxINT16_CLASS` array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `NULL`. A `NULL` return value indicates that `pa` is either empty or not an `mxINT16_CLASS` array.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetInt16s` | `mxSetComplexInt16s`

**Introduced in R2018a**

# mxSetInt16s (C)

Set real data elements in mxINT16_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetInt16s(mxArray *pa, mxInt16 *dt);
```

## Description

Use mxSetInt16s to set mxInt16 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxINT16_CLASS array.

**dt — Data array**
mxInt16 *

Pointer to the first mxInt16 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT16_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing data into an mxArray. The data in the example is defined as mxDouble. To modify this example for int16 data:

- Declare the data variables as `mxInt16`
- Call `mxCreateNumericMatrix` with the numeric type `mxINT16_CLASS`
- Replace `mxSetDoubles` with `mxSetInt16s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetInt16s` | `mxSetComplexInt16s` | `mxSetUint16s`

**Introduced in R2018a**

# mxSetComplexInt16s (C)

Set complex data elements in mxINT16_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetComplexInt16s(mxArray *pa, mxComplexInt16 *dt);
```

## Description

Use mxSetComplexInt16s to set mxComplexInt16 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxINT16_CLASS array.

**dt — Data array**
mxComplexInt16 *

Pointer to the first mxComplexInt16 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT16_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetComplexPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing complex numeric data into an mxArray. The data in the

example is defined as `mxComplexDouble`. You can use this example as a pattern for any complex C numeric type. To modify this example for complex `int16` data:

- Declare the data variables as `mxComplexInt16`
- Call `mxCreateNumericMatrix` with the numeric type `mxINT16_CLASS`
- Replace `mxSetDoubles` with `mxSetComplexInt16s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetComplexInt16s` | `mxSetInt16s`

**Introduced in R2018a**

# mxGetInt32s (C)

Real data elements in mxINT32_CLASS array

## C Syntax

```
#include "matrix.h"
mxInt32 *mxGetInt32s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxINT32_CLASS array.

## Output Arguments

**dt — Data array**
mxInt32 *

Pointer to the first mxInt32 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxINT32_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxINT32_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetComplexInt32s | mxGetUint32s | mxSetInt32s

**Introduced in R2018a**

# mxGetComplexInt32s (C)

Complex data elements in `mxINT32_CLASS` array

## C Syntax

```
#include "matrix.h"
mxComplexInt32 *mxGetComplexInt32s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
`const mxArray *`

Pointer to an `mxINT32_CLASS` array.

## Output Arguments

**dt — Data array**
`mxComplexInt32 *`

Pointer to the first `mxComplexInt32` element of the data. If `pa` is `NULL`, then the function returns `NULL`.

If `mxArray` is not an `mxINT32_CLASS` array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `NULL`. A `NULL` return value indicates that `pa` is either empty or not an `mxINT32_CLASS` array.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetInt32s` | `mxSetComplexInt32s`

**Introduced in R2018a**

# mxSetInt32s (C)

Set real data elements in `mxINT32_CLASS` array

## C Syntax

```
#include "matrix.h"
int mxSetInt32s(mxArray *pa, mxInt32 *dt);
```

## Description

Use `mxSetInt32s` to set `mxInt32` data in the specified array.

All `mxCreate*` functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
`mxArray *`

Pointer to an `mxINT32_CLASS` array.

**dt — Data array**
`mxInt32 *`

Pointer to the first `mxInt32` element of the data array. `dt` must be allocated by the functions `mxCalloc` or `mxMalloc`.

## Output Arguments

**`status` — Function status**
`int`

Function status, returned as `int`. If successful, then the function returns `1`.

If `pa` is `NULL`, then the function returns `0`.

The function is unsuccessful when `mxArray` is not an unshared `mxINT32_CLASS` array, or if the data is not allocated with `mxCalloc`. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`.

## Examples

Refer to the `arrayFillSetPr.c` example in the *`matlabroot`*`/extern/examples/refbook` folder which copies existing data into an `mxArray`. The data in the example is defined as `mxDouble`. To modify this example for `int32` data:

- Declare the data variables as `mxInt32`
- Call `mxCreateNumericMatrix` with the numeric type `mxINT32_CLASS`
- Replace `mxSetDoubles` with `mxSetInt32s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetInt32s` | `mxSetComplexInt32s` | `mxSetUint32s`

**Introduced in R2018a**

# mxSetComplexInt32s (C)

Set complex data elements in mxINT32_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetComplexInt32s(mxArray *pa, mxComplexInt32 *dt);
```

## Description

Use mxSetComplexInt32s to set mxComplexInt32 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxINT32_CLASS array.

**dt — Data array**
mxComplexInt32 *

Pointer to the first mxComplexInt32 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT32_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetComplexPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing complex numeric data into an mxArray. The data in the

example is defined as `mxComplexDouble`. You can use this example as a pattern for any complex C numeric type. To modify this example for complex `int32` data:

- Declare the data variables as `mxComplexInt32`
- Call `mxCreateNumericMatrix` with the numeric type `mxINT32_CLASS`
- Replace `mxSetDoubles` with `mxSetComplexInt32s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetComplexInt32s` | `mxSetInt32s`

**Introduced in R2018a**

# mxGetInt64s (C)

Real data elements in mxINT64_CLASS array

## C Syntax

```
#include "matrix.h"
mxInt64 *mxGetInt64s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxINT64_CLASS array.

