ivl documentation
Version 0.22

iva

June 20, 2013

Contents

Introduction

Ivl is a template C++ library for scientific processing and mathematics including image processing.

1 Fundamentals

As any mathematic book begins by defining the fundamental types and then extends them with various operations, so do we in this manual.

1.1 Simple C++ Expressions

All ivl functions and classes operate on objects of specific types. We will begin by enumerating the basic types of the mathematic objects to be used in our processing. These in our case would be all the numbers consisting of the sets of integers, reals, complex numbers and booleans.
The native types of C++ are used to represent these numbers. In particular, Table 1.1 shows all the C++ types that we use and the according numeric set they represent.

integers	`char`, `short`, `int`, `long long`
unsigned integers	`unsigned char`, `unsigned short`, `unsigned int`, `unsigned long long`
machine dependant integers	`size_t`, `ptrdiff_t`
machine independant integers	`int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`
real numbers	`float`, `double`
boolean numbers	`bool`
complex numbers	`std::complex<float>`, `std::complex<double>`, `std::complex<int>`, `std::complex<...>`

Table 1.1: *The numeric types of C++ that are used in ivl*

As we know, mathematic expressions based on the above types are supported natively by C++. For instance the addition between two numbers. To complicate things a little we may state that additions are allowed between different types as well. The addition between an integer and a double,

```
int a = 1;
double b = 2;
```

is done like this:

```
int y = a + b;
```

Furthermore, we will see more uses of the `operator +` which stands for the algebric addition. The meaning of this operator always stays the same however in ivl it is possible to use it on other objects as well.

1.2 Introducing ivl; the type `ivl::scalar`

Every function and class of ivl belongs to the `ivl` namespace.

```
#include <ivl/ivl>

using namespace ivl;
```

Simple integer, real, complex and boolean types can be used in ivl as they are. However they lack certain few extra properties of ivl that are not compatible with the basic types. To strengthen the basic C++ types with those extra properties we need to use the `ivl::_[.]` expression.
The expression `_[.]` converts any type `T` to `ivl::scalar<T>`. For example if we have a variable `x` of type `int`,

```
int x = 2;
```

We can get a corresponding `ivl::scalar<int>` with:

```
_[x]
```

The expression `_[x]` is essentially equivalent to `x` and it can be used like `x` since the value of `_[.]` is automatically converted back to its initial form. The backward conversion is done implicitly, when needed, by a conversion operator. In our example in which `x` is an `int`, the expression `_[x]` is converted back to `int` as shown below. The box under the line of the code shows the result of the program at the standard output.

```
std::cout << "_[x] = " << _[x] << std::endl;
std::cout << "_[x] + 1 = " << _[x] + 1 << std::endl;
```

```
_[x] = 2
_[x] + 1 = 3
```

In the above example the use of `_[.]` is redundant and practically meaningless. However, as we did mention, once converted from `x` the expression `_[x]` supports some extended ivl operations. One of them is the `operator ->*` which raises a number to a specified power.

```
std::cout << "x^3 = " << _[x] ->* 3 << std::endl;
```

```
x^3 = 8
```

Another reason for using `_[x]` is to explicitly state that `x` is a simple element as opposed to something else, i.e. an array. This will be shown useful later on.

1.3 Algebra on single elements

In C++ the evaluation of a simple expression may be written in the form:

```
<variable> = <expression>;
```

The standard C++ library already provides several math functions in the according headers. In other words C++ already supports a basic background for mathematic processing.

```
#include <cmath>
#include <iostream>

int main()
{
    using namespace std;

    double x = 2.0, y;
    y = exp(pow(x, 2) + sqrt(x));
    cout << "e^(x^2 + sqrt(x)) = " << y << endl;
}
```

```
e^(x^2 + sqrt(x)) = 224.576
```

Ivl is based on the assumption that the native C++ libraries are optimal and therefore their code is to be reused instead of it being rewritten into custom ivl functions. So ivl functions are based on the standard C++ `<cmath>` and `stl` library. However ivl extends that library by adding new overrides to the existing functions and by having its own new functions and constants. All single element math functions of ivl exist in the namespace `ivl::math` but they should be used through the namespace `ivl`. This is because they are redefined in the namespace `ivl` with more ways to be used than the standard mathematic expressions.
The following example introduces two sample additions of ivl. The function `sqr` and the constant `pi`.

```
#include <ivl/ivl>
#include <iostream>

int main()
{
    using namespace std;
    using namespace ivl;

    double y = 2.0 * sqr(pi);
    cout << "2.0 * pi^2 = " << y << endl;
}
```

```
2.0 * pi^2 = 19.7392
```

