Matrix Module

Table of Contents

1 Complex Numbers 2

Overview 2

Available Functions 2

Class Implementation 3

Figures & Tables 3

Data Structure 33

Algorithms 33

2 Column & Row Vectors 34

Overview 34

Available Vector Functions 34

3 Class Matrix 47

Overview 47

Algorithms 83

Diagonalization 83

Matrix Types 84

Matrix Hermitian Type 84

1 Complex Numbers

1.1 Overview

The class ***complex*** generally defines complex numbers for C++, including all the usual operations on them. External users and most functions do not rely on the way complex numbers are stored. To use this class the file ***complex.h*** must be included. This is automatically done if ***gamma.h*** is included.

1.2 Available Functions

Basic Functions

complex	- Constructor	complex(), complex(z), complex(r,i)	page 4
+	- Addition	z3 = z1 + z2, z2 = z1 + r	page 4
+=	- Unary Addition	z1 += z2	page 5
-	- Subtraction, Negation	z3 = z1-z2, -z	page 6
-=	- Unary Subtraction	z1 -= z2	page 6
*	- Multiplication	z3 = z1*z2	page 7
*=	- Unary Multiplication	z1 *= z2	page 7
/	- Division	z3 = z1/z2	page 8
/=	- Unary Division	z1 /= z2	page 9
=	- Assignment	z1 = z2	page 9
==	- Equality	z1 == z2	page 10
!=	- Inequality	z1 != z2	page 10

Input/Output

<<	- Complex Output	cout << z	page 12
>>	- Complex Input	cin >> z	page 12
complex_getf	- Get Complex output format		page 13
complex_setf	- Set Complex output format		page 13

Standard Functions

Re	- Real part	page 15
Im	- Imaginary part	page 15
norm	- Complex magnitidue	page 16
phase	- Complex phase	page 17
conj	- Complex conjugate	page 18
Swap	- Switch two complex numbers	page 18

Trigonometric Functions

sin	- Sine	page 19
asin	- Arcsine	page 19

sinh	- Hyperbolic sine		page 20
asinh	- Hyperbolic arcsine		page 20
cos	- Cosine		page 21
acos	- Arccosine		page 22
cosh	- Hyperbolic cosine		page 22
acosh	- Hyperbolic arccosine		page 24
tan	- Complex tangent		page 24
atan	- Arc-tangent		page 25
tanh	- Hyperbolic tangent		page 25
atanh	- Hyperbolic arc-tangent		page 26

Transcendental Functions

exp	- Complex exponential		page 27
log	- Complex logarithm		page 27
pow	- Power function		page 28
square_norm	- Square of the norm		page 28
AbsNorm	- Absolute norm		page 29
sqrt	- Complex square root		page 29

Additional Functions

add	- Addition	z1 = z2+z3, z1=r+z2, z1=z2+r	page 31
sub	- Subtraction		page 31
mul	- Multiplication		page 31
div	- Division		page 32

1.3 Class Implementation

Data Structure	page 33
Algorithms	page 33

1.4 Figures & Tables

Table 1:	: Parameters Affecting The Output of Complex Numbers	page 13

1.5 Basic Functions

1.5.1 complex

Usage:

```
#include <complex.h>
complex()
complex(double a, double b=0.0)
complex(const complex& z)
```

Description:

The function ***complex*** is the standard constructor for complex numbers[1].

1. ***complex()*** - Constructs a complex number which is zero: z = 0 + i0
2. ***complex(double a, double b)*** - Constructs a complex number which has ***a*** as the real value and ***b*** as the imaginary value: z = ***a*** + i***b***. The argument ***b*** is optional and will be set to zero if not present.
3. ***complex(const complex& z1)*** - Constructs a complex number identical to that input: z = ***z1***.

Examples:

```
#include <gamma.h>
main()
 {
 complex z;                  // Declare an empty complex number
 complex z1(5);              // Declare complex number z1 = 5 + 0i
 complex z2(5,4);            // Declare complex number z2 = 5 + 4i
 complex z3(z2);             // Construct z3 = z2
 z1 = complex(3,2);          // Reset z1 to have the value 3+2i
 }
```

Return Value:

The created complex number.

1.5.2 +

Usage:

```
#include <complex.h>
complex operator + (const complex& z)
complex operator + (const complex& z1, const complex& z2)
complex operator + (const complex& z, double x)
complex operator + (double a, const complex& z)
```

Description:

The ***operator +*** adds two complex numbers. Overloaded forms allow additions mixing complex numbers and dou-

1. These constructors reflect the fact that the current implementation of class ***complex*** internally represents complex numbers as $z = a + ib$. However, one can set a complex number according to $z = Re^{i\varphi}$ through use the functions ***norm*** and ***phase***.

ble precision numbers. The first function form does nothing, it is the counterpart to the negation function (see ***operator*** -).

Examples:

```
#include <gamma.h>
main()
  {
   complex z1, z2;                     // Declare two complex numbers
   double x;                           // Declare a double
   z1 = z2+ z2;                        // Set z1 to the sum of z2 with itself
   z1 = x + z2;                        // Set z1 to the sum of x and z2
   }
```

Return Value:

A complex number.

See Also:

-, -=, +=, add

1.5.3 +=

Usage:

```
#include <complex.h>
void operator += (complex& z1, const complex& z2)
void operator += (complex& z, double x)
```

Description:

The ***unary addition operator* +=** adds a either a complex number or a double precision number to another complex number. Note that ***z1 += z2*** is faster than ***z1 = z1 + z2***!

Note: Expressions like ***c=(a+=1)*** are not allowed for complex numbers.

Examples:

```
include <gamma.h>
main()
  {
   complex z1, z2;                     // Declare two complex numbers
   double x;                           // Declare a double precision number
   z1 += z2;                           // Add z2 directly to z1
   z1 += x;                            // Add x directly to z1
   }
```

Return Value:

Nothing, alters the first complex value.

See Also:

-, -=, +

1.5.4 -

Usage:

```
#include <complex.h>
complex operator - (const complex& z)
complex operator - (const complex& z1, const complex& z2)
complex operator - (const complex& z, double x)
complex operator - (double a, const complex& z)
```

Description:

The ***operator -*** subtracts two complex numbers. It is also used when a subtraction blends double precision and complex numbers. The unitary operator - negates when only a single complex number is involved.

Examples:

```
#include <gamma.h>
main()
 {
 complex z1, z2;                    // Construct two complex numbers
 double x;                          // Construct a double
 z1 = z2 - z2;                      // Set z1 to the difference of z2 with itself (0)
 z1= x - z2;                        // Set z1 to the difference of x and z2
 z2 = -z1;                          // Set z2 to the negated z1
 }
```

Return Value:

A complex number. Either the difference between the arguments or the negated argument.

See Also:

+, -=, +=

1.5.5 -=

Usage:

```
#include <complex.h>
void operator -= (complex& z1, const complex& z2)
void operator -= (complex& z, double& x)
```

Description:

The ***unary operator -=*** subtracts either a complex number or a double precision number from a complex number. Use of ***z1 -= z2*** is faster than ***z1 = z1 - z2***!

Note: Expressions like ***c = (a -=1)*** are not allowed for complex numbers

Examples:

```
include <gamma.h>
main()
 {
 complex z1, z2;                    // Declare two complex numbers
 double x;                          // Declare a double precision number
 z1 -= z2;                          // Subtract z2 from z1
```

```
    z1 -= x;                        // Subtract x from z1
    // x -= z1;                     // This is an error!
    }
```

Return Value:

Returns nothing.

See Also:

-, +=, +

1.5.6 *

Usage:

```
#include <complex.h>
complex operator * (const complex& z1, const complex& z2)
complex operator * (const complex& z, double x)
complex operator * (double x, const complex& z)
```

Description:

The ***operator**** multiplies two complex numbers or a double and a complex number.

Examples:

```
include <gamma.h>
main()
 {
  complex z1, z2, z3;               // Declare three complex numbers
  double x;                         // Declare a double
  z1 = z2 * z3;                     // Set z1 to the product of z2 and z3
  z1 = z2 * x;                      // Set z1 to the product of z2 and x
  }
```

Return Value:

Returns a complex number, the product of the arguments.

See Also:

/, *=, /=

1.5.7 *=

Usage:

```
#include <complex.h>
void operator *= (complex& z1, const complex& z2)
void operator *= (complex& z, double& x)
```

Description:

The ***unary operator *=*** multiplies either another complex number or a double into an existing complex number. Use of ***z1 *= z2*** is faster than ***z1 = z1 * 2***!

Note: Expressions like ***c=(a*=2)*** are not allowed for complex numbers

Examples:

```
#include <gamma.h>
main()
  {
   complex z1, z2;                    // Declare two complex numbers
   double x;                          // Declare a double
   z1 *= z2;                          // The same as z1 = z1 * z2, but faster
   z1 *= x;                           // The same as z1 = x * z2, but faster
   }
```

Return Value:

Nothing.

See Also:

*, /=, /

1.5.8 /

Usage:

```
#include <complex.h>
complex operator / (const complex& z1, const complex& z2)
complex operator / (const complex& z, double x)
complex operator / (double x, const complex& z)
```

Description:

The binary ***operator /*** divides two complex numbers or performs division involving both complex and real numbers.

Example:

```
#include <gamma.h>
main()
  {
   complex x, y;
   double z;
   x = y / y;
   x= z / y;
   }
```

Return Value:

The binary operator / returns the quotient of the arguments.

See Also:

*, *=, /=

1.5.9 /=

Usage:

```
#include <complex.h>
void operator /= (complex& z1, const complex& z2)
void operator /= (complex& z, double& x)
```

Description:

The ***operator /=*** divides a complex number by either another complex number or a double. Use of ***z1 /= z2*** is faster than ***z1 = z1/z2***!

Example:

```
#include <gamma.h>
main()
  {
   complex x, y;
   double z;
   x /= y;                    // The same as x = x/y, but faster
   x /= z;                    // The same as x = x/z, but faster
   }
```

Return Value:

returns nothing (Expressions like c=(a/=2) are not allowed for complex numbers).

See Also:

*, *=, /

1.5.10 =

Usage:

```
#include <complex.h>
complex& operator = (const complex &z)
complex& operator = (double x)
```

Description:

The ***operator =*** assigns either a real or complex number to a complex variable. Note that is is possible to write code such as ***a = (b = c).***

Example:

```
#include <gamma.h>
main()
  {
   complex z1, z2;            // Declare two complex numbers
   double x;                  // Declare a double
   z1 = z2;                   // Set z1 to z2
   z1 = x;                    // Set z1 to x + 0i
   }
```

Return Value:

Returns a complex number.

See Also:

+=, -=, *=, /=

1.5.11 ==

Usage:

```
#include <complex.h>
int operator == (const complex& z1, const complex& z2)
```

Description:

The ***operator ==*** compares the two complex numbers in the argument list ***z1*** & ***z2***. If they are identical the return is true (non-zero) and if they are not equal the return is false(zero).

Example:

```
#include <gamma.h>
main()
  {
  complex z1, z2;
  if (z1 == z2)
    cout << “z1 and z2 are equal! \n”;
  else
    cout << “z1 and z2 are different! \n”;
  }
```

Return Value:

Returns true if the arguments are equal.