## Output Arguments

**dt — Data array**
mxInt64 *

Pointer to the first mxInt64 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxINT64_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxINT64_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetComplexInt64s | mxGetUint64s | mxSetInt64s

**Introduced in R2018a**

# mxGetComplexInt64s (C)

Complex data elements in `mxINT64_CLASS` array

## C Syntax

```
#include "matrix.h"
mxComplexInt64 *mxGetComplexInt64s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
`const mxArray *`

Pointer to an `mxINT64_CLASS` array.

## Output Arguments

**dt — Data array**
`mxComplexInt64 *`

Pointer to the first `mxComplexInt64` element of the data. If `pa` is `NULL`, then the function returns `NULL`.

If `mxArray` is not an `mxINT64_CLASS` array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `NULL`. A `NULL` return value indicates that `pa` is either empty or not an `mxINT64_CLASS` array.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetInt64s` | `mxSetComplexInt64s`

**Introduced in R2018a**

# mxSetInt64s (C)

Set data elements in mxINT64_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetInt64s(mxArray *pa, mxInt64 *dt);
```

## Description

Use mxSetInt64s to set mxInt64 data of the specified mxArray.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxINT64_CLASS array.

**dt — Data array**
mxInt64 *

Pointer to the first mxInt64 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT64_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing data into an mxArray. The data in the example is defined as mxDouble. To modify this example for int64 data:

- Declare the data variables as `mxInt64`
- Call `mxCreateNumericMatrix` with the numeric type `mxINT64_CLASS`
- Replace `mxSetDoubles` with `mxSetInt64s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetInt64s` | `mxSetComplexInt64s` | `mxSetUint64s`

**Introduced in R2018a**

# mxSetComplexInt64s (C)

Set complex data elements in mxINT64_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetComplexInt64s(mxArray *pa, mxComplexInt64 *dt);
```

## Description

Use mxSetComplexInt64s to set mxComplexInt64 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxINT64_CLASS array.

**dt — Data array**
mxComplexInt64 *

Pointer to the first mxComplexInt64 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT64_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetComplexPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing complex numeric data into an mxArray. The data in the

example is defined as `mxComplexDouble`. You can use this example as a pattern for any complex C numeric type. To modify this example for complex `int64` data:

- Declare the data variables as `mxComplexInt64`
- Call `mxCreateNumericMatrix` with the numeric type `mxINT64_CLASS`
- Replace `mxSetDoubles` with `mxSetComplexInt64s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetComplexInt64s` | `mxSetInt64s`

**Introduced in R2018a**

# mxGetInt8s (C)

Real data elements in mxINT8_CLASS array

## C Syntax

```
#include "matrix.h"
mxInt8 *mxGetInt8s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxINT8_CLASS array.

## Output Arguments

**dt — Data array**
mxInt8 *

Pointer to the first mxInt8 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxINT8_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxINT8_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetComplexInt8s | mxGetUint8s | mxSetInt8s

**Introduced in R2018a**

# mxGetComplexInt8s (C)

Complex data elements in mxINT8_CLASS array

## C Syntax

```
#include "matrix.h"
mxComplexInt8 *mxGetComplexInt8s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxINT8_CLASS array.

## Output Arguments

**dt — Data array**
mxComplexInt8 *

Pointer to the first mxComplexInt8 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxINT8_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxINT8_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetInt8s | mxSetComplexInt8s

**Introduced in R2018a**

# mxSetInt8s (C)

Set real data elements in mxINT8_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetInt8s(mxArray *pa, mxInt8 *dt);
```

## Description

Use mxSetInt8s to set mxInt8 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxINT8_CLASS array.

**dt — Data array**
mxInt8 *

Pointer to the first mxInt8 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT8_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing data into an mxArray. The data in the example is defined as mxDouble. To modify this example for int8 data:

- Declare the data variables as `mxInt8`
- Call `mxCreateNumericMatrix` with the numeric type `mxINT8_CLASS`
- Replace `mxSetDoubles` with `mxSetInt8s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetInt8s` | `mxSetComplexInt8s` | `mxSetUint8s`