`isinf`(x)	*Returns true if* `x` *is infinite*
`isnan`(x)	*Returns true if* `x` *is not-a-number*
`sign`(x)	*Returns the sign of* `x` *multiplied by one*
`round`(x)	*Returns the floating number* `x` *rounded to the closest integer value*
`conj`(x)	*Returns the conjugate of* `x`
`angle`(x)	*Returns the angle of a complex number* `x`

Table 1.2: *Unary math functions of ivl that do not exist in standard C++*

The above constant `ivl::pi` is nothing else than the mathematics pi constant. The function `ivl::sqr`(x) returns the square of `x` which is equal to `x * x`. Several more unary functions unique to ivl are listed in Table 1.2.
Whenever possible ivl uses the same function names as standard C++ math when they have the same behaviour.
Some new overloads for the existing stl functions are added in ivl allowing functionality that was, purposely or not, missing. For example some functions that only existed for real numbers now exist for complex numbers too.

1.4 Operators between elements

The list of C++ operators and the list of ivl operators for elements is pretty much the same. Ivl includes the operator `->*` with highest priority for raising to a power.
Table 1.3 shows a list of all the ivl operators for elements.

1.5 Complex numbers in ivl

a = b	Assignment
a + b	Addition
a - b	Subtraction
+a	Unary plus
-a	Additive inverse
a * b	Multiplication
a / b	Division
a % b	Modulo
++a	Increment
a++	Increment after
--a	Decrement
a--	Decrement after
a == b	Equal to
a != b	Not equal to
a > b	Greater than
a < b	Less than
a >= b	Greater than or equal
a <= b	Less than or equal
!a	Logical NOT
a && b	Logical AND
a || b	Logical OR
~a	Bitwise NOT
a & b	Bitwise AND
a | b	Bitwise OR
a \^ b	Bitwise XOR
a << b	Bitwise left shift
a >> b	Bitwise right shift
a ->* b	Raise to power (a should be extended)
a += b	Addition assignment
a -= b	Subtraction assignment
a *= b	Mupliplication assignment
a /= b	Division assignment
a %= b	Modulo assignment
a &= b	Bitwise AND assignment
a |= b	Bitwise OR assignment
a \^= b	Bitwise XOR assignment
a <<= b	Bitwise left shift assignment
a >>= b	Bitwise right shift assignment

Table 1.3: *List of ivl element-wise operators*

As mentioned, ivl makes use of the class `std::complex<T>` for complex numbers. Some of the C++ math functions have been overloaded to extend support to complex numbers which they did not. Many trigonometric functions fall under this case. For example, the cosine function `ivl::cos`(x) is defined for a real but also for a complex argument as we can see below:

```
#include <ivl/ivl>

int main()
{
    using namespace std;
    using namespace ivl;

    complex<double> x(pi / 4, 1);
    cout << "cos(pi/4 + i) = " << cos(x) << endl;
```

```
cos(pi/4 + i) = (1.09112,-0.830993)
```

Instead of writing the full constructor when creating a complex number one may also use the imaginary unit `j` or the synonym `i`. These two constants are defined in the namespace `ivl::math` and are equal to `std::complex<double>(0, 1)`.

```
    using ivl::math::i;
    cout << "cos(pi/4 + i) = " << cos(pi/4 + i) << endl;
```

```
cos(pi/4 + i) = (1.09112,-0.830993)
```

In this second approach in our example, we see that the number

```
pi / 4 + i
```

is essentially

```
std::complex<double>(pi / 4, 1)
```

Complex numbers also support the extended ivl operators, for instance

```
    cout << "(1 + i)^2 = " << _[1.0 + i] ->* 2 << endl;
```

```
(1 + i)^2 = (0,2)
```

```
}
```