See Also:

!=

1.5.12 !=

Usage:

```
#include <complex.h>
int operator!= (const complex& z1, const complex& z2)
```

Description:

The ***operator!=*** compares the two complex numbers in the argument list ***z1*** & ***z2***. If they are identical the return is false (zero) and if they are not equal the return is true (non-zero).

Example:

```
#include <gamma.h>
main()
 {
   complex z1, z2;                           // Declare two complex numbers
   if (z1 != z2)                             // Compare the two numbers
     cout << “z1 and z2 are not equal! \n”;// Write the result to standard output
   else
     cout << “z1 and z2 are equal!\n”;
   }
```

Return Value:

Returns true if the arguments are not equivalent.

See Also:

==

1.6 Input/Output

1.6.1 <<

Usage:

```
#include <complex.h>
ostream& operator << (ostream& ostr, const complex& z)
```

Description:

The ***operator*** << is the standard output for printing a complex number ***z***. The format of the output may be set with the function ***complex_setf***.

Example:

```
#include <gamma.h>
main()
  {
  complex z;                          // Declare a complex number
  cout << z;                          // Print the complex number to standard output.
  }
```

Return Value:

Returns a modified output stream.

See Also:

>>

1.6.2 >>

Usage:

```
#include <complex.h>
istream& operator >> (istream& istr, complex & z)
```

Description:

The standard input ***operator*** >> reads the complex number ***z*** from the input stream ***istr***.

Example:

```
#include <gamma.h>
main()
 {
  complex z;                          // Declare a complex number
  cout << "\n\tPlease input z: ";     // Ask for the complex number
  cin >> z;                           // Read the complex number from standard input
  }
```

Return Value:

Returns a modified input stream.

See Also:

<<

1.6.3 complex_getf

Usage:

```
#include <complex.h>
void complex_getf (int &phase, int &math, int &science, int &digits, int &digits_after_dpoint)
```

Description:

The function ***complex_getf*** returns the current output format for complex numbers. See the function ***complex_setf*** for a description of the arguments.

Return Value:

Nothing. The parameters supplied as arguments are altered.

See Also:

complex_setf

1.6.4 complex_setf

Usage:

```
#include <complex.h>
void complex_setf(int phase, int math, int science, int digits, int digits_after_dpoint)
```

Description:

Function ***complex_setf*** sets the output format for complex numbers. All complex numbers will be output according to the current settings until this function is re-invoked. Parameter meanings are shown in the following table.

Table 1: : Parameters Affecting The Output of Complex Numbers

Parameter	Type	Affect on Output		Default
		TRUE	FALSE	
phase	integer	Norm & Phase	Real & Imaginary	FALSE
math	integer	$a + ib$ or $Re^{i\varphi}$	(a, b) or $[R, \varphi]$	FALSE
science	integer	a, b, R, φ : x.xxxey	a, b, R, φ : xxxx.xxx	FALSE
digits	integer	Total number of digits		6
digits_after_point	integer	Digits after the decimal point[a]		2

a. A negative value is used to indicate no limit on the number of digits.

Return Value:

Nothing.

See Also:

complex_getf, <<

1.7 Standard Functions

1.7.1 Re

Usage:

```
#include <complex.h>
double Re (const complex &z)
void Re (complex& z, double x);
```

Description:

The function ***Re*** can be used to either obtain or set the real part of a complex number. With only a complex number ***z*** for an argument the function performs according to

$$Re(z) = Re(a + ib) = a .$$

With both a complex number ***z*** and a double ***x*** as arguements the real part of ***z*** is set to ***x***.

$$Re(z, x) = Re(a + ib, x) \rightarrow x + ib = z$$

The imaginary component of ***z*** remains unchanged.

Example:

```
#include <gamma.h>
main()
  {
  double d;
  complex z = complex (4, 1);          // z has the value 4+i
  d = Re (z);                          // d has the value 4
  Re(z,5);                             // z has now the value 4+5i
  }
```

Return Value:

Re returns the real part of the input value

See Also:

Im

1.7.2 Im

Usage:

```
#include <complex.h>
double Im (const complex & z)
void Im ( complex& z, double x);
```

Description:

The function ***Im*** can be used to either obtain or set the imaginary part of a complex number. With only a complex number ***z*** for an argument the function performs according to

$$Im(z) = Im(a + ib) = b.$$

With both a complex number ***z*** and a double ***x*** as arguements the imaginary part of ***z*** is set to ***x***.

$$Im(z, x) = Im(a + ib, x) \rightarrow a + ix = z$$

The real component of ***z*** remains unchanged.

Example:

```
#include <gamma.h>
main()
    {
    complex c;
    double d;
    c = complex (4, 1);                         // c has the value 4+i
    d = Im (c);                                 // d has the value 1
    Im(c,5);                                    // c has the value 4+5i
    }
```

Return Value:

Im returns the imaginary part of the input value or return nothing.

See Also:

Re

1.7.3 norm

Usage:

```
#include <complex.h>
double norm (const complex & z);
void norm (complex& z, double x);
```

Description:

The function ***norm*** is used to either obtain or set the magnitude of a complex number. Given a complex number ***z*** as the exclusive argument ***norm*** returns the magnitude of ***z*** as defined by

$$\mathrm{norm}(z) = \mathrm{norm}(a + ib) = \mathrm{norm}(Re^{i\varphi}) = \sqrt{a^2 + b^2} = R.$$

When a double ***x*** is also included in the argument list the complex number is scaled so that its norm equals ***x***, the phase remaining unchanged.

$$\mathrm{norm}(z, x) = \mathrm{norm}(Re^{i\varphi}, x) \rightarrow xe^{i\varphi} = z$$

Example:

```
#include <gamma.h>
main()
    {
  complex z(4,3);                          // Declare a complex nuber z=4+i3
  cout << norm(z);                         // Output the value of the norm, 5
  norm (z, 10);                            // Set the norm to 10, now z = 8+6i
```

```
}
```

Return Value:

Either double or void. The returned double will be the magnitude of the input value.

See Also:

phase

1.7.4 phase

Usage:

```
#include <complex.h>
double phase (const complex & z);
void phase (complex& z, double x);
```

Description:

The function ***phase*** is used to either obtain or set the phase of a complex number. When invoked with a single complex number as the argument, ***z***, phase returns it's phase φ as defined by

$$\mathrm{phase}(z) = \mathrm{phase}(a+ib) = phase(Re^{i\varphi}) = \varphi = \begin{cases} \mathrm{atan}\left(\frac{b}{a}\right) & \text{if} \quad a \geq 0 \\ \mathrm{atan}\left(\frac{b}{a}\right) + \pi & \text{if} \quad a < 0 \wedge b \geq 0 \\ \mathrm{atan}\left(\frac{b}{a}\right) - \pi & \text{if} \quad a < 0 \wedge b < 0 \end{cases}$$

When the function ***phase*** is invoked with both a complex number, ***z***, and a double ***x*** as arguments the phase of ***z*** is set to ***x***. The norm of ***z*** remains unchanged, and ***x*** must be input in ***radians***.

Note that the phase is always maintained within $-\pi < \varphi \leq \pi$. i.e. $\varphi \in (-\pi, \pi]$.

Example:

```
#include <gamma.h>
main()
  {
  complex z(4,3);                       // Declare a complex number z = 4+i3
  cout << phase(z);                     // Output the phase,
  phase(z, 5);                          // Set z to now the phase value of 5
  }
```

Return Value:

Either double or void. The returned double will be the phase of the input value.

See Also:

norm

1.7.5 conj

Usage:

```
#include <complex.h>
complex conj (const complex & z)
complex conj (const complex & z1, const complex& z2)
```

Description:

The function ***conj***, with one complex number ***z*** as an argument, returns its complex conjugate.

$$conj(z) = z^*$$

With two complex numbers as arguments, ***conj*** returns the product of the conjugate of the first with the second.

$$conj(z1, z2) = z1^* \times z2$$

Return Value:

A complex number.

1.7.6 Swap

Usage:

```
#include <complex.h>
complex Swap(const complex & z1, const complex& z2)
```

Description:

The function ***Swap*** switches two complex numbers z1 and z2.

Return Value:

Void

1.8 Trigonometric Functions

1.8.1 sin

Usage:

```
#include <complex.h>
complex sin (const complex& z)
```

Description:

The function ***sin*** returns the sine of the complex number in the argument, ***z***.

$$\sin(z) = \sin(a+ib) = \sin a\frac{e^b+e^{-b}}{2} - i\cos a\frac{e^b-e^{-b}}{2}$$

Example:

```
#include <gamma.h>
main()
   {
  complex z(4,1), z1, z2;          // Declare three complex, z=4+i1 and z1 = z2 = 0
  z1 = sin (z);                    // Set z1 to the sine of z
  z2 = asin (z1);                  // Set z2 to the arcsine of z1 (hopefully back to z)
  cout << z-z2;                    // This should be zero if sin and asin work
  }
```

Return Value:

A complex number, the sine of the input value

See Also:

cos, tan

1.8.2 asin

Usage:

```
#include <complex.h>
complex asin (const complex& z)
```

Description

The function ***asin*** returns the arcsine (inverse sine) of the complex number in the argument, ***z***. There are no restrictions for ***z***.

$$\mathrm{asin}(z) = -i\log(iz+\sqrt{1-z^2})$$

Example:

```
#include <gamma.h>
main()
   {
  complex z(4,1), z1, z2;          // Declare three complex, z=4+i1 and z1 = z2 = 0
  z1 = sin (z);                    // Set z1 to the sine of z
```

```
z2 = asin (z1);                          // Set z2 to the arcsine of z1 (hopefully back to z)
cout << z-z2;                            // This should be zero if sin and asin work
}
```

Return Value:

A complex number, the inverse sine of the input value.

See Also:

atan, acos

1.8.3 sinh

Usage:

```
#include <complex.h>
complex sinh (const complex& z)
```

Description:

The function ***sinh*** returns the complex hyperbolic sine of input complex number, ***z***

$$\sinh(z) = \sinh(a + ib) = \sinh a \cos b + i \cosh a \sin b$$

Example:

```
#include <gamma.h>
main()
  {
  complex c, d, e;
  c = complex (4, 1);                    // c has the value 4+i
  d = sinh (c);                          // calculate hyperbolic sine of c
  e = asinh (d);                         // calculate inverse hyperbolic sine of d
  cout << c-e;
  }
```

Return Value:

sin returns the complex hyperbolic sinus of the input value

See Also:

cosh, tanh

1.8.4 asinh

Usage:

```
#include <complex.h>
complex asinh (const complex& z)
```

Description:

The function ***asinh*** returns the complex inverse hyperbolic sine of ***z***. There are no restrictions for ***z***.

$$\operatorname{asinh}(z) = \log(z + \sqrt{z^2 + 1})$$

Example:

```
#include <gamma.h>
main()
     {
   c = complex (4, 1);                          // c has the value 4+i
   d = sinh (c);                                // calculate hyperbolic sine of c
   e = asinh (d);                               // calculate inverse hyperbolic sine of d
   cout << c-e;
   }
```

Return Value:

asinh returns the complex inverse hyperbolic sine of the input value.

