vif Documentation

Release master

Corentin Schreiber

Jul 12, 2022

The core library

vif is a set of library and tools built to provide user-friendly vector data manipulation, as offered in interpreted languages like IDL, its open source clone GDL, or Python and numpy, but with the added benefit of C++: increased robustness, and optimal speed.

The library can be split into two components:

- The “core” library
- The “support” library

The core library introduces the `vec` type (a “data vector”), which is the most important data type in vif, while the support library provides functions and other tools to manipulate these vectors and perform common tasks. You can think of vif as a separate language inside C++, where the core library defines this language, and the support library is the the “standard” library where all the useful functions are stored.

Below is a code sample written in vif that illustrates the most basic functionalities.

```
using namespace vif;                   // import everything in current namespace
vec2f img = fits::read("img.fits");    // read a FITS image
img -= median(img);                    // subtract the median of the whole image
float imax = max(img);                 // find the maximum of the image
vec1u ids = where(img > 0.5*imax);     // find pixels at least half as bright
float sum = total(img[ids]);           // compute the sum of these pixels
img[ids] = log(img[ids]/sum);          // modify these pixels with a logarithm
fits::write("new.fits", img);          // save the modified image to a FITS file
```

CHAPTER 1

Overview of the core library

At the core of the vif library is the *vector* class, `vec`. This is basically an enhanced `std::vector`, and it therefore shares most of its features and strengths. On top of the `std::vector` interface, the vif vectors have extra functionalities to simplify data analysis and calculations, including overloaded mathematical operators, multi-dimensional indexing, and the ability to create "views" to access and edit subsets of a given vector.

Here we will first describe the properties of the vector class, and then describe the vector views. Lastly, a guide for writing "generic" functions that work with any vector type is provided.

CHAPTER 2

Vectors

2.1 Overview

A vector in vif is basically an enhanced `std::vector` (which, in fact, is used to implement the vif vectors), and it therefore shares most of its features and strengths.

In particular, a vector can contain zero, one, or as many elements as your computer can handle. Its size is defined at *runtime*, meaning that its content can vary depending on user input, and that it can change its total number of elements at any time. The elements of a vector are stored *contiguously* in memory, which provides optimal performances in most situations. Lastly, a vector is an *homogeneous* container, meaning that a given vector can only contain a single type of elements; for example `int` or `float`, but not both.

On top of the `std::vector` interface, the vif vectors have some extra functionalities. The most important ones are *Operator overloading* (which allows writing `v+w` instead of writing a loop to sum all the elements one by one), structured *Indexing* for multi-dimensional data (i.e., images, data cubes, and objects of higher dimensions), and *Views* (which allow accessing and modifying subsets of existing vectors).

Like in most advanced C++ libraries, vif vectors are *template-based*. This means they are designed to work with *any* data type `T` for their elements, for example `int`, `float`, or `std::string`. The type of a vector is therefore spelled `vec<1,T>`, where `T` can be replaced by any type (it could be a vector itself). There is no explicit restriction regarding the data type `T`, however some features may obviously not be available depending on the capabilities of your type. For example, if your type has no `operator*` (such as `std::string`), you will not be able to multiply vectors of this type. Lastly, the vif vector shares the same restrictions as the `std::vector` regarding the *copyable* and *movable* capabilities of the stored type.

The number of dimensions of a vector is specified in its type; this is the `1` in `vec<1,T>`. For example, a 2D image of `float` will be declared as `vec<2,float>` (or `vec2f`, see *Type aliases* below), while a 1D tabulated data set of `int` will be `vec<1,int>` (or `vec1i`). The fact that the number of dimensions is part of the type means that, while the number of *elements* in a vector is determined at runtime, the multi-dimensional nature of a vector is determined at *compile time*. In other words, a 1D vector cannot be turned into a 2D vector; you would have to create a new variable (using the `reform()` and `flatten()` functions). However it is possible to change, say, a 128x128 2D vector into a 256x256 2D vector. Only the *number* of dimensions is fixed, the length of each dimension is free to vary.