**Introduced in R2018a**

# mxSetComplexInt8s (C)

Set complex data elements in mxINT8_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetComplexInt8s(mxArray *pa, mxComplexInt8 *dt);
```

## Description

Use mxSetComplexInt8s to set mxComplexInt8 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxINT8_CLASS array.

**dt — Data array**
mxComplexInt8 *

Pointer to the first mxComplexInt8 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT8_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetComplexPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing complex numeric data into an mxArray. The data in the

example is defined as `mxComplexDouble`. You can use this example as a pattern for any complex C numeric type. To modify this example for complex `int8` data:

- Declare the data variables as `mxComplexInt8`
- Call `mxCreateNumericMatrix` with the numeric type `mxINT8_CLASS`
- Replace `mxSetDoubles` with `mxSetComplexInt8s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetComplexInt8s` | `mxSetInt8s`

**Introduced in R2018a**

# mxGetSingles (C)

Real data elements in mxSINGLE_CLASS array

## C Syntax

```
#include "matrix.h"
mxSingle *mxGetSingles(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
`const mxArray *`

Pointer to an mxSINGLE_CLASS array.

## Output Arguments

**dt — Data array**
`mxSingle *`

Pointer to the first mxSingle element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxSINGLE_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxSINGLE_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetComplexSingles | mxSetSingles

**Introduced in R2018a**

# mxGetComplexSingles (C)

Complex data elements in mxSINGLE_CLASS array

## C Syntax

```
#include "matrix.h"
mxComplexSingle *mxGetComplexSingles(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxSINGLE_CLASS array.

## Output Arguments

**dt — Data array**
mxComplexSingle *

Pointer to the first mxComplexSingle element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxSINGLE_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxSINGLE_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetSingles | mxSetComplexSingles

**Introduced in R2018a**

# mxSetSingles (C)

Set real data elements in mxSINGLE_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetSingles(mxArray *pa, mxSingle *dt);
```

## Description

Use mxSetSingles to set mxSingle data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use the function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxSINGLE_CLASS array.

**dt — Data array**
mxSingle *

Pointer to the first mxSingle element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxSINGLE_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing data into an mxArray. The data in the example is defined as mxDouble. To modify this example for single data:

- Declare the data variables as `mxSingle`
- Call `mxCreateNumericMatrix` with the numeric type `mxSINGLE_CLASS`
- Replace `mxSetDoubles` with `mxSetSingles` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetSingles` | `mxSetComplexSingles`

**Introduced in R2018a**

# mxSetComplexSingles (C)

Set complex data elements in mxSINGLE_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetComplexSingles(mxArray *pa, mxComplexSingle *dt);
```

## Description

Use mxSetComplexSingles to set mxComplexSingle data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxSINGLE_CLASS array.

**dt — Data array**
mxComplexSingle *

Pointer to the first mxComplexSingle element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxSINGLE_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetComplexPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing complex numeric data into an mxArray. The data in the

example is defined as `mxComplexDouble`. You can use this example as a pattern for any complex C numeric type. To modify this example for complex `single` data:

- Declare the data variables as `mxComplexSingle`
- Call `mxCreateNumericMatrix` with the numeric type `mxSINGLE_CLASS`
- Replace `mxSetDoubles` with `mxSetComplexSingles` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetComplexSingles` | `mxSetSingles`

**Introduced in R2018a**

# mxGetUint16s (C)

Real data elements in mxUINT16_CLASS array

## C Syntax

```
#include "matrix.h"
mxUint16 *mxGetUint16s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxUINT16_CLASS array.

## Output Arguments

**dt — Data array**
mxUint16 *

Pointer to the first mxUint16 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxUINT16_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxUINT16_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetComplexUint16s | mxGetInt16s | mxSetUint16s

**Introduced in R2018a**

# mxGetComplexUint16s (C)

Complex data elements in mxUINT16_CLASS array

## C Syntax

```
#include "matrix.h"
mxComplexUint16 *mxGetComplexUint16s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxUINT16_CLASS array.

## Output Arguments

**dt — Data array**
mxComplexUint16 *

Pointer to the first mxComplexUint16 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxUINT16_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxUINT16_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetUint16s | mxSetComplexUint16s

**Introduced in R2018a**

# mxSetUint16s (C)

Set real data elements in mxUINT16_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetUint16s(mxArray *pa, mxUint16 *dt);
```

## Description

Use mxSetUint16s to set mxUint16 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxUINT16_CLASS array.

**dt — Data array**
mxUint16 *

Pointer to the first mxUint16 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT16_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing data into an mxArray. The data in the example is defined as mxDouble. To modify this example for uint16 data:

- Declare the data variables as `mxUint16`
- Call `mxCreateNumericMatrix` with the numeric type `mxUINT16_CLASS`
- Replace `mxSetDoubles` with `mxSetUint16s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetUint16s` | `mxSetComplexUint16s` | `mxSetInt16s`