See Also:

atanh, acosh

1.8.5 cos

Usage:

```
#include <complex.h>
complex cos (const complex& z)
```

Description:

The function ***cos*** returns the cosine of the input complex number ***z*** as defined by

$$\cos(a + ib) = \cos a \cosh b - i \sin a \sinh b$$

where $z = a + ib$.

Example:

```
#include <gamma.h>
main()
   {
  complex c, d, e;
  c = complex (4, 1);                         // c has the value 4+i
  d = cos (c);                                // calculate cosine of c
  e = acos (d);                               // calculate inverse cosine of d
  cout << c-e;
  }
```

Return Value:

A complex number, the complex cosine of the input value.

See Also:

sin, tan

1.8.6 acos

Usage:

```
#include <complex.h>
complex acos (const complex& z)
```

Description

The function ***acos*** returns the complex inverse cosine of the input complex number ***z*** according to

$$\mathrm{acos}(x) = -i\log(x + \sqrt{x^2 - 1})$$

There are no restrictions on ***z***.

Example:

```
#include <gamma.h>
main()
  {
  complex c, d, e;
  c = complex (4, 1);            // c has the value 4+i
  d = cos (c);                   // calculate cosine of c
  e = acos (d);                  // calculate the inverse cosine of d
  cout << c-e;
  }
```

Return Value:

A complex number, the complex inverse cosine of the input value.

See Also:

atan, asin

1.8.7 cosh

Usage:

```
#include <complex.h>
complex cosh (const complex& z)
```

Description:

The function ***cosh*** returns the complex hyperbolic cosine.

$$\cosh(a + ib) = \cosh a \cos b + i \sinh a \sin b \text{ with } a, b \in \Re .$$

Example:

```
#include <gamma.h>
main()
  {
  complex c, d, e;
  c = complex (4, 1);            // c has the value 4+i
  d = cosh (c);                  // calculate hyperbolic cosine of c
  e = acosh (d);                 // calculate inverse hyperbolic cosine of d
  cout << c-e;
```

}

Return Value:

cosh returns the complex hyperbolic cosine of the input value

See Also:

sinh, tanh

1.8.8 acosh

Usage:

```
#include <complex.h>
complex acosh (const complex& z)
```

Description

The function ***acosh*** returns the complex inverse hyperbolic cosine of ***z***. There are no restrictions for ***z***.

$$\operatorname{acosh}(z) = \log(z + \sqrt{z^2 - 1})$$

Example:

```
#include <gamma.h>
main()
   {
  complex c, d, e;
  c = complex (4, 1);                     // c has the value 4+i
  d = cosh (c);                           // calculate hyperbolic cosine of c
  e = acosh (d);                          // calculate inverse hyperbolic cosine of d
  cout << c-e;
  }
```

Return Value:

A complex number, the inverse hyperbolic cosine of the input value.

See Also:

atanh, asinh

1.8.9 tan

Usage:

```
#include <complex.h>
complex tan(const complex & z)
```

Description:

The function ***tan*** returns the complex tangent.

$$\tan(a + ib) = \frac{\sin 2a + i\sinh 2b}{\cos 2a + \cosh 2b}$$

Example:

```
#include <gamma.h>
main()
   {
  complex c, d, e;
  c = complex (4, 1);                     // c has the value 4+i
  d = tan (c);                            // calculate tangent of c
  e = atan (d);                           // calculate inverse tangent of d
  cout << c-e;
  }
```

Return Value:

tan returns the complex tangent of the input value. Results in an error if $\cos(2a) = -\cosh(2b)$

See Also:

sin, cos

1.8.10 atan

Usage:

```
#include <complex.h>
complex atan (const complex& z)
```

Description

The function ***atan*** returns the complex inverse tangent of x. There are no restrictions for x.

$$\operatorname{atan}(x) = \frac{i}{2}\log\left(\frac{1+ix}{1-ix}\right)$$

Example:

```
#include <gamma.h>
main()
   {
  complex c, d, e;
  c = complex (4, 1);                    // c has the value 4+i
  d = tan (c);                           // calculate tangent of c
  e = atan (d);                          // calculate inverse tangent of d
  }
```

Return Value:

atan returns the complex inverse tangent of the input value.

See Also:

asin, acos

1.8.11 tanh

Usage:

```
#include <complex.h>
complex tanh(const complex& z)
```

Description:

The function ***tanh*** returns the complex hyperbolic tangent.

$$\tanh(x) = \frac{\sinh x}{\cosh x}$$

Example:

```
#include <gamma.h>
main()
```

```
 {
complex c, d, e;
c = complex (4, 1);                    // c has the value 4+i
d = tanh (c);                          // calculate tangent of c
e = atanh (d);                         // calculate inverse tangent of d
}
```

Return Value:

tanh returns the complex hyperbolic tangent of the input value. Results in an error if $\cosh x = 0$.

1.8.12 atanh

Usage:

```
#include <complex.h>
complex atanh (const complex& z)
```

Description

The function ***atanh*** returns the complex inverse hyperbolic tangent of x. There are no restrictions for x.

$$\operatorname{atanh}(x) = \frac{1}{2}\log\left(\frac{1+x}{1-x}\right)$$

Example:

```
#include <gamma.h>
main()
 {
 complex c, d, e;
 c = complex (4, 1);          // c has the value 4+i
 d = tanh (c);                // calculate tangent of c
 e = atanh (d);               // calculate inverse tangent of d
```

Return Value:

atanh returns the complex inverse hyperbolic tangent of the input value.

See Also:

asinh, acosh

1.9 Transcendental Functions

1.9.1 exp

Usage:

```
#include <complex.h>
complex exp(const complex& z)
```

Description:

The function ***exp*** returns the natural exponentiation of the input complex number, ***z***.

$$\exp(z) = \exp(a+ib) = e^a \cos b + ie^a \sin b$$

Example:

```
#include <gamma.h>
main()
  {
  complex z(4,3);                    // Declare a complex number
  cout << exp(z);                    // Output the exponential
  }
```

Return Value:

A complex number, the natural exponent of the input value.

See Also:

pow, log

1.9.2 log

Usage:

```
#include <complex.h>
complex log (const complex& z)
```

Description:

The function ***log*** returns the natural logarithm of input complex number ***z***. The imaginary part is between $-\pi$ and π.

$$\log(z) = \log(a+ib) = \frac{\log(a^2+b^2)}{2} + i\,\mathrm{atan}\frac{b}{a}$$

Note that this fuction will results in an error for $z = 0$.

Example:

```
#include <gamma.h>
main()
  {
```

```
complex z(4,3);                    // Declare a complex number
cout << log(z);                    // Output the natural log of z
}
```

Return Value:

A complex number, the natural logarithm of the input value. It fails for zero input.

See Also:

exp, pow

1.9.3 pow

Usage:

```
#include <complex.h>
complex pow (const complex& z, const complex& z1)
```

Description:

The function ***pow*** returns the complex number x^y . x must be non zero.

$$\mathrm{pow}(x, y) = x^y = e^{x\log(y)}$$

Example:

```
#include <gamma.h>
main()
 {
  complex c(4,3);
  cout << pow(c);
```

Return Value:

A complex number. pow returns x^y .

See Also:

exp

1.9.4 square_norm

Usage:

```
#include <complex.h>
double square_norm (const complex& z)
```

Description:

The function ***square_norm*** returns the square of the magnitude of the input complex number, ***z***.

$$\mathrm{square\_norm}(a + ib) = a^2 + b^2$$

Example:

```
#include <complex.h>
complex z(4,3);                 // Construct a complex number, z = 4 + i3
cout << square_norm(z);         // Output the square norm (25)
```

Return Value:

A double precision number, the square of the magnitude of the input value.

See Also:

norm, phase

1.9.5 AbsNorm

Usage:

```
#include <complex.h>
double AbsNorm (const complex& z)
```

Description:

The function ***AbsNorm*** returns the absolute norm of the input complex number, ***z***.

$$AbsNorm(z) = AbsNorm(a + ib) = |a| + |b|$$

Example:

```
#include <gamma.h>
main()
 {
 complex z(4,3);                // Construct a complex number, z = 4 + i3
 cout << AbsNorm(z);            // Output the absolute norm (7)
```

Return Value:

A double precision number, the absolute norm of the input value.

See Also:

norm, phase

1.9.6 sqrt

Usage:

```
#include <complex.h>
complex sqrt (const complex& z)
```

Description:

The function ***sqrt*** returns the square root of of the input complex number ***z***.

$$sqrt(z) = sqrt(a + ib) = \sqrt{a + ib} = \sqrt{\frac{a + \sqrt{a^2 + b^2}}{2}} + \sqrt{\frac{-a + \sqrt{a^2 + b^2}}{2}}$$

Example:

```
#include <gamma.h>
main()
 {
  complex z (4, 1);                 // c has the value 4+i
  z = sqrt(z);
```

Return Value:

A complex number, square root of the input value.

See Also:

pow

1.10 Additional Functions

1.10.1 add

Usage:

```
#include <complex.h>
void add ( complex& z, const complex& 1, const complex& z2);
void add ( complex& z, double x, const complex& z1);
void add ( complex& z, const complex& z1, double x);
```

Description

The function ***add*** performs the same calculation as the ***operator* +** but is faster.

Return Value:

Nothing.

See Also:

+, sub, mul, div

1.10.2 sub

Usage:

```
#include <complex.h>
void sub ( complex& z, const complex& z1, const complex& z2);
void sub ( complex& z, double x, const complex& z1);
void sub ( complex& z, const complex& z1, double x);
```

Description

The function ***sub*** performs the same calculation as the ***operator*** - but is faster.

Return Value:

Nothing.

See Also:

-, mul, div, add

1.10.3 mul

Usage:

```
#include <complex.h>
void mul ( complex& z, const complex& z1, const complex& z2);
void mul ( complex& z, double x, const complex& 1);
void mul ( complex& z, const complex& z1, double x);
```

Description

The function ***mul*** performs the same calculation as the ***operator **** but is faster.

Return Value:

Nothing.

See Also:

+, sub, div, add

1.10.4 div

Usage:

```
#include <complex.h>
void div ( complex& z, const complex& z1, const complex& z2);
void div ( complex& z, double x, const complex& z1);
void div ( complex& z, const complex& z1, double x);
```

Description

The function ***div*** performs the same calculation as the ***operator /*** but is faster.

Return Value:

Nothing.

See Also:

+, sub, mul, add

1.11 Class Implementation

1.11.1 Data Structure

The class ***complex*** implements a complex number, z, as a pair of two double precision numbers, herein called a & b, as given in the following equation.

$$z = a + \iota b \tag{2-1}$$

There is no additional information stored per complex number. However class ***complex*** does contain a few static variables which define the output format. These are stored only once per program that includes the class. These are the five integers (_phase, _math, _science, _digits, _digits_after_point) described in the function ***complex_setf***. There is also an array of 128 characters called _form which is used in internal print statements fo incorporate the format specified by the five static integers.

1.11.2 Algorithms

Currently, all complex number algorithms are designed in a straight forward manner. The more complicated arithmetic functions are implemented through use of the library functions which have been provided for double numbers.