This restriction was imposed for two reasons: first, type safety, and second, performance. Since the compiler knows how many dimensions there are, it will output an error whenever you try to perform operations on two vectors of different dimensionality (for example, adding a 1D array to a 2D image; which would make no sense). In terms of performance, this also means that we also know at the time of compilation how many dimensions we need to deal with, so the compiler can more easily decide whether (or how) to unroll loops.

The hard limit on the number of dimensions depends on your compiler, as each dimension involves an additional level of template recursion. The C++ standard does not guarantee anything in this respect, but you should be able to go as high as 256 on all major compilers. Beyond this, you should probably see a therapist first.

2.2 Type aliases

While templates are a fantastic tool for library writers, they can easily become a burden for the *user* of the library, because of the additional syntax complexity (the `<...>` in the name of the vector type). Since vif is a numerical analysis library, we know in advance what types will most often be stored inside the vectors, and we therefore introduce type aliases for the most common vector types:

- `vec1f`: vector of `float`,
- `vec1d`: vector of `double`,
- `vec1cf`: vector of `std::complex<float>`,
- `vec1cd`: vector of `std::complex<double>`,
- `vec1i`: vector of `int` (precisely, `int_t = std::ptrdiff_t`),
- `vec1u`: vector of `unsigned int` (precisely, `uint_t = std::size_t`),
- `vec1b`: vector of `bool`,
- `vec1s`: vector of `std::string`,
- `vec1c`: vector of `char`.

Such type aliases are provided for dimensions up to 6 (i.e., `vec6f` exists, but not `vec7f`).

2.3 Size and dimensions

For any vector `v`, `v.dims` is an `std::array` that holds the length of each dimension of the vector: `v.dims[0]` is the number of elements along the first dimension, etc. This array is set automatically when the vector is created or resized, and should not be modified manually (except in very specific and rare circumstances).

The total number of elements in a vector can be obtained using `v.size()`, and this is equal to the product of all the dimensions of the vector. This is a cheap function, as the size is stored internally and does not need to be computed for each call. One can check if a vector is empty (i.e., contains no element) using `v.empty()`, which returns either `true` or `false` (this does *not* empty the vector).

2.4 Initialization and assignment

There are several ways to initialize and assign data to a vector:

- The "default" initialization, where the vector is empty.
- The "size" initialization, where the vector contains `n` default-constructed elements.
- The "list" initialization, where the vector is assigned a set of values.
- The "copy" initialization, where the vector contains a copy of the data from another vector.

- The "move" initialization, where the vector steals the data from another vector.

The last three can be used both for initialization and assignment (using `operator=`).

The "default" initialization is very cheap, since it involves no (or very little) allocation:

```
vec1f v; // empty
```

The "size" initialization pre-allocates memory for the data, which is very useful if you know in advance how many elements your vector needs to contain. The allocated data consists of elements which are default-constructed; this means a value of `0` for arithmetic types, `false` for `bool`, and empty strings for `std::string`.

```
vec1f v(10); // 10 zeros
vec2f w(10,20); // 200 zeros, arranged in a 2D shape of 10x20
vec3f z(w.dims,4); // 800 zeros, arranged in a 3D shape of 10x20x4
```

As shown in the last example (initializing `z`), the arguments of the constructor can be a mixture of integer values and `std::array`; the arrays can come, for example, from the dimensions of other existing vectors.