**Introduced in R2018a**

# mxSetComplexUint16s (C)

Set complex data elements in mxUINT16_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetComplexUint16s(mxArray *pa, mxComplexUint16 *dt);
```

## Description

Use mxSetComplexUint16s to set mxComplexUint16 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxUINT16_CLASS array.

**dt — Data array**
mxComplexUint16 *

Pointer to the first mxComplexUint16 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT16_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetComplexPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing complex numeric data into an mxArray. The data in the

example is defined as `mxComplexDouble`. You can use this example as a pattern for any complex C numeric type. To modify this example for complex `uint16` data:

- Declare the data variables as `mxComplexUint16`
- Call `mxCreateNumericMatrix` with the numeric type `mxUINT16_CLASS`
- Replace `mxSetDoubles` with `mxSetComplexUint16s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetComplexUint16s` | `mxSetUint16s`

**Introduced in R2018a**

# mxGetUint32s (C)

Real data elements in mxUINT32_CLASS array

## C Syntax

```
#include "matrix.h"
mxUint32 *mxGetUint32s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxUINT32_CLASS array.

## Output Arguments

**dt — Data array**
mxUint32 *

Pointer to the first mxUint32 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxUINT32_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxUINT32_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetComplexUint32s | mxGetInt32s | mxSetUint32s

**Introduced in R2018a**

# mxGetComplexUint32s (C)

Complex data elements in mxUINT32_CLASS array

## C Syntax

```
#include "matrix.h"
mxComplexUint32 *mxGetComplexUint32s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxUINT32_CLASS array.

## Output Arguments

**dt — Data array**
mxComplexUint32 *

Pointer to the first mxComplexUint32 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxUINT32_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxUINT32_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetUint32s | mxSetComplexUint32s

**Introduced in R2018a**

# mxSetUint32s (C)

Set real data elements in mxUINT32_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetUint32s(mxArray *pa, mxUint32 *dt);
```

## Description

Use mxSetUint32s to set mxUint32 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxUINT32_CLASS array.

**dt — Data array**
mxUint32 *

Pointer to the first mxUint32 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT32_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing data into an mxArray. The data in the example is defined as mxDouble. To modify this example for uint32 data:

- Declare the data variables as `mxUint32`
- Call `mxCreateNumericMatrix` with the numeric type `mxUINT32_CLASS`
- Replace `mxSetDoubles` with `mxSetUint32s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetUint32s` | `mxSetComplexUint32s` | `mxSetInt32s`

**Introduced in R2018a**

# mxSetComplexUint32s (C)

Set complex data elements in mxUINT32_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetComplexUint32s(mxArray *pa, mxComplexUint32 *dt);
```

## Description

Use mxSetComplexUint32s to set mxUint32 data of the specified mxArray.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call this function to replace the existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxUINT32_CLASS array.

**dt — Data array**
mxComplexUint32 *

Pointer to the first mxComplexUint32 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT32_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetComplexPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing complex numeric data into an mxArray. The data in the

example is defined as `mxComplexDouble`. You can use this example as a pattern for any complex C numeric type. To modify this example for complex `uint32` data:

- Declare the data variables as `mxComplexUint32`
- Call `mxCreateNumericMatrix` with the numeric type `mxUINT32_CLASS`
- Replace `mxSetDoubles` with `mxSetComplexUint32s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetComplexUint32s` | `mxSetUint32s`

**Introduced in R2018a**

# mxGetUint64s (C)

Real data elements in mxUINT64_CLASS array

## C Syntax

```
#include "matrix.h"
mxUint64 *mxGetUint64s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxUINT64_CLASS array.

## Output Arguments

**dt — Data array**
mxUint64 *

Pointer to the first mxUint64 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxUINT64_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxUINT64_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetComplexUint64s | mxGetInt64s | mxSetUint64s

**Introduced in R2018a**

# mxGetComplexUint64s (C)

Complex data elements in `mxUINT64_CLASS` array

## C Syntax

```
#include "matrix.h"
mxComplexUint64 *mxGetComplexUint64s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
`const mxArray *`

Pointer to an `mxUINT64_CLASS` array.

## Output Arguments

**dt — Data array**
`mxComplexUint64 *`

Pointer to the first `mxComplexUint64` element of the data. If `pa` is `NULL`, then the function returns `NULL`.

If `mxArray` is not an `mxUINT64_CLASS` array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `NULL`. A `NULL` return value indicates that `pa` is either empty or not an `mxUINT64_CLASS` array.

## Examples

See these examples in *`matlabroot`*`/extern/examples/mex`:

- `explore.c`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetUint64s` | `mxSetComplexUint64s`

**Introduced in R2018a**

# mxSetUint64s (C)

Set real data elements in mxUINT64_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetUint64s(mxArray *pa, mxUint64 *dt);
```

## Description

Use mxSetUint64s to set mxUint64 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxUINT64_CLASS array.