The functions *Re* and *Im* allow the user to access the real and imaginary parts of the complex number, a and b respectively. Similarly, the functions *norm* and *phase* give the user access their corresponding parts of the complex number, R and φ respectively.

$$z = a + \iota b = Re^{i\varphi} \tag{2-2}$$

The analogous functions to set these quantities are also provided. It is currently much faster to get and set the real and imaginary parts of z than it is to perform the same operations with the phase or norm.

2 Column & Row Vectors

2.1 Overview

The class column vector implements column vectors. The corresponding class row vectors implements the row vectors. Both classes are derived from the class matrix.

2.2 Available Vector Functions

Basic Functions

row_vector() col_vec-	Constructors		page 35
col_vector()	-		page 35
=	- Assignment	v1 = v2	page 35

Basic Functions

+	- Addition	v3 = v1 + v2	page 37
+=	- Unary Addition	v1 += v2	page 37
-	- Subtraction, Negation	v3 = v1-v2, -z	page 37
-=	- Unary Subtraction	v1 -=v2	page 38
*	- Multiplication	v3 = v1*v2	page 38
*=	- Unary Multiplication	v1 *= v2	page 39
/	- Division	v3 = v1/v2	page 39
/=	- Unary Division	v1 /= v2	page 39
==	- Equality	v1 == v2	page 40
!=	- Inequality	v1 != v2	page 40

Vector Transcendental Functions

Vector Comparison & Test Functions

Input/Output

row_vec col_vec	- Constructors		page 46
+	- Addition	z3 = z1 + z2, z2 = z1 + r	page 4
+=	- Unary Addition	z1 += z2	page 5

2.3 Constructors

2.3.1 row_vector()

2.3.2 col_vector()

Usage:

```
#include <row_vector.h>
#include <col_vector.h>
row_vector();
row_vector(int N);
row_vector(int N, const complex& z);
col_vector();
col_vector(int N);
col_vector(int N, const complex& z);
```

Description:

The ***row_vector*** and ***col_vector*** are the constructors for row and column vectors respectively. With no arguments the vector is constructed with no elements. With a single integer ***N*** as the argument, the vector is constructed with ***N*** uninitilized complex elements. With an additional value added (either double or complex), ***z***, the ***N*** elements are initilized to the value specified.

Examples:

```
row_vector rv;                        // Null row_vector
col_vector cv(298);                   // Column vector with 298 points
row_vector rvx(3459, complex(2,7));   // Row vector, 3498 pts, all 2+7i
col_vector cvs[27];                   // 27 unitialized column vectors
```

Return Value:

A new row or column vector is returned.

See Also:

put, get, put_h

2.3.3 =

Usage:

```
#include <column_vector.h>
void operator = ( const col_vector& cv);
void operator = (const  row_vector& rv);
```

Description:

= assigns a vector to an other vector. The old data will be deleted.

Return Value:

Nothing.

See Also:

col_vector, row_vector

2.4 Basic Functions

2.4.1 +

Usage:

```
#include <column_vector.h>
col_vector operator + ( col_vector& a, col_vector& b);
row_vector operator + ( row_vector& a, row_vector& b);
```

Description:

+ adds two col_vectors/row_vectors with the same number of rows or cols.

Return Value:

returns the sum of a and b.

See Also:

-, *, /, +=, -=, *=, /=

2.4.2 +=

Usage:

```
#include <column_vector.h>
void operator += ( col_vector& a);
void operator += ( row_vector& a);
```

Description:

+= adds a to this.

Return Value:

returns nothing.

See Also:

+, -, *, /, -=, *=, /=

2.4.3 -

Usage:

```
#include <column_vector.h>
col_vector operator - ( col_vector& a, col_vector& b);
row_vector operator - ( row_vector& a, row_vector& b);
```

Description:

- subtracts two col_vectors/row_vectors with the same number of rows or cols.

Return Value:

returns the difference of a and b.

See Also:

+, *, /, +=, -=, *=, /=

2.4.4 -=

Usage:

```
#include <column_vector.h>
void operator -= ( col_vector& a);
void operator -= ( row_vector& a);
```

Description:

-= subtracts a from this.

Return Value:

returns nothing.

See Also:

+, -, *, /, -=, *=, /=

2.4.5 *

Usage:

```
#include <row_vector.h>
#include <col_vector.h>
complex operator* ( row_vector& a, row_vector& b);          (1)
complex operator* ( col_vector& a, row_vector& b);          (1)
complex operator* ( row_vector& a, col_vector& b);          (1)
complex operator* ( col_vector& a, col_vector& b);          (1)
col_vector operator* ( matrix& a, col_vector& b);           (2)
row_vector operator* ( row_vector& a, matrix& b);           (2)
col_vector operator* ( complex a, col_vector& b);           (3)
col_vector operator* ( col_vector& a, complex b);           (3)
row_vector operator* ( complex a, row_vector& b);           (3)
row_vector operator* ( row_vector& a, complex b);           (3)
```

Description:

(1) calculates the scalar product of two vectors, ignoring the type of them.

(2) calculates the product of a vector with a matrix.

(3) multiplies the vector with a scalar.

Return Value:

(1) returns the scalar product (complex number).

(2) and (3) return a vector.

See Also:

+, -, /, +=, -=, *=, /=

2.4.6 *=

Usage:

```
#include <column_vector.h>
void operator *= ( complex a);
```

Description:

*= multiplies the vector by a.

Return Value:

returns nothing.

See Also:

+, -, *, /, +=, -=, /=

2.4.7 /

Usage:

```
#include <column_vector.h>
row_vector operator/ ( complex a, row_vector& b);
row_vector operator/ ( row_vector& a, complex b);
```

Description:

/ divides the vector with a scalar.

Return Value:

/ returns a vector.

See Also:

+, -, *, +=, -=, *=, /=

2.4.8 /=

Usage:

```
#include <column_vector.h>
void operator /= ( complex a);
```

Description:

/= divides the vector by a.

Return Value:

returns nothing.

See Also:

+, -, *, /, +=, -=, *=

2.4.9 ==

Usage:

```
#include <column_vector.h>
int operator == ( col_vector& mx1, col_vector& mx2);
int operator == ( row_vector& mx1, row_vector& mx2);
```

Description:

== compares two vectors. It returns TRUE if they are equal.

Return Value:

A flag, wether the vectors are equal or not.

See Also:

!=

2.4.10 !=

Usage:

```
#include <column_vector.h>
int operator != ( col_vector& mx1, col_vector& mx2);
int operator != ( row_vector& mx1, row_vector& mx2);
```

Description:

== compares two vectors. It returns FALSE if they are equal.

Return Value:

A flag, wether the vectors are equal or not.

See Also:

==

2.5 Access Functions

2.5.1 ()

Usage:

```
#include <column_vector.h>
complex& operator() ( int i);
```

Description:

() returns the value of the vector at Position i.. The elements are numbered from 0 to elements-a.

Example:

```
complex c;
row_vector a(3);
a(1) = 5;
c = a(1);
```

Return Value:

A reference to a complex number.

See Also:

put, get

See Also:

2.5.2 elements

Usage:

```
#include <column_vector.h>
int elements( );
```

Description:

elements returns the number of elements in the vector.

Example:

```
row_vector a;
cout << a.elementss();
```

Return Value:

elements returns the number of elementss in the vector.

See Also:

2.5.3 FFT

Usage:

```
#include <column_vector.h>
row_vector FFT (row_vector& a);
col_vector FFT (col_vector& a);
```

Description:

FFT returns the fouiertransform b of the row or column vector a. The vector a must be of size 2^n.

$$FFT(a)_n = b_n = \sum_{k=0}^{size(a)-1} a_k e^{(2\Pi ikn)/size(a)} \tag{2-3}$$

Return Value:

FFT returns the transformed vector.

See Also:

IFFT

2.5.4 get

Usage:

```
#include <column_vector.h>
complex get(int i);
```

Description:

get returns the value of the vector at position i. The elements are numbered from 0 to elements-1.

Example:

```
row_vector a(2);
cou << a.get (1);
```

Return Value:

A complex number.

See Also:

put, ()

2.5.5 IFFT

Usage:

#include <column_vector.h>
row_vector IFFT (row_vector& a);
col_vector IFFT (col_vector& a);

Description:

IFFT returns the inverse fouiertransform b of the vector a. The vector a must be of size 2^n.

$$IFFT(a)_n = b_n = \frac{1}{size(a)} \sum_{k=0}^{size(a)-1} a_k e^{(-2\Pi ikn)/size(a)} \tag{2-4}$$

Return Value:

IFFT returns the transformed vector.

See Also:

FFT

2.5.6 prod

Usage:

#include <column_vector.h>
row_vector prod (row_vector& a, row_vector& b);
col_vector prod (col_vector& a, row_vector& b);
row_vector prod (row_vector& a, col_vector& b);
col_vector prod (col_vector& a, col_vector& b);

Description:

prod calculates the vector product of two vectors, ignoring the type of them.

Return Value:

prod return a vector.

See Also:

+, -, /, +=, -=, *=, /=

2.5.7 put

Usage:

#include <column_vector.h>
void put(complex& c, int i);

Description:

put sets the value of the vector at i. The elements are numbered from 0 to elements-1.

Example:

complex c;
row_vector a(2);
a.put (c,1);

Return Value:

Nothing.

See Also:

get, ()

2.5.8 row_vector

Usage:

#include <column_vector.h>
row_vector ( ); (1)
row_vector (int i); (2)
row_vector (int i, complex& c); (3)
row_vector (row_vector& a); (4)
row_vector (matrix& a); (5)

Description:

Constructs a row vector.
(1) is the default constructor.
(2) constructs a vector with i elements;
(3) constructs a vector with i elements of value c.
(4) is the copy constructor.
(5) creates a vector from a matrix (if possible).

Return Value:

The constructed vector.

See Also:

2.5.9 ~col_vector

Usage:

#include <column_vector.h>
~col_vector ( )

Description:

destroys a vector

Return Value:

nothing

See Also:

col_vector

2.5.10 ~row_vector

Usage:

```
#include <column_vector.h>
~row_vector ( )
```

Description:

destroys a vector

Return Value:

nothing

See Also:

row_vector

2.6 Input/Output

2.6.1 <<

Usage:

```
#include <row_vector.h>
#include <col_vector.h>
ostream& operator << (ostream& ostr, const col_vector& cv)
ostream& operator << (ostream& ostr, const row_vector& rv)
```

Description:

The ***operator*** **<<** is the standard output for printing a complex number ***z***. The format of the output may be set with the function ***complex_setf***.

Example:

```
#include <gamma.h>
main()
  {
  complex z;                    // Declare a complex number
  cout << z;                    // Print the complex number to standard output.
  }
```

Return Value:

Returns a modified output stream after having written the vector. The format is

Matrix Size1 x 2

(1.000000, 0.000000)

(0.000000, 0.000000)

In this example, the output format of each complex number is (a, b) to indicate a + ib. For other possible formats see *Class COMPLEX*.

See Also:

2.6.2 >>

Usage:

```
#include <column_vector.h>
ostream& operator >> ( istream& a, col_vector& b);
ostream& operator >> ( istream& a, row_vector& b);
```

Description:

Reads a vector from the inputstrem. The format is identical to matrix.

Return Value:

Returns the modified input stream.