The "list" initialization explicitly specifies a set of values to be stored in the vector. This uses initializer lists, which can be nested for multidimensional vectors.

```
vec1f v = {1, 2, 3, 4}; // values from 1 to 4
vec2f w = {{1, 2}, {3, 4}, {5, 6}}; // values from 1 to 6 arranged in a 2D shape of␣
↪3x2
```

The "copy" initialization trivially copies (and optionally converts, see *Type conversion, and casting*) the values of a vector into another one.

```
vec1f v = {1, 2, 3, 4}; // values from 1 to 4
vec1f w = v;            // also contains values from 1 to 4
```

The "move" initialization will "steal" the values of another vector. The vector from which the values are "stolen" then becomes empty, and can be reused for other purposes later. This will usually be much faster than the "copy" initialization above, if you do not mind the side effect.

```
vec1f v = {1, 2, 3, 4}; // values from 1 to 4
vec1f w = std::move(v); // also contains values from 1 to 4, but now 'v' is empty
```

2.5 Resizing and adding elements

The dimensions and size of a vector can be modified in three main ways.

First, `v.clear()` will erase all the values from the vector and set all its dimensions to zero. This will set the vector in the state of a default-initialized vector (see above).

Second, `v.resize(...)` will change the dimensions of the vector. The parameters it accepts are the same as the "size" constructor (see above), i.e., either integral values for individual dimensions, or an `std::array` containing multiple dimensions, or any combination of these. While the total number of elements can be modified at will, the number of *dimensions* of the vector must remain unchanged.

```
vec2f w;             // empty, zero elements
w.resize(20,10);     // 20x10, 200 elements
w.resize(200,10);    // 200x10, 2000 elements
w.resize(200,10,5);  // error: cannot change a 2D vector into a 3D vector
```

Once the vector has been resized, its previous content is left in an undefined state, i.e., you can generally assume the previous values (if any) have been lost and replaced by meaningless garbage. The only exception is for 1D vectors. If the resize operation *decreases* the total number of elements, then values are erased from the end of the vector and the rest remains untouched. On the other hand, if the resize operation *increased* the total number of elements, new elements are inserted at the end of the vector, default constructed (i.e., zeros for integral types, etc.). This is the same behavior as `std::vector`.

Third, `v.push_back(...)` will add new values at the end of the vector, increasing its size. The behavior of this function is different for 1D and multidimensional vectors. For 1D vectors, this function appends a new element at the end of the vector, and therefore takes for argument a single scalar value. For multidimensional vectors, this function takes for argument another vector of `D-1` dimensions, and whose lengths match the *last* `D-1` dimensions of the first vector. The new vector is inserted after the existing elements in memory, and the *first* dimension of the first vector is increased by one.

```
vec1i v = {1,2,3};
v.push_back(4); // {1,2,3,4}

vec2i w = {{1,2,3}, {4,5,6}}; // shape 2x3
w.push_back({7,8,9});         // shape 3x3
w.push_back({7,8});           // error: dimensions do not match
```

For optimization, the `push_back(...)` function will generally be used in conjunction with `v.reserve()`. This function is identical to `std::vector::reserve()`. To understand what this function actually does, one needs to know the internal behavior of `std::vector`. At any instant, the `std::vector` only has enough memory to hold `N` elements, which is usually larger than the actual size of the vector. `N` is called the capacity of the vector. Once the allocated memory is full, and a new `push_back()` is called, `std::vector` allocates a larger amount of memory (typically `2*N` elements), copies the existing elements inside this new memory, and frees the old memory. This strategy allows virtually unlimited growth of a given vector; it is quite efficiently tuned, but it remains an expensive operation. Performances can be greatly improved if one knows *in advance* (and even approximatively) the total number of objects that need to be stored in the vector, so that the right amount of memory is allocated from the start, and no further reallocation is required. This function does just that: it tells `std::vector` how many elements it will (or might) contain at some point, so that the vector can already allocate enough memory to store them contiguously. Later, if you have reserved way too much memory, you can always ask the vector to free the surplus by calling `std::vector::shrink_to_fit()`, which will result in an additional reallocation but will free some unused memory.