**dt — Data array**
mxUint64 *

Pointer to the first mxUint64 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT64_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing data into an mxArray. The data in the example is defined as mxDouble. To modify this example for uint64 data:

- Declare the data variables as `mxUint64`
- Call `mxCreateNumericMatrix` with the numeric type `mxUINT64_CLASS`
- Replace `mxSetDoubles` with `mxSetUint64s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetUint64s` | `mxSetComplexUint64s` | `mxSetInt64s`

**Introduced in R2018a**

# mxSetComplexUint64s (C)

Set complex data elements in mxUINT64_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetComplexUint64s(mxArray *pa, mxComplexUint64 *dt);
```

## Description

Use mxSetComplexUint64s to set complex, mxComplexUint64 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxUINT64_CLASS array.

**dt — Data array**
mxComplexUint64 *

Pointer to the first mxComplexUint64 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT64_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetComplexPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing complex numeric data into an mxArray. The data in the

example is defined as `mxComplexDouble`. You can use this example as a pattern for any complex C numeric type. To modify this example for complex `uint64` data:

- Declare the data variables as `mxComplexUint64`
- Call `mxCreateNumericMatrix` with the numeric type `mxUINT64_CLASS`
- Replace `mxSetDoubles` with `mxSetComplexUint64s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetComplexUint64s` | `mxSetUint64s`

**Introduced in R2018a**

# mxGetUint8s (C)

Real data elements in mxUINT8_CLASS array

## C Syntax

```
#include "matrix.h"
mxUint8 *mxGetUint8s(const mxArray *pa);
```

## Input Arguments

### pa — MATLAB array

const mxArray *

Pointer to an mxUINT8_CLASS array.

## Output Arguments

### dt — Data array

mxUint8 *

Pointer to the first mxUint8 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxUINT8_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxUINT8_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetComplexUint8s | mxGetInt8s | mxSetUint8s

**Introduced in R2018a**

# mxGetComplexUint8s (C)

Complex data elements in mxUINT8_CLASS array

## C Syntax

```
#include "matrix.h"
mxComplexUint8 *mxGetComplexUint8s(const mxArray *pa);
```

## Input Arguments

**pa — MATLAB array**
const mxArray *

Pointer to an mxUINT8_CLASS array.

## Output Arguments

**dt — Data array**
mxComplexUint8 *

Pointer to the first mxComplexUint8 element of the data. If pa is NULL, then the function returns NULL.

If mxArray is not an mxUINT8_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns NULL. A NULL return value indicates that pa is either empty or not an mxUINT8_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/mex:

- explore.c

## API Version

This function is available in the interleaved complex API. To build myMexFile.c using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

mxGetUint8s | mxSetComplexUint8s

**Introduced in R2018a**

# mxSetUint8s (C)

Set real data elements in mxUINT8_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetUint8s(mxArray *pa, mxUint8 *dt);
```

## Description

Use mxSetUint8s to set mxUint8 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxUINT8_CLASS array.

**dt — Data array**
mxUint8 *

Pointer to the first mxUint8 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT8_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing data into an mxArray. The data in the example is defined as mxDouble. To modify this example for uint8 data:

- Declare the data variables as `mxUint8`
- Call `mxCreateNumericMatrix` with the numeric type `mxUINT8_CLASS`
- Replace `mxSetDoubles` with `mxSetUint8s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetUint8s` | `mxSetComplexUint8s` | `mxSetInt8s`

**Introduced in R2018a**

# mxSetComplexUint8s (C)

Set complex data elements in mxUINT8_CLASS array

## C Syntax

```
#include "matrix.h"
int mxSetComplexUint8s(mxArray *pa, mxComplexUint8 *dt);
```

## Description

Use mxSetComplexUint8s to set mxComplexUint8 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mxArray *

Pointer to an mxUINT8_CLASS array.

**dt — Data array**
mxComplexUint8 *

Pointer to the first mxComplexUint8 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
int

Function status, returned as int. If successful, then the function returns 1.

If pa is NULL, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT8_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## Examples

Refer to the arrayFillSetComplexPr.c example in the *matlabroot*/extern/examples/refbook folder which copies existing complex numeric data into an mxArray. The data in the

example is defined as `mxComplexDouble`. You can use this example as a pattern for any complex C numeric type. To modify this example for complex `uint8` data:

- Declare the data variables as `mxComplexUint8`
- Call `mxCreateNumericMatrix` with the numeric type `mxUINT8_CLASS`
- Replace `mxSetDoubles` with `mxSetComplexUint8s` to put the C array into an `mxArray`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.c` using this function, type:

```
mex -R2018a myMexFile.c
```

## See Also

`mxGetComplexUint8s` | `mxSetUint8s`

**Introduced in R2018a**

# mxGetDoubles (Fortran)

Real data elements in mxDOUBLE_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetDoubles(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxDOUBLE_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer | 0

Pointer to the first mxDouble element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxDOUBLE_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxDOUBLE_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetDoubles (Fortran)

**Introduced in R2018b**

# mxGetComplexDoubles (Fortran)

Complex data elements in `mxDOUBLE_CLASS` array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetComplexDoubles(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
`mwPointer` | `0`

Pointer to an `mxDOUBLE_CLASS` array.