2.7 Implementation

Vector is a derived class from matrix and many arithmetic functions from matrix. A row vector has one row and n columns, a column vector has n rows and one column.

Only the functions with special meaning inside vector are defined in row_vector or col_vector. The multiplication of vectors returns a complex number, and not a 1 by 1 matrix.

3 Class Matrix

3.1 Overview

The class ***matrix*** provides most of the functions GAMMA uses to create and manipulate matrices. Transparent to the user, class ***matrix*** internally supports several different matrix types[1]. The principal advantage is that calculations with, and storage of, zero's can be neglected. For example, the following table indicates the relative time for matrix multiplication of n*n matrices

Table 2: Compuation Time Order for Matrix Multiplication.

	n_matrix	d_matrix	i_matrix
n_matrix	$O(n^3)$	$O(n^2)$	O(1)
d_matrix	$O(n^2)$	O(n)	O(1)
i_matrix	O(1)	O(1)	O(1)

This computational savings occurs not only for common algebraic manipulations (+, -, *, /) but also in more complex functions which can take advantage of specific matrix structures (exponentiation, diagonalization, inversion, etc.). Furthermore, there is a savings in memory as well, obviously zeros are not stored when the matrix structure is known. There is a small overhead in GAMMA in that each matrix stores its type and that type checking is done during computations. However, in all but 2x2 arrays, the advantages far outweigh this overhead for virtually all calculations.

Keep in mind that conversion between the different matrix types is normally done automatically, the GAMMA user need not be concerned about this. If the user wants to insure that matrices are stored properly there are routines both to test whether a matrix is of a given type and to force a matrix to be of a given type (set_type, test_type, check_type, stored_type).

To use class ***matrix*** one must include the file ***matrix.h*** or all of GAMMA, i.e. ***gamma.h***

Note: Indexing of matrices follows the standard of C and C++, beginning with 0.

1. Currently, GAMMA uses only complex double precision arrays. This will change as GAMMA is extended. Any number of new matrix types may be added into the platform without affecting GAMMA program codes.

3.2 Available Matrix Functions

Matrix Algebraic

matrix	- Constructor		page 69
~matrix	- Destructor		page 80
=	- Assignment	mx1 = mx2	page 57
+	- Addition	mx3 = mx1 + mx2	page 50
+=	- Unary Addition	mx1 += mx2	page 52
-	- Subtraction, Negation	mx3 = mx1-mx2, -mx	page 52
-=	- Unary Subtraction	mx1 -= mx2	page 53
*	- Multiplication	mx1*mx2, z*mx, mx*z	page 53
*=	- Unary Multiplication	mx1 *= mx2, mx1 *= z	page 54
/	- Division		page 54
/=	- Unary Division		page 52

Matrix Access Functions

()	- Retrieve a matrix element	mx(i,j)	page 58
get	- Retrieve a matrix element	mx.get(i,j)	page 66
put	- Set a matrix element	mx.put(z,i,j)	page 69
put_h	- Set a Hermitian matrix element	mx.put_h(z,i,j)	page 70
get_block	- Retrieve a matrix sub-block	mx.get_block(i,j,rows,cols)	page 66
put_block	- Set a sub-block in a matrix	mx.put_block(i,j,mx)	page 70

Basic Matrix Functions

cols	- Retrieve # of matrix columns	mx.cols()	page 63
rows	- Retrieve # of matrix rows)	mx.rows()	page 72

Complex Matrix Functions

tensor_product	- Tensor product of two matrices	$mx3 = mx1 \otimes mx2$	page 75
adjoint	- Take the adjoint of a matrix	$mx1 = mx2^{\dagger}$	page 59
conj	- Take the conjugate of a matrix	$mx1 = mx2^{*}$	page 63
adjoint_times	- Multiply matrix adjoint & matrix	$mx3 = mx1^{\dagger} * mx2$	page 59
times_adjoint	- Multiply matrix & matrix adjoint	$mx3 = mx1 * mx2^{\dagger}$	page 77
transpose	- Take the transpose of a matrix	$mx1 = mx2^{T}$	page 80
diag	- Diagonalize a matrix	diag(mx)	page 65
inv	- Invert a matrix	$mx1 = mx2^{-1}$	page 68
det	- Take the determinant of a matrix	det(mx)	page 63
trace	- Trace of matrix or matrices	Tr(mx), Tr(mx1*mx2)	page 78
rank	- Get the rank of a matrix	rank(mx)	page 72
FFT	- Fourier transform of a matrix	mx2 = FFT(mx1)	page 65
IFFT	- Matrix inverse Fourier transform	mx2 = IFFT(mx1)	page 68
resize	- Resize a matrix	mx1 = mx2.resize(i,j)	

Matrix Comparison & Test Functions

==	- Matrix Equality	mx1 == mx2	page 57
!=	- Matrix Inequality	mx1 != mx2	page 58

check_type	- Check Matrix type	mx.check_type(type, x)	page 61
set_type	- Set Matrix type		page 73
stored_type	- Stored Matrix type		page 74
test_type	- Test Matrix type		page 77
test_hermitian	- Test Matrix Hermitian type		page 75
stored_hermitian	- Test Matrix Stored Hermitian type		page 73
set_hermitian	- Set Matrix Hermitian type		page 72
check_hermitian	- Check Matrix Hermitian type		page 61

Matrix Input & Output Functions

<<	- Matrix Output	page 50
>>	- Matrix Input	page 55
write	- Matrix output to a binary file	
read	- Matrix input from a binary file	

3.3 Routines

3.3.1 <<

Usage:

```
#include <matrix.h>
ostream& operator << (ostream& ostr, matrix& mx);
```

Description:

This matrix ***operator*** **<<** is used to send a matrix into an ostream. This can be used for printing to standard output or to a file. An example of the printed matrix format is

```
Matrix Size 2 x 3
( 1.000000, 0.000000) ( 0.000000, 0.000000) ( 3.000000, 3.000000)
( 0.000000, 0.000000) ( 1.000000, 0.000000) ( 3.000000, 3.000000)
```

In this example, the format of each element (a complex number) can be set by the user. For other possible formats see ***Class COMPLEX,*** function ***complex_setf***.

Return Value:

Returns the modified output stream.

See Also:

>>, complex::<<

3.3.2 +

Usage:

```
#include <matrix.h>
matrix operator + (matrix& mx1, matrix& mx2);
```

Description:

The matrix ***addition operator*** **+** adds two matrices having the same number of rows and cols. The type of the returned matrix depends on the type of the input matrices. For example, the sum of a diagonal matrix and a normal matrix is a normal matrix. An error will occur if addition is attempted on matrices not having the same dimensions.

Return Value:

returns a matrix.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1, mx2, mx3;                  // Construct three matrices
  mx3 = mx1+mx2;                         // Set mx3 to the sum of mx1 and mx2
  cout << mx3+mx2                     ;  // Output the result of mx3 added to mx2
  }
```

See Also:

-, *, /, +=, -=, *=, /=

3.3.3 +=

Usage:

#include <matrix.h>
void matrix::operator += (matrix& mx);

Description:

The matrix ***unary addition operator* +=** adds a matrix to the original matrix. The original matrix is modified by this operation. This addition, mx1 += mx2, is much preferred over mx1 = mx1 + mx2 because it is both faster and uses less memory.

Return Value:

returns nothing.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1, mx2;                     // Construct two matrices
  mx1 += mx2;                          // Set mx1 to the sum of mx1 and mx2
  }
```

See Also:

+, -, *, /, -=, *=, /=

3.3.4 -

Usage:

#include <matrix.h>
matrix operator - (matrix& mx1, matrix& mx2);
matrix operator - (matrix& mx);

Description:

The matrix ***operator*** - is used to both subtract two matrices and to negate a matrix.

1. -(matrix& mx1, matrix& mx2)- Subtracts matrix mx2 from matrix mx1 to produces a new matrix. An error will result unless mx1 and mx2 have the same number of rows and cols. The type of the returned matrix depends on the type of the input matrices.
2. -matrix(matrix& mx)- A matrix is returned having all its elements the negatives of the input matrix elements.

Return Value:

Returns a matrix

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1, mx2, mx3;                // Construct three matrices
  mx3 = mx1-mx2;                       // Set mx3 to the difference of mx1 and mx2
```

```
}
```

See Also:

+, *, /, +=, -=, *=, /=

3.3.5 -=

Usage:

#include <matrix.h>

void matrix::operator -= (matrix& mx1);

Description:

The matrix ***unary subtraction operator* -=** subtracts the matrix given as an argument from the original matrix.

The original matrix is modified by this operation. Unary subtraction, mx1 -= mx2, is much preferred over mx1 = mx1 - mx2 because it is both computationally faster and uses less memory.

Return Value:

returns nothing.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1, mx2, mx3;                    // Construct two matrices
  mx2 -= mx1;                              // Subtract mx1 from mx2
  }
```

See Also:

+, -, *, /, +=, *=, /=

3.3.6 *

Usage:

#include <matrix.h>

matrix operator * (complex& z, matrix& mx);

matrix operator * (matrix& mx, complex& z);

matrix operator * (matrix& mx1, matrix& mx2);

Description:

The matrix ***multiplication operator* *** multiplies two matrices together or multiplies a matrix by a complex number.

1. *(complex&z, matrix& mx)- Returns a matrix with all elements equal to the input matrix elements multiplied by the complex number z.
2. *matrix(matrix& mx, complex& z)- Performs the identical operation as the previous use.
3. *matrix(matrix& mx1, matrix& mx2)- A matrix is returned having all its elements the negatives of the input matrix elements. Here, the number of rows of the second matrix has to be the number of cols of the first matrix.

Return Value:

Returns a matrix

Example:

```
#include <gamma.h>
main()
  {
  complex z;                        // Declare a complex number
  matrix mx1(2,5), mx2(5,3);        // Construct two matrices, one 2x5 and one 5x3
  matrix mx3 = mx1*mx2;             // Make mx3, set to mx1*mx2 a 2x3 array.
  mx3 = z*mx1 + mx2*z;              // Set mx3 to be z*(mx1+mx2) the hard way
  }
```

See Also:

-, *, /, +=, -=, *=, /=

3.3.7 *=

Usage:

#include <matrix.h>

void matrix::operator *= (matrix& a, matrix& b);

void matrix::operator *= (matrix& a, complex& z);

Description:

The ***unary multiplication operator *=*** multiplies the first matrix with the second argument.