2.6 Type conversion, and casting

The rules for converting a vector of a type `T` into a vector of another type `U` follow the rules for converting `T` itself into `U`. If `T` is implicitly/explicitly convertible to `U`, then it is always possible to implicitly/explicitly convert a `vec<1,T>` into `vec<1,U>`. For example here with a conversion from `vec1f` to `vec1i`:

```
vec1f v1 = {1.5, -2.2, 100.0};
vec1i v2 = v1; // this works
```

There is one notable exception to this rule, which is for vectors of type `bool`. In C++, `bool` can be implicitly converted to (and from) any other arithmetic type (such as `int` or `float`). While implicit conversion is very convenient in most cases, in the case of `bool` the risk of unwanted narrowing conversion (where data is lost) is much greater, while the actual use cases for implicit conversion are rarer; `bool` indeed carries a very different semantic compared to the other arithmetic types. For this reason, in vif it was decided to disable implicit conversion to and from `bool`. If needed, the conversion is still possible at no extra cost by using an explicit cast:

```
vec1f v1 = {1.5, -2.2, 100.0};
vec1b v2 = v1;         // does *not* work! compiler error
vec1b v2 = vec1b{v1}; // this works
```

2.7 Operator overloading

When dealing with `std::vector`, the only thing you can do to operate on all the elements of an `std::vector` is to iterate over these elements explicitly, either using a C++11 range-based loop, or using indices:

```
// Goal: multiply all elements by two.
std::vector<float> v = {1,2,3,4};

// Either using a range-based loop,
for (float& x : v) {
    x *= 2;
}

// ... or an index-based loop.
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] *= 2;
}
```

While this is fairly readable (especially the first version), it is still not very concise and expressive. For vif vectors, we have *overloaded* the usual mathematical operators to make it possible to write the above code in a much simpler way:

```
// Using vif vector.
vec1f v = {1,2,3,4};
v *= 2;
```

Not only this, but we can also perform operations on a pair of vectors:

```
// Goal: sum the content of the two vectors.
vec1f x = {1,2,3,4}, y = {4,3,2,1};
vec1f z = x + y;
// z: {5,5,5,5}
```

Almost all the mathematical and logical operators are overloaded. Therefore, as a rule of thumb, if you can do an operation with a type `T`, you can do it with `vec<1,T>` as well. The one notable exception are bitwise operators: `|`, `&`, and `^`. The reason is twofold: first, these are not so commonly used in data analysis, and second, the `^` operator can be mistakenly interpreted as the exponentiation operator, that some other languages possess (if you need to do exponentiation, use `pow()`).

Note: In Python+NumPy and some C++ libraries (like xtensor), it is possible to mix vectors with different numbers of dimensions in a given operation, if some conditions are satisfied. For example, this allows multiplying the same 1D vector to each row or column of a 2D vector without having to write an explicit loop. This mechanism is called *broadcasting*. It is *not* implemented in vif, and will likely never be. Indeed, it is perceived that the benefits are not worth the costs, both in terms of making the vif codebase more complex, but also in introducing more complex rules for the user of the library. Therefore, in vif, one can only do arithmetics on vectors which have the *exact* same dimensions. If you require an operation similar to what broadcasting provides, you can always write the loop explicitly.

Note: Contrary to some other C++ libraries with vectorized arithmetic (such as Eigen, blazelib, or xtensor), vif does not use *expression templates*. Instead, each operation is executed immediately (no lazy evaluation) and operates if necessary on temporary intermediate vectors. While this may appear to be a sub-optimal implementation, vif was tuned to makes good use of return value optimization, move semantics, and for reusing the memory of temporaries in chained expressions. As a result, performance was found to be on par with expression templates in the most common situations, but memory consumption is generally higher in vif. This is of course dependent on the precise calculation to perform. The benefit of not using expression templates is a reduced compilation time, and a much simpler code base.

2.8 Constant vectors

It is not possible to create a vector of constant elements, `vec<1, const int>` will not compile. The correct way to create a constant vector is to make the vector itself constant: `const vec<1,int>`.

CHAPTER 3

Indexing

3.1 Flat and multidimensional indices

The `std::vector`, which is used to implement vif vectors, is a purely linear container: one can access its elements using `v[i]`, with `i` ranging from `0` up to `std::vector::size()-1` (included). In C++, all indexing is zero-based: the first value has an index of zero (contrary to R, or Fortran).