## Output Arguments

**dt — Data array**
`mwPointer`

Pointer to the first `mxComplexDouble` element of the data. If `pa` is `0`, then the function returns `0`.

If `mxArray` is not an `mxDOUBLE_CLASS` array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`. A `0` return value indicates that `pa` is either empty or not an `mxDOUBLE_CLASS` array.

## Examples

See these examples in *`matlabroot`*`/extern/examples/refbook`:

- `convec.F`
- `complexAdd.F`

## API Version

This function is available in the interleaved complex API. To build `myMexFile.F` using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxSetComplexDoubles (Fortran)`

**Introduced in R2018b**

# mxSetDoubles (Fortran)

Set real data elements in mxDOUBLE_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetDoubles(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetDoubles to set mxDouble data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxDOUBLE_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxDouble element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxDOUBLE_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also
`mxGetDoubles (Fortran)`

**Introduced in R2018b**

# mxSetComplexDoubles (Fortran)

Set complex data elements in mxDOUBLE_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetComplexDoubles(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetComplexDoubles to set mxComplexDouble data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxDOUBLE_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxComplexDouble element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxDOUBLE_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxGetComplexDoubles (Fortran)`

**Introduced in R2018b**

# mxGetInt16s (Fortran)

Real data elements in mxINT16_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetInt16s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT16_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer

Pointer to the first mxInt16 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxINT16_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxINT16_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetInt16s (Fortran)

**Introduced in R2018b**

# mxGetComplexInt16s (Fortran)

Complex data elements in `mxINT16_CLASS` array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetComplexInt16s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
`mwPointer`

Pointer to an `mxINT16_CLASS` array.

## Output Arguments

**dt — Data array**
`mwPointer | 0`

Pointer to the first `mxComplexInt16` element of the data. If `pa` is `0`, then the function returns `0`.

If `mxArray` is not an `mxINT16_CLASS` array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`. A `0` return value indicates that `pa` is either empty or not an `mxINT16_CLASS` array.

## API Version

This function is available in the interleaved complex API. To build `myMexFile.F` using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxSetComplexInt16s (Fortran)`

**Introduced in R2018b**

# mxSetInt16s (Fortran)

Set real data elements in mxINT16_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetInt16s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetInt16s to set mxInt16 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT16_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxInt16 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT16_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxGetInt16s (Fortran)`

**Introduced in R2018b**

# mxSetComplexInt16s (Fortran)

Set complex data elements in mxINT16_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetComplexInt16s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetComplexInt16s to set mxComplexInt16 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT16_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxComplexInt16 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT16_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxGetComplexInt16s (Fortran)`

**Introduced in R2018b**

# mxGetInt32s (Fortran)

Real data elements in mxINT32_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetInt32s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT32_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer

Pointer to the first mxInt32 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxINT32_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxINT32_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetInt32s (Fortran)

**Introduced in R2018b**

# mxGetComplexInt32s (Fortran)

Complex data elements in mxINT32_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetComplexInt32s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT32_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer | 0

Pointer to the first mxComplexInt32 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxINT32_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxINT32_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetComplexInt32s (Fortran)

**Introduced in R2018b**

# mxSetInt32s (Fortran)

Set real data elements in mxINT32_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetInt32s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetInt32s to set mxInt32 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT32_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxInt32 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT32_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also
mxGetInt32s (Fortran)

**Introduced in R2018b**

# mxSetComplexInt32s (Fortran)

Set complex data elements in mxINT32_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetComplexInt32s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetComplexInt32s to set mxComplexInt32 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT32_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxComplexInt32 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT32_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxGetComplexInt32s (Fortran)`

**Introduced in R2018b**

# mxGetInt64s (Fortran)

Real data elements in mxINT64_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetInt64s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT64_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer

Pointer to the first mxInt64 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxINT64_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxINT64_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetInt64s (Fortran)

**Introduced in R2018b**

# mxGetComplexInt64s (Fortran)

Complex data elements in `mxINT64_CLASS` array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetComplexInt64s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
`mwPointer`

Pointer to an `mxINT64_CLASS` array.

## Output Arguments

**dt — Data array**
`mwPointer | 0`

Pointer to the first `mxComplexInt64` element of the data. If `pa` is `0`, then the function returns `0`.

If `mxArray` is not an `mxINT64_CLASS` array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`. A `0` return value indicates that `pa` is either empty or not an `mxINT64_CLASS` array.