1.6 Type Abstraction

Mathematic expressions can always be applied to user-defined symbolic items as long as the meaning for the various operations are defined for them. C++ makes no exception for this as user-defined classes support overloaded operators and functions so that they can be used as objects in complicated mathematic expressions. Ivl furthermore supports custom classes as elements. All functions and primitives like arrays etc have abstract element types.
This means that the various operations of ivl accept elements of any class and are not limited to the basic C++ types.
This is achieved using template programming and the basic principles of the stl library are followed.
We will use a custom class as an example where we will call the ivl function `min(x, y)` which returns the minimum between two elements x and y. Since min uses the comparison operator ¡ between the two objects this operator needs to be defined for the objects x and y for the code to compile. Even though the function `ivl::min` already exists in stl as `std::min` does exactly the same, it was merely used as a trivial example and the general idea applies for every ivl function.

```
//Tutorial sample samples/manpage/s2.cpp
#include <ivl/ivl>
#include <iostream>

// a custom class that holds two integers
class B
{
public:
    int key, val;
    B(int k, int v) : key(k) , val(v) { }

    // the class elements are compared by comparing the first
   integer
    bool operator <(const B& o) const { return key < o.key; }
};

// we overload the output of the class members to a stream
std::ostream& operator <<(std::ostream& out, const B& b)
{
    return out << "( key = " << b.key << ", val = " << b.val << "
   )";
}

int main()
{
    using namespace std;
    using namespace ivl;

    B x(1, 5);
    B y(2, 15);
    cout << "minimum of x, y = " << ivl::min(x, y) << endl;
}
```

```
minimum of x, y = ( key = 1, val = 5 )
```

Among other operations the `_[.]` expression is also valid for elements of custom class. The extended ivl operators apply to them the usual way. Of course in order to be able to raise a custom class object x to the power y using `_[x] ->*(y)` the underlying power function needs to be overriden. In our case this function is `ivl::pow(x, y)`.

1.7 Type Evolution

Since the types of the participating elements are the basis of any operation, we insist on a few more properties of these types.
There are cases when the types that we use are unknown. These might be the cases of template programming. In template programming we may need to use information from template types to our aid. Particularly we may need to produce new types `Y` based on certain properties of template types `T`. The namespace `ivl::types` includes tools to this end. We use the name *type evolution* to indicate the production of types based on already known types from which they evolve.
One example is to produce a floating-point number of type `Y`, depending on a type `T`, i.e., `Y` is complex only when `T` is complex itself.
The struct `to_floating`, defines a dependant type as below.

```
typename types::to_floating<T>::type
```

Table 1.4 shows many of the possible type transitions that occur with `to_floating`. In other words, `to_floating<int>::type` is a `double` and so on.
An example where we may use `to_floating` is to write a template function that returns the mean of two numbers. The mean of two `ints` may be a decimal number, so we want it to be a `double`. But the mean of two `floats` is a `float`. Instead of writing several specializations to achieve this we use `to_floating` and we only write one function. The following example has almost the same code as the actual `ivl::mean`.

```
template<class T>
typename types::to_floating<T>::type mean(T a, T b)
{
```

original type	$\rightarrow$ *resulting type*
`int`	$\rightarrow$ `double`
`float`	$\rightarrow$ `float`
`double`	$\rightarrow$ `double`
`std::complex<float>`	$\rightarrow$ `std::complex<float>`
`std::complex<int>`	$\rightarrow$ `std::complex<double>`

Table 1.4: *Type transitions made when using* `ivl::types::to_floating<T>` *on a type.*

`to_complex<T>`	*Changes the type to a complex number*
`to_real<T>`	*Changes any complex number type to real*
`to_signed<T>`	*Changes any unsigned type to signed*
`to_unsigned<T>`	*Changes any signed type to unsigned*
`promote<T>`	*Makes the type larger in size if possible*

Table 1.5: *Other type evolution classes of* `ivl::types`

```
    typedef typename types::to_floating<T>::type result_type;

    return (result_type(a) + result_type(b)) / 2.0;
}

int main()
{
    using ivl::math::i;
    // The produced type is double
    cout << "mean(1, 2) = " << mean(1, 2) << endl;
```

```
mean(1, 2) = 1.5
```

```
    // The produced type is std::complex<double>
    cout << "mean(1+i, 2.5-2i) = "
        << mean(1.0 + i, 2.5 - 2.0 * i) << endl;
```

```
mean(1+i, 2.5-2i) = (1.75,-0.5)
```

```
}
```

Other similar structs useful for type evolution are shown in Table 1.5
Note that the usage of the above types is based solely on template code, which only runs at compile time. All that the mentioned structs do is provide type definitions, not instructions. As a result they do not affect performance however complicated they might be.