One can do exactly the same thing on a vif vector. In this context, `i` is called a "flat" index, since it conveys no particular structure about the data. The vif vectors go beyond this, and also allow N-dimensional indexing, that is, using the combination of multiple indices to identify one element. This is particularly useful to work on images, which can be seen as 2-dimensional objects where each pixel is identified by its coordinates `x` and `y`. In this context, the pair `x,y` is called a "multidimensional" index.

Since regular vectors use the syntax `v[i]` to access 1-dimensional data, the natural syntax for 2-dimensional indexing would be `img[x,y]`. This syntax is valid C++ code, but unfortunately will not do what you expect... This will call the dreaded *comma* operator, which evaluates both elements `x` and `y` and returns the last one, i.e., `y`. So this code actually means `img[y]`. If you ever make this mistake, most compilers should emit a warning, since in this context `x` is a useless statement. So watch out! Unfortunately, there is no sane way around it. The only valid syntax is therefore `img(x,y)`.

Below is an example of manipulation of a 2D matrix:

```
// Create a simple matrix.
vec2f m = {{1,2,3}, {4,5,6}, {7,8,9}};

// Index ordering is similar to C arrays: the last index is contiguous
// in memory. Note that this is *opposite* to the IDL convention.
m(0,0); // 1
m(0,1); // 2
m(1,0); // 4

// It is also possible to access elements as they are laid out in memory,
// using square brackets and "flat" indices.
m[0]; // 1
```

(continues on next page)

(continued from previous page)

```
m[1]; // 2
m[3]; // 4

// ... but doing so with parenthesis will generate a compiler error:
m(0); // error: wrong number of indices for this vector
```

The elements of a vector can therefore be accessed in two ways: either using a flat index (and square brackets), or using the appropriate multidimensional index (and parentheses). As illustrated above, when using the multidimensional index, you must provide as many indices as the number of dimensions in the vector, no more, no less.

Indexing a vector can only be done with integers. Indexing with *unsigned* integers is faster, because it removes the need to check if the index is negative, and it should therefore be preferred whenever possible. Negative indices are allowed, and they are interpreted as *reverse* indices, that is, `-1` refers to the last element of the vector, `-2` to the one before the last, etc.

3.2 Bounds checking, and safe indexing

All the indexing methods described above perform *bound checks* before accessing each element. In other words, the vector class makes sure that each index is smaller than either the total size of the vector (for flat indices) or the length of its corresponding dimension (for multidimensional indices). If this condition is not satisfied, an assertion is raised explaining the problem, with a backtrace, and the program is stopped immediately to prevent memory corruption.

```
vec1f v(10):
v[20] = 3.1415;
```

When executed, the code above produces:

```
error: operator[]: index out of bounds (20 vs. 10)

backtrace:
 - vif_main(int, char**)
   at /home/cschreib/test.cpp:5
```

This bound checking has a small but sometimes noticeable impact on performances. In most cases, the added security is definitely worth it. Indeed, accessing a vector with an out-of-bounds index has very unpredictable impacts on the behavior of the program: sometimes it will crash, but most of the time it will not and memory will be silently corrupted. These problems are hard to notice, and can have terrible consequences… Identifying the root of the problem and fixing it may prove even more challenging. This is why these checks are enabled by default, even in “Release” mode.

However, there are cases where bound checking is superfluous, for example if we already know *by construction* that our indices will always be valid, and no check is required. Sometimes the compiler may realize that and optimize the checks away, but one should not rely on it. If these situations are computation-limited, i.e., a lot of time is spent doing some number crushing for each element, then the performance hit of bound checking will be negligible, and one should not worry about it. On the other hand, if very little work is done per element and most of the time is spent iterating from one index to the next and loading the value in the CPU cache, then bounds checking can take a significant amount of the total time.