## API Version

This function is available in the interleaved complex API. To build `myMexFile.F` using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxSetComplexInt64s (Fortran)`

**Introduced in R2018b**

# mxSetInt64s (Fortran)

Set data elements in mxINT64_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetInt64s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetInt64s to set mxInt64 data of the specified mxArray.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT64_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxInt64 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT64_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also
`mxGetInt64s (Fortran)`

**Introduced in R2018b**

# mxSetComplexInt64s (Fortran)

Set complex data elements in mxINT64_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetComplexInt64s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetComplexInt64s to set mxComplexInt64 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT64_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxComplexInt64 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT64_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also
`mxGetComplexInt64s (Fortran)`

**Introduced in R2018b**

# mxGetInt8s (Fortran)

Real data elements in mxINT8_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetInt8s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT8_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer

Pointer to the first mxInt8 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxINT8_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxINT8_CLASS array.

## Examples

See these examples in *matlabroot*/extern/examples/refbook:

- matsqint8.F

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetInt8s (Fortran)

**Introduced in R2018b**

# mxGetComplexInt8s (Fortran)

Complex data elements in mxINT8_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetComplexInt8s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT8_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer | 0

Pointer to the first mxComplexInt8 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxINT8_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxINT8_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetComplexInt8s (Fortran)

**Introduced in R2018b**

# mxSetInt8s (Fortran)

Set real data elements in mxINT8_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetInt8s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetInt8s to set mxInt8 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT8_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxInt8 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT8_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also
mxGetInt8s (Fortran)

**Introduced in R2018b**

# mxSetComplexInt8s (Fortran)

Set complex data elements in mxINT8_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetComplexInt8s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetComplexInt8s to set mxComplexInt8 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxINT8_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxComplexInt8 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxINT8_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxGetComplexInt8s (Fortran)`

**Introduced in R2018b**

# mxGetSingles (Fortran)

Real data elements in mxSINGLE_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetSingles(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxSINGLE_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer | 0

Pointer to the first mxSingle element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxSINGLE_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxSINGLE_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetSingles (Fortran)

**Introduced in R2018b**

# mxGetComplexSingles (Fortran)

Complex data elements in `mxSINGLE_CLASS` array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetComplexSingles(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
`mwPointer`

Pointer to an `mxSINGLE_CLASS` array.

## Output Arguments

**dt — Data array**
`mwPointer | 0`

Pointer to the first `mxComplexSingle` element of the data. If `pa` is `0`, then the function returns `0`.

If `mxArray` is not an `mxSINGLE_CLASS` array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`. A `0` return value indicates that `pa` is either empty or not an `mxSINGLE_CLASS` array.

## API Version

This function is available in the interleaved complex API. To build `myMexFile.F` using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxSetComplexSingles (Fortran)`

**Introduced in R2018b**

# mxSetSingles (Fortran)

Set real data elements in mxSINGLE_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetSingles(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetSingles to set mxSingle data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use the function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxSINGLE_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxSingle element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxSINGLE_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also
mxGetSingles (Fortran)

**Introduced in R2018b**

# mxSetComplexSingles (Fortran)

Set complex data elements in mxSINGLE_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetComplexSingles(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetComplexSingles to set mxComplexSingle data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxSINGLE_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxComplexSingle element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxSINGLE_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxGetComplexSingles (Fortran)`

**Introduced in R2018b**

# mxGetUint16s (Fortran)

Real data elements in mxUINT16_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetUint16s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT16_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer

Pointer to the first mxUint16 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxUINT16_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxUINT16_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetUint16s

**Introduced in R2018b**

# mxGetComplexUint16s (Fortran)

Complex data elements in mxUINT16_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetComplexUint16s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT16_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer | 0

Pointer to the first mxComplexUint16 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxUINT16_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxUINT16_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also
mxSetComplexUint16s (Fortran)

**Introduced in R2018b**

# mxSetUint16s (Fortran)

Set real data elements in mxUINT16_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetUint16s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetUint16s to set mxUint16 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT16_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxUint16 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT16_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also
`mxGetUint16s (Fortran)`

**Introduced in R2018b**

# mxSetComplexUint16s (Fortran)

Set complex data elements in mxUINT16_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetComplexUint16s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetComplexUint16s to set mxComplexUint16 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT16_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxComplexUint16 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT16_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also
`mxGetComplexUint16s (Fortran)`

**Introduced in R2018b**

# mxGetUint32s (Fortran)

Real data elements in mxUINT32_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetUint32s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT32_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer

Pointer to the first mxUint32 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxUINT32_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxUINT32_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxGetComplexUint32s | mxSetUint32s

**Introduced in R2018b**

# mxGetComplexUint32s (Fortran)

Complex data elements in `mxUINT32_CLASS` array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetComplexUint32s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
`mwPointer`

Pointer to an `mxUINT32_CLASS` array.

## Output Arguments

**dt — Data array**
`mwPointer` | `0`

Pointer to the first `mxComplexUint32` element of the data. If `pa` is `0`, then the function returns `0`.

If `mxArray` is not an `mxUINT32_CLASS` array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`. A `0` return value indicates that `pa` is either empty or not an `mxUINT32_CLASS` array.