2 Arrays

2.1 Array; a class for storing linear data

The ivl container for one dimensional arrays that similates std::vector is called ivl::array. We will make a superficial presentation of `ivl::array` in this section focusing more on its use in algebra. The details of the class are explained in later sections. Like all other classes `ivl::array` is a template class. It has two template parameters however only the first one will be discussed for now since the second one is optional.
The first template argument T of `ivl::array<T>` is the class of the elements of the array. There are a few ways to construct an `ivl::array<T>`. One way is from an existing C array. This is an example.

```
#include <ivl/ivl>

using namespace ivl;
using namespace std;

int main()
{
    double values[] = { 0.1, 0.2, 0.3, 0.2, 0.1 };
    array<double> a(values);
```

A synonym to the two above lines of code can be written on a single line using `ivl::arr`. The template function `ivl::arr<T>` constructs an array with a few given elements, while detecting the type.

```
    array<double> a2 = arr( 0.1, 0.2, 0.3, 0.2, 0.1 );
```

An array can also be constructed via copy from another array, of the same or different type.

```
    array<double> a3 = a2;
```

Arrays do not need to be initialized at first. An uninitialized array has zero size at first. At any point assigning it means copying to it. When assigning to an array any values it holds are lost. Also its size may be changed accordingly.

```
    array<double> a4;
    a4 = a;
```

2.2 Streaming an array

An ivl array can be streamed to any C++ stream including the cout. The result is the array elements formatted.

```
    cout << a << endl;
```

```
[       0.1000        0.2000        0.3000        0.2000        0.1000 ]
```

2.3 Algebra on arrays

Now suppose that a was of a standard C++ type. As mentioned above the evaluation of a simple expression may be written in the form:

```
<variable> = <expression>;
```

It is possible to evaluate expressions between ivl arrays the exact same way as if they were standard types.
For instance the C++ expression for adding `1` to `a` would be `a + 1`. and the return type of the expression would be the same as the type of a. Here is the example of this addition.

```
    array<double> y;
    y = a + 1;

    cout << y << endl;
```

```
[       1.1000        1.2000        1.3000        1.2000        1.1000 ]
```

The number is added to every element of `a`. We call this an element wise operation because it is repeated for each element. Doing the same with the number on the left side of the expression requires a scalar conversion

```
    y = _[1] + a;
    cout << y << endl;
```

```
[       1.1000        1.2000        1.3000        1.2000        1.1000 ]
```

Obviously this produces the same result. We can also have binary operations between two arrays. The constraint is that the size of the arrays must be the same.

```
array<double> b = arr(1.1, 1.2, 1.3, 1.2, 1.1);
cout << "a = " << a << endl << "b = " << b << endl;
```

```
a = [      0.1000      0.2000      0.3000      0.2000      0.1000 ]
b = [      1.1000      1.2000      1.3000      1.2000      1.1000 ]
```

The addition of two numbers `a` and `b` in C++ is done by the expression `(a + b)`.

```
y = a + b;
cout << "a+b = " << y << endl;
```

```
a+b = [      1.2000      1.4000      1.6000      1.4000      1.2000 ]
```

As we can see the addition is an element-wise expression, meaning that the addition is done on all the elements of both arrays, producing an array with the individual results.
Finally a last way to make an addition is to do it in-place with the `operator +=`.

```
cout << "y = " << y << endl;
cout << "a = " << a << endl;
```

```
y = [      1.2000      1.4000      1.6000      1.4000      1.2000 ]
a = [      0.1000      0.2000      0.3000      0.2000      0.1000 ]
```

```
y += a;
cout << "y = " << y << endl;
```

```
y = [      1.3000      1.6000      1.9000      1.6000      1.3000 ]
```

We have focused on the addition operation. All the ivl element-wise operators shown in Table 1.3 can be applied to arrays. The same holds for functions that operate on elements. The main principle is that when an operation that is defined for a single element is be applied to an array, the operation is done to all its elements.
Also we may note that arrays of different element types can be combined in expressions the same way different types of C++ numbers can be mixed in them.
The sample below contains the line `y = sin(y)`. Ivl always does overlapping checks when assigning to arrays making such expressions safe. They may also be done optimally when this is possible. Here the ivl::sin function is applied to the array `y`.

```
    cout << "y = " << y << endl;
    y = sin(y);
    cout << "y = " << y << endl;
```

```
y = [      1.3000      1.6000      1.9000      1.6000      1.3000 ]
y = [      0.9636      0.9996      0.9463      0.9996      0.9636 ]
```

In C++ we are allowed to combine different types in expressions, e.g. we can combine double and int:

```
    {
        double a = 0.5;
        int c = 1;
        double y = a + c;
        cout << "y = " << y << endl;
    }
```

```
y = 1.5
```