For this reason, the vif vector also offers an alternative indexing interface, the so-called “safe” interface. It behaves exactly like the standard indexing interface described above, except that it does not perform bound checking. One can use this interface using `v.safe[i]` instead of `v[i]` for flat indexing, or `v.safe(x,y)` instead of `v(x,y)` for multidimensional indexing. The safe interface can also be used to create views. This interface is not meant to be used in daily coding, but rather for computationally intensive functions that you write once but use many times. Just be certain to use this interface when you know *for sure* what you are doing, and you can prove that the index will always be valid. A good strategy is to always use the standard interface and, only when the program runs successfully, switch to the safe interface for the most stable but time-consuming code sections.

Note: One may wonder why the word `safe` was used instead of `unsafe`, since indexing without bounds checking is actually an "unsafe" operation. The reason why is that writing `v.safe[i]` can be understood as: "we are in a context where we know where the index `i` came from, we're *safe*, we can disable bounds checking". Perhaps another reason is that I would feel somewhat uncomfortable at writing `unsafe` everywhere in the core functions of the library, which is supposed to only contain safe code...

3.3 Loops, and traversing data

The fastest way to loop over all the elements of a vector is to use a range-based loop, since this avoids having to deal with indexing and bound checking altogether:

```
for (float& v : m) {
    do_some_stuff_with(v);
}
```

If you care not only about the values but also about their flat index in `m`, then the fastest loop will be:

```
for (uint_t i : range(m)) {
    do_some_more_stuff_with(i, m[i]);
}
```

The `range(m)` function can only be used in `for` loops. It will generate integers from `0` to `m.size()` (excluded) if `m` is a vector, or from `0` to `m` (excluded) if `m` is an integer. It can also have a different starting value when called as `range(i0,n)`, in which case the first value will be `i0`.

Lastly, if you care about the multidimensional index, then you need to loop on each dimensions explicitly. When doing so, always loop on the *first* dimension in the outermost loop, and on the *last* dimension in the innermost loop, to make best use of memory locality and CPU caches:

```
for (uint_t i : range(m.dims[0]))
for (uint_t j : range(m.dims[1])) {
    do_even_more_stuff_with(i, j, m(i,j));
}
```

3.4 Partial indexing

When dealing with multi-dimensional vectors, in some cases one may not want to access a single element, but instead all the elements along a certain dimension, or a handful of elements at once. For example, you may want to access an entire row of pixels in an image, or only the values which are greater than zero. This can be done with *Views*.

CHAPTER 4

Views

4.1 Overview

As shown in *Operator overloading*, instead of accessing each element of a vector individually to perform some operation, we can use operator overloading to act on all the elements of the vector at once: `v *= 2`. As shown in *Indexing*, we can also, like with `std::vector`, modify each element individually using their indices: `v[2] *= 2`.

The *view* is a generalization of this concept, allowing you to access and operate on an arbitrary number of elements from an existing vector. Each element of the view is actually a *reference* to an element in this vector, and performing operations on the elements of the view will modify the elements in the vector. The interface of views is almost indistinguishable from that of vectors, and both can be used interchangeably in almost all cases. Generic codes and functions that work with vectors will therefore also work with views.

Creating a view is simple: instead of indexing a vector with integer values, as in `v[2]`, you would index the vector using *another vector* containing multiple indices:

```
// Create a simple vector.
vec1f w = {1,2,3,4,5,6};

// We want to "view" only the second, the third and the fifth
// elements. So we first create a vector containing the
// corresponding indices.
vec1u id = {1,2,4};

// We create the view and multiply all the elements by two.
w[id] *= 2;
// 'w' now contains {1,4,6,4,10,6};
```

In the example above, the type of the epxression `w[id]` is `vec<1,float*>`. Views have the same elements type as the vector they point to, but the template parameter of `vec<...>` is specified as a *pointer* (`T` -> `T*`) to distinguish them from vectors. The dimensions of the view are set by the vector that was used for *indexing* (regardless of the dimensions of the pointed vector). For example, if `v` is a 1D vector and we create a 2D array of indices `id`, then `v[id]` will be a 2D view.