## API Version

This function is available in the interleaved complex API. To build `myMexFile.F` using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxSetComplexUint32s (Fortran)`

**Introduced in R2018b**

# mxSetUint32s (Fortran)

Set real data elements in mxUINT32_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetUint32s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetUint32s to set mxUint32 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT32_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxUint32 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT32_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxGetUint32s (Fortran)

**Introduced in R2018b**

# mxSetComplexUint32s (Fortran)

Set complex data elements in `mxUINT32_CLASS` array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetComplexUint32s(pa, dt)
mwPointer pa, dt
```

## Description

Use `mxSetComplexUint32s` to set `mxUint32` data of the specified `mxArray`.

All `mxCreate*` functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call this function to replace the existing values with new values.

## Input Arguments

**pa — MATLAB array**
`mwPointer`

Pointer to an `mxUINT32_CLASS` array.

**`dt` — Data array**
`mwPointer`

Pointer to the first `mxComplexUint32` element of the data array. `dt` must be allocated by the functions `mxCalloc` or `mxMalloc`.

## Output Arguments

**`status` — Function status**
`integer*4`

Function status, returned as `integer*4`. If successful, then the function returns `1`.

If `pa` is `0`, then the function returns `0`.

The function is unsuccessful when `mxArray` is not an unshared `mxUINT32_CLASS` array, or if the data is not allocated with `mxCalloc`. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns `0`.

## API Version

This function is available in the interleaved complex API. To build `myMexFile.F` using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxGetComplexUint32s (Fortran)`

**Introduced in R2018b**

# mxGetUint64s (Fortran)

Real data elements in mxUINT64_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetUint64s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT64_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer

Pointer to the first mxUint64 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxUINT64_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxUINT64_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetUint64s

**Introduced in R2018b**

# mxGetComplexUint64s (Fortran)

Complex data elements in mxUINT64_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetComplexUint64s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT64_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer | 0

Pointer to the first mxComplexUint64 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxUINT64_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxUINT64_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetComplexUint64s (Fortran)

**Introduced in R2018b**

# mxSetUint64s (Fortran)

Set real data elements in mxUINT64_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetUint64s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetUint64s to set mxUint64 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT64_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxUint64 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT64_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxGetUint64s (Fortran)

**Introduced in R2018b**

# mxSetComplexUint64s (Fortran)

Set complex data elements in mxUINT64_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetComplexUint64s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetComplexUint64s to set complex, mxComplexUint64 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT64_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxComplexUint64 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT64_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxGetComplexUint64s (Fortran)`

**Introduced in R2018b**

# mxGetUint8s (Fortran)

Real data elements in mxUINT8_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetUint8s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT8_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer

Pointer to the first mxUint8 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxUINT8_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxUINT8_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetUint8s

**Introduced in R2018b**

# mxGetComplexUint8s (Fortran)

Complex data elements in mxUINT8_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
mwPointer mxGetComplexUint8s(pa)
mwPointer pa
```

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT8_CLASS array.

## Output Arguments

**dt — Data array**
mwPointer | 0

Pointer to the first mxComplexUint8 element of the data. If pa is 0, then the function returns 0.

If mxArray is not an mxUINT8_CLASS array:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0. A 0 return value indicates that pa is either empty or not an mxUINT8_CLASS array.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxSetComplexUint8s (Fortran)

**Introduced in R2018b**

# mxSetUint8s (Fortran)

Set real data elements in mxUINT8_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetUint8s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetUint8s to set mxUint8 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT8_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxUint8 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT8_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

mxGetUint8s (Fortran)

**Introduced in R2018b**

# mxSetComplexUint8s (Fortran)

Set complex data elements in mxUINT8_CLASS array

## Fortran Syntax

```
#include "fintrf.h"
integer*4 mxSetComplexUint8s(pa, dt)
mwPointer pa, dt
```

## Description

Use mxSetComplexUint8s to set mxComplexUint8 data in the specified array.

All mxCreate* functions allocate heap space to hold data. Therefore, you do not ordinarily use this function to initialize the elements of an array. Rather, call the function to replace existing values with new values.

## Input Arguments

**pa — MATLAB array**
mwPointer

Pointer to an mxUINT8_CLASS array.

**dt — Data array**
mwPointer

Pointer to the first mxComplexUint8 element of the data array. dt must be allocated by the functions mxCalloc or mxMalloc.

## Output Arguments

**status — Function status**
integer*4

Function status, returned as integer*4. If successful, then the function returns 1.

If pa is 0, then the function returns 0.

The function is unsuccessful when mxArray is not an unshared mxUINT8_CLASS array, or if the data is not allocated with mxCalloc. If the function is unsuccessful, then:

- MEX file — Function terminates the MEX file and returns control to the MATLAB prompt.
- Standalone (non-MEX file) application — Function returns 0.

## API Version

This function is available in the interleaved complex API. To build myMexFile.F using this function, type:

```
mex -R2018a myMexFile.F
```

## See Also

`mxGetComplexUint8s (Fortran)`

**Introduced in R2018b**