The same rule applies to ivl arrays. Considering the previous example, we can use the addition operator between an array of double and an array of int. The same thing applies for operations between array and scalar. This is the analogous example:

```
    array<int> c = arr(1, 2, 3, 2, 1);
    y = a + c;
    cout << "y = " << y << endl;
```

```
y = [      1.1000      2.2000      3.3000      2.2000      1.1000 ]
```

We end this section by adding that ivl expressions can be nested as many times as we want in any way we want. In fact nesting expressions is the preferred way to do a series of computations on arrays. That is because nesting is implemented in a way which enables a possible optimization of the combined expression, which is done per element. Here is an example of a nested expression:

```
    y = log(sin(a + power(a, c) ->* 2) - sin(c));
    cout << "y = " << y << endl;
```

```
y = [     -2.2093     -1.6083     -1.2167     -1.6083     -2.2093 ]
```

2.4 Array non-element operations

The element operations we discussed above are similarly defined for single elements and arrays. Apart from these operations, there is a large set of functions involving arrays in ivl. The function `sum` returns the sum of all elements of an array.

```
    cout << "y = " << y << endl;
```

```
y = [     -2.2093      -1.6083      -1.2167      -1.6083      -2.2093 ]
```

```
    cout << "sum(y) = " << sum(y) << endl;
```

```
sum(y) = -8.852
```

There are quite a few many-to-one functions for arrays. For example the `ivl::min`. This function is essentialy an override of the `std::min` that returns the minimum of two elements. The principle of using the names of C math functions as overrides also holds for arrays. The function `ivl::min`(a) where `a` is an `array<T>` returns the minimum element of that array:

```
    //to resolve the ambiguity between ivl::min and std::min
    using ivl::min;
    cout << "min(y) = " << min(y) << endl;
```

```
min(y) = -2.209
```

The library of ivl functions is large. However we will see more about ivl functions when tuples are discussed.

```
}
```

2.5 Ranges

Arrays in ivl exist in several variants. Each have different properties. One significant variant is an `ivl::range<T>`. This class represents an array with elements in a specific range, e.g. $1, 2, \ldots, n$. Ranges are arrays but do not take space to store all the elements. Constructing a range of `double` is done like this:

```
range<double> r(1.0, 5.0);
std::cout << "r = " << r << std::endl;
```

```
r = (1:5)
```

```
array<double> a = r;
std::cout << "a = " << a << std::endl;
```

```
a = [       1.0000       2.0000       3.0000       4.0000       5.0000 ]
```

The element type of a range may be of any type. Ranges may also have a step, positive or negative. For example a range of `int` with a step 2:

```
range<int> r2(1, 2, 10);
std::cout << "r2 = " << r2 << std::endl;
```

```
r2 = (1:2:10)
```

```
array<int> a2 = r2;
std::cout << "a2 = " << a2 << std::endl;
```

```
a2 = [ 1 3 5 7 9 ]
```

```
std::cout << array<double>(range<double>(5, -2, -3)) << std::endl;
```

```
[       5.0000       3.0000       1.0000      -1.0000      -3.0000 ]
```

Ranges can be created by the `rng` function which constructs a `range<T>` by autodetecting the type of elements. The following examples have the same result as the previous ones:

```
a = rng(1.0, 5.0);
std::cout << "a = " << a << std::endl;
```

```
a = [       1.0000       2.0000       3.0000       4.0000       5.0000 ]
```

```
a2 = rng(1, 2, 10);
std::cout << "a2 = " << a2 << std::endl;
```

```
a2 = [ 1 3 5 7 9 ]
```

2.6 Indices

The type `size_t` has a special meaning in ivl: To hold indices. This type has size equal to the pointer size, i.e. 32-bit for 32-bit executables, 64-bit for 64-bit executables etc. It has the ability to hold any size that could possibly fit into the program memory and thus is ideal for storing indices. Similarly it is ideal for storing sizes.
There are times when an array of sizes or indices is needed in order to be supplied to certain functions. Thus there is a shorthand for constructing an `array<size_t>` named `idx`. The function `idx` is practically the same as the function `arr<size_t>`.

```
std::cout << "idx(1, 3, 4, 5) = " << idx(1, 3, 4, 5) << std::endl;
```

```
idx(1, 3, 4, 5) = [ 1 3 4 5 ]
```