Note: Since a view keeps *references* to the elements of the original vector, the lifetime of the view must not exceed that of the original vector. Else, it will contain *dangling* references, pointers to unused memory, and this should be avoided at *all cost*. For this reason, views are not meant to be stored into named variables, but should only be used in temporary expressions as above. This point is discussed also in *Known issues, problems, and limitations*. The only notable exception to this rule is when passing views to functions (see *Guidelines for writing generic functions*).

As for regular indexing, views can only be created using vectors of *integer* indices (signed, or unsigned). Bounds checking will be done on each element of the index vector, to make sure that no index goes past the end of the vector. If you know this cannot happen, and therefore that this bounds checking is superfluous, you may want to use "safe" indexing (see *Indexing*) to improve performances.

Lastly, views on different vectors can be involved in the same expression, as long as their dimensions are the same:

```
// Create two vectors.
vec1f x = {1,2,3,4,5,6};
vec1f y = {6,5,4,3,2,1};

// Indices
vec1u idx = {1,2,4};
vec1u idy = {4,0,5};

// Do some operation
vec1f z = x[idx] + y[idy]; // {2,3,5} + {2,6,1} = {4,9,6}
```

Mixing views (or vectors) of different sizes will trigger an error at runtime.

4.2 Range indexing

Sometimes, one will want to use views to access all the elements at once, for example to set all the elements of a vector to a specific value. This can be done with a loop, of course, but the whole point of vif is to avoid explicit loops whenever possible. An alternative is to use a view, with an index vector that contains all the indices of the target vector:

```
vec1i v = {1,2,3,4};
vec1u id = {0,1,2,3}; // all the indices of 'v'
v[id] = 12;           // all the values are now equal to 12

// Note that, by design, the following will not compile (too error prone):
v = 12; // "error: no viable overloaded '='"
```

However, not only is this not very practical to write, it is error prone and not very clear. If someday we decide to add an element to `v`, we also have to modify `id`. Not only this, but it will most likely be slower than writing the loop directly, because the compiler may not realize that you are accessing all the elements contiguously, and will fail to optimize it properly.

The optimal way to do this in vif is to use the "placeholder" symbol, defined as a single underscore _. When used as an index, it means "all the indices in the range". Coming back to our example:

```
vec1i v = {1,2,3,4};
v[_] = 12; // it cannot get much shorter!
```

This placeholder index can be used in all situations, with both flat and multidimensional indexing:

```
vec2f img(128,128);
img(0,_) = 12; // accessing the first row of the image

// Any combination is allowed
vec4f crazy(5,4,12,8);
crazy(5,_,2,_) = 5.0; // this creates a 2D view of shape 4x8

// The above is equivalent to:
for (uint_t i : range(crazy.dims[1]))
for (uint_t j : range(crazy.dims[3])) {
    crazy(5,i,2,j) = 5.0;
}
```

This can be further refined to only encompass a fraction of the whole range, using a specific syntax:

```
vec1i v = {1,2,3,4};
v[_-2] = 12;   // only access the indices from 0 to 2 (included)
v[2-_] = 12;   // only access the indices from 2 to 3 (the last, included)
v[1-_-2] = 12; // only access the indices from 1 to 2 (included)

// Watch out, this is *not* range indexing!
v[1-2] = 12;   // only access index 1-2 = -1
```

4.3 Filtering and selecting elements

In the previous section we have seen that a view can be created using a vector of indices. In most cases, such vector is not created manually, as in the examples above, but comes from a *filtering* function, `where()`. This function is part of the support library, but it is important enough to be mentioned here.