Return Value:

returns nothing.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1(2,2), mx2;             // Construct two matrices, one 2x2 and one NULL
  mx2 = mx1;                        // Now mx2 is also 2x2 and identical to mx1
  if(mx1 == mx2)                    // Test whether mx2 and mx1 are equivalent
    cout << "Matrices are equal";
  }
```

See Also:

+, -, *, /, +=, -=, /=

3.3.8 /

Usage:

#include <matrix.h>

matrix operator / (matrix& mx, complex& z);

matrix operator / (complex& z, matrix& mx);

matrix operator / (matrix& mx1, matrix& mx2);

Description:

The matrix ***division operator* *=** defines division of one matrix by another matrix and division of a matrix by a scalar.

1. /(matrix& mx, complex&z) - divides a matrix by a complex number, this is the same as multiplying with the inverse of the complex number.
2. /(complex&z, matrix& mx) - divides a complex number by a matrix. This is the same as multiply the complex number with the inverse matrix (if it exists).
3. /matrix(matrix& mx1, matrix& mx2) - divides a matrix by a matrix. This is the same as multiplying with the inverse matrix. The inverse of mx2 must exist.

Return Value:

A matrix.

See Also:

+, -, *, +=, -=, *=, /=, inv

3.3.9 /=

Usage:

#include <matrix.h>
void matrix::operator /= (matrix& mx1, matrix& mx2);
void matrix::operator /= (matrix& mx, complex& z);

Description:

The ***unary division operator* /=** divides the first matrix by the second argument, either another matrix or a scalar.

Return Value:

returns nothing.

See Also:

+, -, *, /, +=, -=, *=, inv

3.3.10 >>

Usage:

#include <matrix.h>
istream& operator >> (istream& a, matrix& b);

Description:

>> read a matrix from an input stream. The format is:
#rows #cols
complex numbers (#cols*#rows)

Example:

The input for a 2 by 3 matrix would be:

2 3		
1 0	2 3	3 2
3 2	3 4	4 3

Where each pair of numbers represents a complex number.

Return Value:

returns the modified input stream and the matrix b.

See Also:

<<

3.3.11 =

Usage:

#include <matrix.h>

void matrix::operator = (matrix& mx);

Description:

The ***matrix operator*** = assigns a matrix to another matrix. Any old data concerning the matrix being assigned is automatically deleted.

Return Value:

Nothing.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1(2,2), mx2;                    // Construct two matrices, one 2x2 and one NULL
  mx2 = mx1;                               // Now mx2 is also 2x2 and identical to mx1
  }
```

See Also:

matrix

3.3.12 ==

Usage:

#include <matrix.h>

int operator == (matrix& mx1, matrix& mx2);

Description:

The ***matrix operator*** == compares two matrices. It returns TRUE if all elements of the two matrices are equal.

Return Value:

An integer.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1(2,2), mx2;                    // Construct two matrices, one 2x2 and one NULL
  mx2 = mx1;                               // Now mx2 is also 2x2 and identical to mx1
  if(mx1 == mx2)                           // Test whether mx2 and mx1 are equivalent
    cout << "Matrices are equal";
  }
```

See Also:

!=

3.3.13 !=

Usage:

#include <matrix.h>
int operator != (matrix& mx1, matrix& mx2);

Description:

The ***matrix operator !=*** compares two matrices. It returns FALSE if they are equal.

Return Value:

A flag, whether the matrices are equal or not.

```
#include <gamma.h>
main()
  {
  matrix mx1(2,2), mx2;                    // Construct two matrices, one 2x2 and one NULL
  mx2 = mx1;                               // Now mx2 is also 2x2 and identical to mx1
  if(mx1 == mx2)                           // Test whether mx2 and mx1 are equivalent
    cout << "Matrices are equal";
  }
```

See Also:

==

3.3.14 ()

Usage:

#include <matrix.h>
complex& operator() (int i, int j);

Description:

The ***matrix operator ()*** returns the value of the matrix at i,j. The columns and rows are numbered from 0 to cols-1 and 0 to rows-1. Please note that it is much preferable to use the **get** function for element access.

Example:

```
#include <gamma.h>
main()
  {
  complex z;
  matrix a(2,2);
  a(1,0) = 5;
  z = a(1,0);
  }
```

Return Value:

A reference to a complex number. If the element is not stored, for example, the off diagonals in diagonal matrices, the reference will point to a dummy variable. Assignment to it will be ignored. Such will not occur with the ***get*** function.

See Also:

put, get, put_h

3.3.15 adjoint_times

Usage:

```
#include <matrix.h>
matrix adjoint_times (matrix& mx1, matrix& mx2);
```

Description:

The function ***adjoint_times*** returns the product of the adjoint of the first matrix input with the second matrix input. This is given mathematically as

$$mx3 = (mx1)^{\dagger} mx2 = (mx1^{*})^{T} mx2$$

or equivalently

$$mx3(i, j) = \sum_{k=0}^{rows-1} [mx1(i, k)]^{\dagger} \times mx2(k, j) = \sum_{k=0}^{rows-1} mx1^{*}(k, i) \times mx2(k, j)$$

Use of this function is computationally faster and uses less memory than performing the same mathematics using the adjoint function and the multplication function.

Return Value:

Returns a matrix.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1, mx2, mx3;                  // Construct three matrices
  mx3 = adjoint_times(mx1, mx2);         // Set mx3 to be [mx1]† * mx2
  }
```

See Also:

adjoint, times_adjoint

3.3.16 adjoint

Usage:

```
#include <matrix.h>
matrix adjoint (matrix& mx);
```

Description:

The function ***adjoint*** returns the adjoint of the input matrix,

$$mx_{out} = mx_{in}^{\dagger} = (mx_{in}^{*})^{T}$$

where

$$mx_{out}(i, j) = mx_{in}^{\dagger}(i, j) = [mx_{in}^{*}(i, j)]^{T} = mx_{in}^{*}(j, i) \quad .$$

Return Value:

A matrix is returned

Example:

```
#include <gamma.h>
main()
  {
  matrix mx(2,2);                      // Construct a 2x2 matrix
  matrix mx1 = adjoint(mx);            // Construct a second matrix set to the adjoint of mx
  }
```

See Also:

adjoint_times, times_adjoint

3.3.17 check_hermitian

Usage:

```
#include <matrix.h>
hermitian_type check_hermitian (hermitian_type h = _hermitian);
```

Description:

The function ***check_hermitian*** is used to check whether a matrix is Hermitian or non-Hermitian. If the input argument ***h*** is _hermitian, then the matrix elements are checked and, if possible, the matrix is converted to a hermitian matrix. if possible and _hermitian is returned, otherwise non_hermitian is returned. If ***h*** is non_hermitian then the matrix Hermtian type is converted to an non_hermitian. This function will alter the matrix Hermitian type if necessary!.

Example:

```
#include <gamma.h>
main()
  {
matrix a(2,2);
if (a.check_hermitian ())
    {cout << “a converted to a hermitian matrix”;}
  }
```

Return Value:

A flag whether a is nor hermitian or not.

See Also:

set_hermitian, stored_hermitian, test_hermitian, hermitian_type

3.3.18 check_type

Usage:

```
#include <matrix.h>
matrix_type matrix::check_type (matrix_type t, double d=0.001);
```

Description:

The function ***check_type*** checks whether a matrix is of type t or not and converts it. It returns the type of the new matrix. Elements with a norm smaller than d are assumed to be zero.

Example:

```
#include <gamma.h>
main()
  {
  matrix a(2,2);
  if (a.check_type (d_matrix))
    {cout << “a converted to a d_matrix”;}
  }
```

Return Value:

The new type of them matrix.

See Also:

set_type, stored_type, test_type, matrix_type

3.3.19 cols

Usage:

#include <matrix.h>
int matrix::cols ();

Description:

The function ***cols*** returns the number of columns in the input matrix.

Return Value:

An integer is returned.

See Also:

rows

3.3.20 conj

Usage:

#include <matrix.h>
matrix conj (matrix& mx);

Description:

The function ***conj*** returns the complex conjugate of the input matrix, $mx_{out} = mx_{in}^*$ where $mx_{out}(i, j) = mx_{in}^*(i, j)$.

Return Value:

A matrix is returned

Example:

```
#include <gamma.h>
main()
  {
  matrix mx(2,2);                    // Construct a 2x2 matrix
  matrix mx1 = conj(mx);             // Construct a 2nd matrix set to the conjugate of mx
  }
```

See Also:

adjoint_times, times_adjoint

3.3.21 det

Usage:

#include <matrix.h>
complex det (matrix& a);

Description:

Funciton ***det*** calculates the determinant of a matrix.

Return Value:

Returns the determinate of a matrix.

3.3.22 diag

Usage:

#include <matrix.h>
void diag (matrix& mx1, matrix& mx2, matrix& mx3);

Description:

The function ***diag*** diagonalizes a matrix[1]. The first argument is the input matrix. The second and the third arguments contain the result of the diagonalization proceedure; mx2 is the diagonal matrix filled with the eigenvalues of mx1 and mx3 is a unitary matrix whose columns are filled with the eigenvectors of mx1.

These matrices are mathematically related by the following

$$mx1_{matrix} \cdot mx3_{vectors} = mx3_{vectors} \cdot mx2_{diag}$$

For more explicit details see the discussion at the end of this chapter, page 83.

Return Value:

Returns no value.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1, mx2, mx3;                    // Construct e matrices
  diag(mx1, mx2, mx3);                     // Diagonalize mx1 to mx2, eigenvectors mx3
  }
```

See Also:

check_hermitian

3.3.23 FFT

Usage:

#include <matrix.h>
matrix FFT (matrix& mx);

Description:

The function ***FFT*** returns a matrix that is the Fourier transform of the input matrix. The input matrix must be of size $1*2^n$ or 2^n*1.

$$FFT(a)_n = b_n = \sum_{k=0}^{size(a)-1} a_k e^{(2\Pi ikn)/size(a)}$$

1. Currently GAMMA deals with complex Hermitian matrices and may have trouble on other types of arrays. Originally if the input matrix was not Hermitian an error message is printed and the routine quit, but this is being fixed.

Return Value:

FFT returns the transformed matrix.

See Also:

IFFT

3.3.24 get

Usage:

#include <matrix.h>

complex matrix::get(int i, int j);

Description:

The function ***get*** returns the value of the matrix element at position i,j. The columns and rows are numbered from 0 to cols-1 and 0 to rows-1.

Example:

```
#include <gamma.h>
main()
  {
  complex z;                    // Declare a complex number
  matrix mx(2,2);               // Declare a 2x2 matrix
  z = mx.get(1,2);              // Set the number to the matrix element 1,2
  z = get (mx,1,2);             // An alternative method of accessing an element
  }
```

Return Value:

A complex number.

See Also:

put, (), put_h, get_block

3.3.25 get_block

Usage:

#include <matrix.h>

matrix matrix::get_block(int row, int col, int rows, int cols);

Description:

The function ***get_block*** returns a new matrix which is a sub-block of the input matrix. The sub-block begins at the element (row, col) and has the size (rows x cols). The type of matrix returned depends on the type of the matrix input and the position of the sub-block. The columns and rows are numbered from 0 to cols-1 and 0 to rows-1. Keep in mind that the returned matrix will have a row dimension rows and a column dimension cols.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1(10,10), mx2;       // Declare 2 matrices, one 10x10, one NULL
```

```
mx2 = mx1.get_block(1,2,5,5);            // Set mx2 to a 5x5 matrix, the sub-block of mx1
}
```

Return Value:

A matrix.

See Also:

put, (), put_h, get, put_block

See Also:

check_hermitian, set_hermitian, stored_hermitian, test_hermitian

3.3.26 IFFT

Usage:

#include <matrix.h>
matrix IFFT (matrix& a);

Description:

The function ***IFFT*** returns the inverse Fourier transform b of the matrix a. The matrix a must be of size $1*2^n$ or 2^n*1.

$$IFFT(a)_n = b_n = \frac{1}{size(a)} \sum_{k=0}^{size(a)-1} a_k e^{(-2\Pi ikn)/size(a)}$$

Return Value:

IFFT returns the transformed matrix.