3 Multidimensional arrays

3.1 The class `array_2d`

The class for representing matrices or two-dimensional arrays is `array_2d<T>`. This class is very similar to an `array<T>` in the way it is used. To construct an `array_2d<T>` with initial data the sizes, which are the rows and the columns of the array, are required. Given the number of rows and columns and a C array with the data column by column we may construct an `array_2d` as shown below:

```
double a_v[] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,
   10.0, 11.0, 12.0 };
array_2d<double> a(3, 4, a_v);
cout << a << endl;
```

```

     1.0000      4.0000      7.0000     10.0000
     2.0000      5.0000      8.0000     11.0000
     3.0000      6.0000      9.0000     12.0000
```

A more elegant way to construct an `array_2d<T>` is by using concatenation with conjuction to the `row`() and `col`() function for creating row and column vectors accordingly.

```
array_2d<double> a2 =
    row(1, 2, 3, 4)() &
    row(5, 6, 7, 8);

cout << a2 << endl;
```

```
1.000    2.000    3.000    4.000
5.000    6.000    7.000    8.000
```

There are many other ways to construct an `array_2d<T>`. All the constructors are listed in the ivl reference manual.

3.2 Algebra on two-dimensional arrays

The usual element functions and operators apply to two dimensional arrays as well. For example:

```
array_2d<double> y = a2 + 1;
cout << "a2 = " << endl << a2 << endl;
cout << "a2 + 1 = " << endl << y << endl;
```

```
a2 =


     1.0000       2.0000       3.0000       4.0000
     5.0000       6.0000       7.0000       8.0000


a2 + 1 =


     2.0000       3.0000       4.0000       5.0000
     6.0000       7.0000       8.0000       9.0000
```

Two-dimensional arrays may also be combined together in element operations, provided they have the exact same size.

```
array_2d<double> z;
z = a2 + y;
cout << "z = " << endl << z << endl;
```

```
z =


     3.0000       5.0000       7.0000       9.0000
    11.0000      13.0000      15.0000      17.0000
```

3.3 Matrix Operators

Besides the element operations there are certain operators that are specific to matrices. Suppose we want to multiply matrices `a` and `a2` from the previous example. We can see that the number of columns of `a` is equal to the number of rows of `a2`:

```
cout << "a = " << endl << a << endl;
cout << "a2 = " << endl << a2 << endl;
```

```
a =

     1.0000      4.0000      7.0000     10.0000
     2.0000      5.0000      8.0000     11.0000
     3.0000      6.0000      9.0000     12.0000

a2 =

     1.0000      2.0000      3.0000      4.0000
     5.0000      6.0000      7.0000      8.0000
```

The `operator *` is already defined for element multiplication, so we need to use something that is distinct. For example:

```
cout << a2 * a2 << endl;
```

```

     1.0000      4.0000      9.0000     16.0000
    25.0000     36.0000     49.0000     64.0000
```

In order to perform matric multiplication we use the `operator ()` on the left matrix followed by an `operator *`. The parenthesis operator, `operator ()`, when applied to an `array_2d<T>` creates a temporary array object changing the behaviour of operators to behave as matrix operators. That way, the expression `a() * a2` stands for matrix multiplication as opposed to element multiplication:

```
cout << a() * a2 << endl;
```

```

    95.0000    117.0000    139.0000     56.0000
   109.0000    135.0000    161.0000     70.0000
   123.0000    153.0000    183.0000     84.0000
```

`a() || b`	Horizontal Concatenation
`a() & b`	Vertical Concatenation
`a() * b`	Matrix Multiplication
`a() / b`	Matrix Division
`a() | b`	Matrix Left-Division
`a() % b`	Matrix Modulo
`a() ->* b`	Matrix Power
`!a()`	Matrix Transpose
`*a()`	Matrix Conjugate Transpose
`a(!_)`	(discuss) Matrix Transpose
`a(*_)`	(discuss) Matrix Conjugate Transpose

Table 3.1: *List of ivl matrix operators*

If we want to do a matrix operation after another matrix operation we need to add the `operator()` after the second operand:

```
f = a() * a2() * a3;
```

and this could go on for as many matrices as we want, paying attention to the operator precedence. It is also possible to parenthesize the whole expression to obtain the same result:

```
f = (a() * a2())() * a3;
```

Table 3.1 shows a list of all the ivl matrix operators.
Note that to multiply $a' * a$ the use of `operator()` is required twice. First time it is required for the transpose operation and second time for the multiplication:

```
f = !a()() * a;
```

3.4 The class `array_nd`

When we need an array with a specified number of dimensions we use the class `array_nd<T>`. The class `array_nd<T>` is similar to `array_2d<T>` and in fact is its base class as it is shown later when arrays are discussed in more detail.

The difference from `array_2d<T>` in the construction of an `array_nd<T>` is that we supply an `array<size_t>` of the dimensions instead of the rows and the columns. The dimensions are given in order. The first dimension is the rows, then the columns, then the third dimension, then the fourth and so on. The linear data is "folded" accordingly, i.e. it is supplied and stored column by column then by the third dimension, then by the fourth and so on. We give an example of an `array_nd<double>` with three dimensions.
Here, instead of supplying the data from a C array we use the more elegant `ivl::range`. This is not a unique property of `array_nd<double>` as every ivl array can be constructed with a size and another ivl array.

```
array_nd<double> a(idx(3, 3, 2), rng(1.0, 18.0));
cout << a << endl;
```

```

[0]
     1.0000       4.0000       7.0000
     2.0000       5.0000       8.0000
     3.0000       6.0000       9.0000
```

An `array_nd<T>` with two dimensions can be copied to or from an `array_2d<T>`.

```
array_2d<double> b(3, 3, rng(1.0, 9.0));
a = b;
cout << a << endl;
```

3.5 Algebra on `array_nd`

The properties of `array_nd` are similar to `array` and `array_2d`. In element based algebra `array_nd` can be mixed with `array_2d` when the shape is the same. Operations with single elements is done as in `array` and `array_2d`. Here we raise the elements of an `array_nd` to the power 2 instead of repeating the usual example with `operator +`:

```
cout << a ->* 2 << endl;
```

```
[0]
     1.0000       4.0000       7.0000
     2.0000       5.0000       8.0000
     3.0000       6.0000       9.0000

[1]
    10.0000      13.0000      16.0000
    11.0000      14.0000      17.0000
    12.0000      15.0000      18.0000
```

3.6 Nested arrays

We may want to construct a type of multiple-sized arrays in a single array. The principle that is followed in ivl is that the element type of an array is abstract and may as well be another array.
More than being allowed the type `array<array<T> >` has various properties that arise from the properties of `ivl::array`.

```
array<double> a = rng(1.0, 4.0);
array<array<double> > b;
b = a;
cout << b << endl;
```

```
[ [       1.0000 ] [       2.0000 ] [       3.0000 ] [       4.0000 ] ]
```

```
b += 1;
cout << b << endl;
```

```
[ [       2.0000 ] [       3.0000 ] [       4.0000 ] [       5.0000 ] ]
```

```
b += a;
cout << b << endl;
```

```
[ [       3.0000 ] [       5.0000 ] [       7.0000 ] [       9.0000 ] ]
```

```
cout << cast<array<double> >(b) << endl;
```

```
[ [ [       3.0000 ] ] [ [       5.0000 ] ] [ [       7.0000 ] ] [ [
  9.0000 ] ] ]
```

```
cout << cast<int>(b) << endl;
```

```
[ [ 3 ] [ 5 ] [ 7 ] [ 9 ] ]
```

4 Subscripting arrays

4.1 Accessing elements of an array

The class `array<T>` allows us to access its elements using subscripts. The `operator []` takes a size_t argument and returns a reference to the according element at the specified index. The zero-index C convention is used for indices. Hence the elements of an array of size n are indexed $0, 1, \ldots, n-1$. Besides being indexed in order the elements of an `array<T>` are also stored in order in the memory.

```
array<double> x = rng(1.0, 8.0);
cout << x << endl;
```

```
[       1.0000       2.0000       3.0000       4.0000       5.0000
6.0000       7.0000       8.0000 ]
```

```
cout << x[0] << endl;
```

```
1
```

Trying to access anything outside the array bounds in Release mode is like accessing an illegal pointer address. When in Debug mode ivl throws an `ivl::exception`.

```
array<double> x = rng(1.0, 8.0);
cout << x[8] << endl;
```

```
s2: /usr/local/include/ivl/details/array/impl/specialization/
array_class.hpp:253: typename std::vector<T>::reference ivl::array
<T, S>::operator[](size_t) [with T = double; OPTS = ivl::data::mem<>;
 typename std::vector<T>::reference = double&; size_t = unsigned int
]: Assertion ‘offset >= 0 && offset < length()’ failed.
```

The number of elements of an array is determined by the `.length()` member function. Taking advantage of the `.length()` member we may output all its elements like this:

```
for(int i = 0; i < x.length(); i++)
    cout << x[i] << " ";
cout << endl;
```

```
1 2 3 4 5 6 7 8
```

Generally the `ivl::array` behaves like an `std::vector`. However for the moment being we still hide the class members and the class itself to focus on the subscripting. In a following section we will get into thorough details about the array class and its family.