See Also:

FFT

3.3.27 inv

Usage:

#include <matrix.h>
matrix inv (matrix& mx);

Description:

The function ***inv*** returns a matrix that is the inverse of the input matrix

$$mx_{out} = mx_{in}^{-1}$$

where

$$mx_{out} \cdot mx_{in} = I$$

The algorithm utilized depends on the type of the matrix which is input. Keep in mind that not all matrices are invertible; singlular matrices have no inverse.

Return Value:

A matrix is returned.

See Also:

/, /=

3.3.28 matrix

Usage:

```
#include <matrix.h>
matrix ( );
matrix ( int i, int j = 0, matrix_type t = n_matrix_type, hermitian_type h = non_hermitian);
matrix (int i, int j, complex& z, matrix_type t = n_matrix_type, hermitian_type h = non_hermitian);
matrix (matrix& mx);
```

Description:

The function ***matrix*** is the constructor for the class matrix.

1. matrix() - Creates an empty matrix which can later be explicitly specified.
2. matrix(int i, int j=0, matrix_type t = n_matrix_type, hermitian_type h = non_hermitian) - Constructs a matrix of dimension i x j. By default, the matrix type is set to normal and non_hermitian but this can be specified by including the optional arguments t (the matrix type) and h (the hermitian type)[1].
3. matrix(int i, int j, complex z, matrix_type t = n_matrix_type, hermitian_type h = non_hermitian) - Same as the previous function form except all matrix elements are initialized to the value z.
4. matrix(matrix& mx) - Constructs a matrix which is identical to the matrix given as an argument

Example:

```
#include <gamma.h>
main()
  {
  complex z;                    // Declare a complex number
  matrix mx1;                   // Creates a NULL matrix (1x1)
  matrix mx2(2,2);              // Creates a 2x2 matrix
  matrix mx3(3,2,z);            // Creates a 3x2 matrix, all elements having value z
  }
```

Return Value:

The created matrix.

See Also:

matrix, check_type, set_type, stored_type, test_type

3.3.29 put

Usage:

```
#include <matrix.h>
void matrix::put(complex& z, int i, int j);
```

Description:

The function ***put*** sets the value of the matrix element at position i,j. The columns and rows are numbered from 0 to cols-1 and 0 to rows-1.

1. Current matrix types supported in GAMMA are listed at the end of this chapter, see page 84.

Example:

```
#include <gamma.h>
main()
  {
  complex z;                      // Declare a complex number z
  matrix mz(2,2);                 // Declare a 2x2 matrix
  mz.put (z,1,1);                 // Set the last element to z
  }
```

Return Value:

Nothing.

See Also:

get, (), put_h

3.3.30 put_block

Usage:

```
#include <matrix.h>
void matrix::put_block(int row, int col, matrix& mx);
```

Description:

The function ***put_block*** sets values of a specified matrix sub-block to the values of the input matrix. The sub-block has the same dimensions as the input matrix and it values are copied into the matrix beginning at the row specified by row and the column specified by col.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1(10,10);              // Declare a 10x10 matrix
  matrix mx2(2,3);                // Declare a 2x3 matrix
  mx1.put_block(1,2,mx2);         // Fill 2x3 block of mx1 with mx2 starting at 1,2
  }
```

Return Value:

Nothing.

See Also:

put, (), put_h, get, put_block

3.3.31 put_h

Usage:

```
#include <matrix.h>
void matrix::put_h(complex& z, int i, int j);
```

Description:

The function ***put_h*** simultaneously sets the value of the matrix at position i,j and the value of the matrix element j,i to the conjugate of the i,j value. Thus the matrix is treated as if it were Hermitian.The columns and rows are numbered from 0 to cols-1 and 0 to rows-1.

Example:

```
#include <gamma.h>
main()
  {
  complex z;                          // Declare a complex number z
  matrix mx(2,2);                     // Declare a 2x2 matrix
  mx.put_h (z,0,1);                   // Set the 0,1 element to z & the 1,0 element to z*
  }
```

Return Value:

Nothing, the function is void.

See Also:

get, (), put

3.3.32 rank

Usage:

#include <matrix.h>
int rank (matrix& mx);

Description:

The matrix function ***rank*** calculates the rank of a matrix. A matrix rank is the number of linear independent rows or columns.

Return Value:

An integer is returned

See Also:

3.3.33 rows

Usage:

#include <matrix.h>
int matrix::rows ( );

Description:

The function ***rows*** returns the number of rows in the matrix.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx;                          // Declare a matrix mx
  cout << mx.rows();                  // Print the number of rows to standard output
  }
```

Return Value:

An integer is returned

See Also:

cols

3.3.34 set_hermitian

Usage:

#include <matrix.h>
void matrix::set_hermitian (hermitian_type h = _hermitian);

Description:

Converts the matrix to hermitian (or non_hermitian) without testing whether it is possible without loss of data.

Example:

```
#include <gamma.h>
```

```
main()
  {
  matrix a;
  cin >> a;
  a.set_hermitian();
  }
```

Return Value:

nothing

See Also:

check_hermitian, stored_hermitian, test_hermitian

3.3.35 set_type

Usage:

```
#include <matrix.h>
void matrix::set_type (matrix_type t);
```

Description:

The function ***set_type*** internally sets a matrix to a new matrix type. This means, fills in the zero's which where not stored in the original matrix or removes elements which are assumed to be zero in the new matrix. This may result in some lost data, if the matrix doesn't fit in the new type.

Example:

```
#include <gamma.h>
main()
  {
  matrix a(2,2);
  a.set_type (d_matrix_type)
  }
```

Return Value:

Nothing

See Also:

check_type, stored_type, test_type, matrix_type

3.3.36 stored_hermitian

Usage:

```
#include <matrix.h>
hermitian_type stored_hermitian ();
```

Description:

Returns _hermitian if the matrix is stored as a hermitian matrix, this means only the upper half of the matrix is stored (currently only true for i_matrix).

Example:

```
#include <gamma.h>
main()
  {
  matrix a;
  if (_hermitian=a.stored_hermitian())
    {cout << “matrix is stored as hermitian matrix”);
  }
```

Return Value:

Flag whether the matrix is stored hermitian or nor

See Also:

hermitian_type, check_hermitian, set_hermitian, test_hermitian

3.3.37 stored_type

Usage:

#include <matrix.h>

matrix_type matrix::stored_type();

Description:

The function ***stored_type*** returns how the matrix is currently stored in memory. It has no effect on the matrix itself. Current matrix type supported in GAMMA are listed at the end of this chapter, see page 84.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx(2,2);                          // Construct a matrix, 2x2
  if (i_matrix_type==mx.stored_type())     // See if mx is an identity matrix
      cout << “An identity matrix”;
  }
```

Return Value:

The matrix type

See Also:

check_type, set_type, test_type, matrix_type

3.3.38 tensor_product

Usage:

```
#include <matrix.h>
matrix tensor_product (matrix& mx1, matrix& mx2);
```

Description:

The function ***tensor_product*** calculates a new matrix which is the tensor product of the two input matrices.

$$mx3 = mx1 \otimes mx2$$

where the elements of the new matrix $mx3$ are given by

$$mx3(i, j) = mx1(i/rows, j/cols)mx2(i\mathrm{mod}rows, j\mathrm{mod}cols)$$

with $cols$ the number of columns of $mx2$ and $rows$ the number of rows of $mx2$. The resultant matrix will thus have the size (rows(mx1)*rows(mx2), cols(mx1)*cols(mx2)).

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1,mx2,mx3;                    // Declare three arrays
  mx3 = tensor_product(mx1,mx2);         // Set the mx3 to the tensor product mx1 ⊗ mx2
  }
```

Return Value:

The new matrix.

See Also:

3.3.39 test_hermitian

Usage:

```
#include <matrix.h>
hermitian_type test_hermitian (double d = 0.001);
```

Description:

Returns _hermitian if the matrix is a Hermitian matrix, this means

$$\forall(0 < i, j < rows - 1 \rightarrow \left|a_{ij} - \overline{a_{ji}}\right| < d) .$$

Example:

```
#include <gamma.h>
main()
  {
  matrix a;
  if (_hermitian=a.test_hermitian(0.1))
    {cout << “matrix is a hermitian matrix”);
  }
```

Return Value:

Flag whether the matrix is Hermitian or not.

See Also:

hermitian_type, check_hermitian, set_hermitian, stored_hermitian

3.3.40 test_type

Usage:

```
#include <matrix.h>
matrix_type matrix::test_type (matrix_type t, double d = 0.001);
```

Description:

The function ***test_type*** tests if it is possible to internally store the matrix as a matrix of type t ***without*** loss of data. The factor d may be optionally given as an argument such that elements with a norm smaller than d are assumed to be zero. Current matrix types in GAMMA are given at the end of this chapter, see page 84.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx(2,2);                          // Declare a matrix, 2x2
  if (mx.test_type(d_matrix_type))         // Check if mx can be maintained as a diagonal array
    cout << "Matrix can be converted to a diagonal matrix";
  }
```

Return Value:

Matrix type.

See Also:

check_type, set_type, stored_type, matrix_type

3.3.41 times_adjoint

Usage:

```
#include <matrix.h>
matrix times_adjoint (matrix& mx1, matrix& mx2);
```

Description:

The function ***times_adjoint*** returns the product of the first matrix input with the adjoint of the second matrix input. This is given mathematically as

$$mx3 \; = \; mx1 \times (mx2)^{\dagger} \; = \; mx1 \times (mx2^*)^T$$

or equivalently

$$mx3(i,j) \; = \; \sum_{k=0}^{rows-1} mx1(i,k) \times [mx2(k,j)]^{\dagger} \; = \; \sum_{k=0}^{rows-1} mx1(i,k) \times mx2^*(j,k)$$

Use of this function is computationally faster and uses less memory than performing the same mathematics using the adjoint function and the multplication function.

Return Value:

returns a matrix.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1, mx2, mx3;                // Declare three matrices
  mx3 = times_adjoint(mx1,mx2);        // Set mx3 to be mx1 * [mx2]†
```

See Also:

adjoint, adjoint_times

3.3.42 trace

Usage:

```
#include <matrix.h>
complex trace (matrix& mx);
complex trace (matrix& mx1, matrix& mx2);
```

Description:

The function ***trace*** is provided for obtain the trace of a matrix or the trace of a matrix produce. With one matrix given as an argument the function returns the trace as defined by[1]

$$Trace(mx) = \sum_{i=0}^{rows-1} mx_{ii}.$$

When two matrices are input as arguments ***trace*** returns the trace of the product of the two, as given by

$$Trace(mx1 \times mx2) = \sum_{i=0}^{rows(mx1)-1} \sum_{j=0}^{cols(mx1)-1} mx1(i,j)mx2(j,i)$$

The algorithm which computes this trace is faster and consumes less memory than performing the entire matrix multiplication followed by a trace calculation on the resultant matrix.

Return Value:

A complex number is returned.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1, mx2, mx3;                // Declare three matrices
  complex z;                           // Declare a complex number
  z = trace(mx1) + trace(mx2,mx3);     // Set z to be Tr(mx1) + Tr(mx2*mx3)
  }
```

1. Keep in mind that the indexing in GAMMA follows the standard of C and C++ in that the first element is element 0.

See Also:

rank, det

3.3.43 transpose

Usage:

#include <matrix.h>
matrix transpose (matrix& mx);

Description:

The matrix function ***transpose*** returns a matrix which is the transpose of the input matrix. This is given mathematically as

$$mx_{out} = mx_{in}^T$$

or equivalently

$$mx_{out}(i, j) = [mx_{in}(i, j)]^T = mx_{in}(j, i)$$

Return Value:

A matrix is returned.

Example:

```
#include <gamma.h>
main()
  {
  matrix mx1, mx2;                          // Declare two matrices
  mx2 = transpose(mx1);                     // Set mx2 to be mx1^T
```

See Also:

adjoint

3.3.44 ~matrix

Usage:

#include <matrix.h>
matrix::~matrix ();

Description:

This is the matrix destructor. It destroys a matrix and deallocates any memory that the matrix utilized. This function is normally called automatically by the various routines which use matrices and automatically by GAMMA programs when a matrix goes out of scope.

Return Value:

nothing

See Also:

matrix

3.4 Description

The class ***matrix*** provides most of the functions GAMMA uses to create and manipulate matrices. Transparent to the user, class matrix internally supports several different matrix types[1]. It is currently possible to store a matrix as

1 A full matrix (n_matrix), this is with all n*m elements,

2 A diagonal matrix (d_matrix), this is with only the diagonals stored and all off-diagonals assumed to be zero.

3 An identity matrix (i_matrix), this is only the size and all diagonals assumed to be one, all off-diagonals to be zero.

4 A Hermitian matrix (h_matrix), contains upper diagonal (& diagonal) elements only.

The principal advantage is, of course, that calculations with zero's can be neglected. For example, the following table indicates the relative time for matrix multiplication of n*n matrices

Compuation Time Order for Matrix Multiplication

	n_matrix	d_matrix	i_matrix
n_matrix	$O(n^3)$	$O(n^2)$	O(1)
d_matrix	$O(n^2)$	O(n)	O(1)
i_matrix	O(1)	O(1)	O(1)

Table 19-1 Relative computation for matrix multiplication of GAMMA arrays.

This computational savings occurs not only for common algebraic manipulations (+, -, *, /) but also in more complex functions which can take advantage of specific matrix structures (exponentiation, diagonalization, inversion, etc.). Furthermore, there is a savings in memory as well, obviously zeros are not stored when the matrix structure is known. There is a small overhead in GAMMA in that each matrix stores its type and that type checking is done during computations. However, in all but 2x2 arrays, the advantages far outweigh this overhead for virtually all calculations.

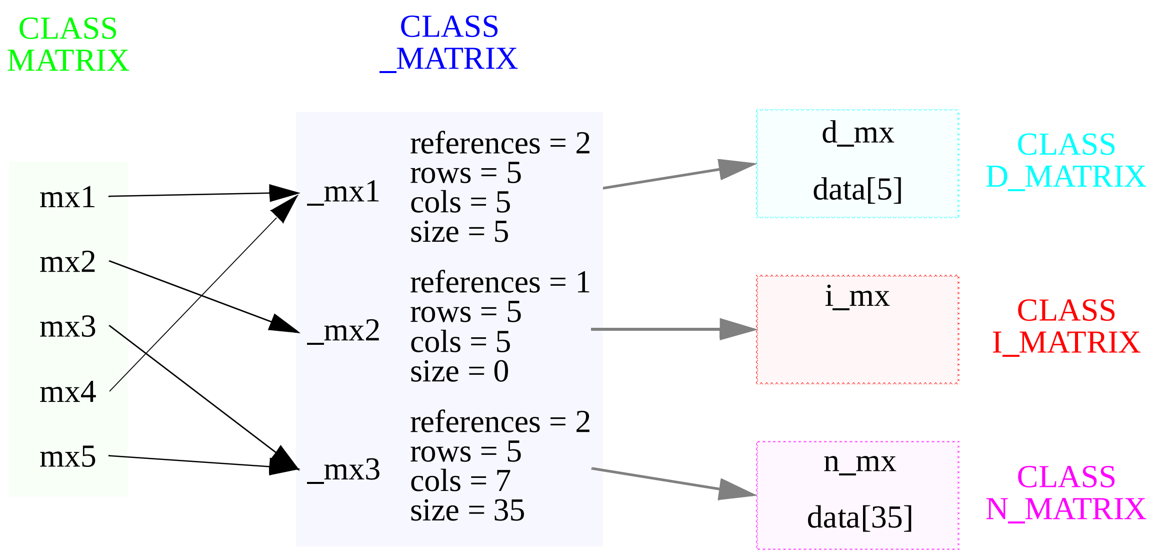
Keep in mind that conversion between the different matrix types is normally done automatically, the GAMMA user need not be concerned about this. If the user wants to insure that matrices are stored properly there are routines both to test whether a matrix is of a given type and to force a matrix to be of a given type (set_type, test_type, check_type, stored_type).

1. Currently, GAMMA uses only complex double precision arrays. This will change as GAMMA is extended.

3.5 Implementation

The class ***matrix*** contains merely a pointer an element of the class ***_matrix***. In turn, the class ***_matrix*** serves as the base class for individual matrix types in GAMMA. Examples of individual matrix classes (derived from ***_matrix***) are n_matrix, d_matrix and i_matrix for normal, diagonal, and identity matrices respectively. Thus, each matrix will normally point to a member of one of these derived classes. The following figure depicts this structure.

Implementation of Matrix Class Hierarchy

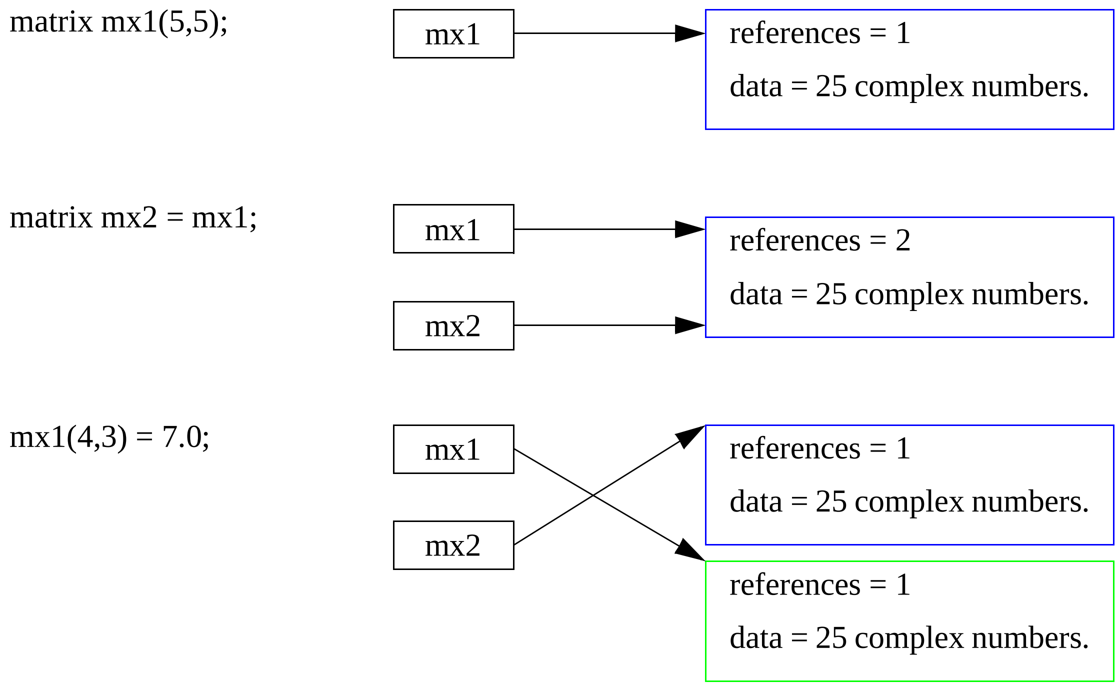


In this figure five matrices, mx1 - mx5, are actually 5 pointers to ***_matrix*** quantities. Since the matrices mx1 and mx4 are identical they point to the same ***_matrix***, _mx1. Similarly, matrices mx3 and mx5 are identical and point to the same ***_matrix***, _mx3. Each ***_matrix*** contains 4 integers which track how many "copies" of the matrix exist (***references***), the number of rows (***rows***) and columns (***cols***), and the size of the matrix data (***size***). Since ***_matrix*** serves as base class for individual matrix types, each ***_matrix*** is actually one of its derived classes and may contain the actual matrix elements (in complex array ***data***). Thus _mx1 in the figure is actually a ***d_matrix*** (diagonal matrix) which is pointed to by the two matrices mx1 and mx4. The diagonal array(s) are 5x5 and have 5 data points stored. Similarly, _mx2 is an ***i_matrix*** (identity matrix) pointed to by ***matrix*** mx2. This matrix is also 5x5, but it does not store any data since we know what the elements of any identity matrix are. The last ***_matrix*** _mx3 is a general complex array which is 5x7 and all 35 elements are stored.

The assignment operation and copy operation only duplicate the reference to the real data and increment (or decrement) the counter of references to this data. If the matrix is modified, for example by modifying a specific element (mx(5,3)=12.0), and the reference counter is not equal to one then the data is first copied, the old reference counter is decremented and the new data set is marked as referenced once.

An example:

GAMMA Code Versus Matrix Storage



3.5.1 Algorithms

Most commands to a matrix are just forwarded to the corresponding class of the matrix pointed to. This is done by a virtual call.

Some functions can have two matrices as arguments. In this case one argument is taken to make a virtual call and the other element is turned over to the function as a pointer. Then the called function itself has to use virtual calls to decide of what type this matrix is.

For the operators +=, -=, /= and *= another problem occurs, the type of the matrix on the left side could change. Therefore it is not possible to make the virtual call through this matrix. It has to be done to the matrix on the right side, which will not change.

3.5.2 Diagonalization

The function diag uses the Householder-QR algorithm to find the eigenvalues of the matrix. The unitary transformations used to achieve the diagonal form are accumulated and constitute to the eigenbase of the matrix. A maximum of 10 iterations per eigenvalue is allowed, if this limit is exceeded the routine issues an error message and returns.

3.5.3 Matrix Types

Currently GAMMA internally recognizes the following matrix types.

n_matrix_type : normal matrices
d_matrix_type : diagonal matrices
i_matrix_type : identity matrices
h_matrix_type : Hermitian matrices

The reason for doing so is obvious, routines are written specifically to take advantage of these types. These values (in bold italic) may be used as arguments in some matrix functions.

Table 20: Unique Matrix Types

type	description	element
i_matrix	identity	complex
n_matrix	normal	complex
d_matrix	diagonal	complex
h_matrix	hermitian	complex

3.5.4 Matrix Hermitian Type

Currently GAMMA internally matains a flag which indicates whether or not the matrix is Hermitian, this is the data type hermitian_type. This flag is an enumeration with two values:

_hermitian = TRUE
non_hermitian = FALSE

This means the result of the functions test_hermitian, check_hermitian and stored_hermitian can be used in boolean expressions.