4.2 Subarray of an array

We may access a subset of elements of an array using a single subscript. When doing this, the requested subarray behaves like an array. One way of taking a subarray of an `array<T>` is with a `range`. We supply a range of the requested indices at the `operator []` to take a subarray with only those indices.

```
array<complex<double> > a = rng(1.0, 8.0);
std::cout << a << std::endl;
```

```
[ (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0) ]
```

```
std::cout << a[rng(2,4)] << std::endl;
```

```
[ (3,0) (4,0) (5,0) ]
```

```
std::cout << a[rng(0,2,4)] << std::endl;
```

```
[ (1,0) (3,0) (5,0) ]
```

Another way to get a subarray of an `array<T>` is to explicitly specify the requested indices in an `array<size_t>`. Repeating elements or going back and forth is also allowed:

```
std::cout << a << std::endl;
```

```
[ (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0) ]
```

```
std::cout << a[idx(0,1,3,5)] << std::endl;
```

```
[ (1,0) (2,0) (4,0) (6,0) ]
```

```
std::cout << a[idx(5,1,3,1,7,7,2)] << std::endl;
```

```
[ (6,0) (2,0) (4,0) (2,0) (8,0) (8,0) (3,0) ]
```

The subarrays taken with subscript are actually referenced elements. No copies are made and the subarray itself is writable. For example.

```
a[idx(5,1,3,1,7,7,2)] += 2.0 * ivl::math::i;
std::cout << a << std::endl;
```

```
[ (1,0) (2,4) (3,2) (4,2) (5,0) (6,2) (7,0) (8,4) ]
```

```
a[rng(0,2,4)] = -1.0;
std::cout << a << std::endl;
```

```
[ (-1,0) (2,4) (-1,0) (4,2) (-1,0) (6,2) (7,0) (8,4) ]
```

Subarrays are temporary objects. Getting a non-const reference to a temporary object is prohibited in C++. We want add an unary `operator *`, the dereference operator, in front of a subarray when writing to it, e.g.

```
*a[idx(5,1,3,1,7,7,2)]
```

This way we are certain that the subarray may be passed to functionsas a non-const reference. Passing subarrays to tuples also requires the dereference operator. Simple assignments or add-assignments etc. do not require this operator however we will use it always in our examples.
A special value for subarrays is `empty`. When assigning `empty<T>` to a subarray, all its elements are removed from the array. A synonym to `empty<T>` is the object `_`.

4.3 The subscript `all`.

When assigning a single element to an array the array resizes itself to fit a single element. In other words:

```
std::cout << a << std::endl;
```

```
[ (-1,0) (2,4) (-1,0) (4,2) (-1,0) (6,2) (7,0) (8,4) ]
```

```
a = 1.0;
std::cout << a << std::endl;
```

```
[ (1,0) ]
```

Subarrays are an exception as assigning them to a single value assigns the requested value to all the elements. When we want to assign a value to all the elements of an array we use the subscript `all`. A synonym to âll is the expression `*_`:

```
std::cout << a << std::endl;
```

```
[ (-1,0) (2,4) (-1,0) (4,2) (-1,0) (6,2) (7,0) (8,4) ]
```

```
a[*_] = 1.0;
std::cout << a << std::endl;
```

```
[ (1,0) (1,0) (1,0) (1,0) (1,0) (1,0) (1,0) (1,0) ]
```

The subscript `all` is also a useful way to view a multi-dimensional array as a single-dimensional `array`. This is somehow similar to creating a subarray as it is a reference and not a copy of the elements. It is also writeable:

```
array_2d<double> d(3, 3, rng(1.0, 9.0));
std::cout << d << std::endl;
```

```


      1.0000       4.0000       7.0000
      2.0000       5.0000       8.0000
      3.0000       6.0000       9.0000
```

```
std::cout << d[*_] << std::endl;
```

```
[       1.0000       2.0000       3.0000       4.0000       5.0000
6.0000       7.0000       8.0000       9.0000 ]
```

4.4 Accessing elements of a multidimensional array

4.5 Multidimensional subarray

5 Tuples and functions

5.1 The tuple

5.2 Calling functions

5.3 Linear algebra example

6 Images

6.1 The class image

6.2 Saving, loading and image preview

7 Array class inheritance

8 Functional programming within sequential programming