`where()` accepts a vector of `bool` (of any dimension) as single argument, and returns all the *flat* indices where the vector values are `true`. This can be combined with views to perform complex operations on vectors. For example:

```
// Set all negative values to zero
vec1f v1 = {-1.01, 2.0, 5.0, -2.1, 6.5};
v1[where(v1 < 0.0)] = 0.0;
v1;      // { 0.0,  2.0, 5.0,  0.0, 6.5}

// Add one to all values between 0 and 6
vec2f v2 = {{-1.0, 2.0}, {8.0, 3.4}};
v2[where(v2 > 0.0 && v2 < 6.0)] += 1.0;
v2;      // {{-1.0, 3.0}, {8.0, 4.4}}
```

4.4 Differences between views and vectors

While views are mostly compatible with vectors in terms of interface, by design some features of vectors are not available for views:

- Initialization: views can only be created as described above.
- Assignment and resizing: assigning anything to the view will affect the target vector, not the view itself. Therefore once a view is created, you cannot change which elements it points to.

4.5 Constant views and views on constant data

There are two ways that views can have "constant" semantics, where it is only possible to *read* the viewed data and not modify it. The first way is when constructing a view from a constant vector, in which case the view carries the `const` qualifier in its data type (`vec<1,const int*>`):

```
const vec1i v = {1,2,3,4};
v[_] = 12; // error: cannot modify values of vec<1,const int*>
```

The second way arises when views are function parameters (see *Guidelines for writing generic functions* for more detail):

```
void set_values(const vec<1,int*>& v) {
    v[_] = 12; // error: cannot modify values of const vec<1,int*>
}
```

There is no difference between these two cases: "a constant view on non-constant data" and "a view on constant data", `const vec<1,float*>` is semantically identical to `vec<1,const float*>`. This is different from raw pointers, because a pointer can be modified to point to a different value, while views cannot (by design).

4.6 Aliasing

The implementation of vectors and views in vif is such that aliasing *never* occurs in vectorized operations. More precisely, any assignment of the form `x = y` (or `x += y`, etc.) occurs *as if* executed in the following order:

1. `y` (the right-hand-side) is evaluated,
2. the values of `y` are copied in a temporary vector,
3. `x` (the left-hand-side) is evaluated,
4. the values of the temporary vector are assigned to the elements of `x`.

In practice, the creation of the temporary vector (step 2) may be dropped for optimization purposes, but only in cases where it would not change the outcome of the operation, that is, when aliasing is guaranteed not to occur. The following illustrates when aliasing *could* occur, and describes in practice how it is avoided in vif.

Because views hold *references* to existing data, there is the possibility of the same data being read and modified in the same expression. This is, essentially, what is called "aliasing":

```
vec1i v = {1,2,3,4};
vec1u id = {1,2,3,0};
v[id] = v; // what happens here?
```

This can create confusing situations, like the above, where it matters in which *order* the operations are performed. These situations are identified using a check, made prior to every assignment between a vector and view, a view and a vector, or two views. Each view carries a pointer to the original vector: if this pointer matches the vector involved in the assignment (or the pointer of the other view), then aliasing is detected. In such cases, the data on the *right* side of the equal sign is copied to a temporary vector, which is then assigned to the data on the *left* side of the equal sign. In all other cases, aliasing is ignored and no temporary is created to avoid the performance hit.

So, the example above first creates a copy of `v`, then assigns it to itself following the order in the view. The vector then contains the values `{4, 1, 2, 3}`, as one would expect if the data on the right side of the equal sign originated from another vector. If aliasing had not been detected, one possible outcome would have been `{1, 1, 1, 1}`, as some of the vector's values would have been modified *before* being read.

A similar problem can arise without views:

```
vec1i v = {1,2,3,4};
v += v[0]; // what happens here?
```

Possible outcomes are `{2,3,4,5}` if `v[0]` is treated as the *value* `1`, or `{2,4,5,6}` if `v[0]` is treated as the *reference* to the first element of `v`, leading to aliasing. To avoid the latter, assigning operators such as `+=` always take scalar arguments by value. The outcome will therefore be `{2,3,4,5}`.

This means that the above codes are *not* identical to their equivalent with explicit loops:

```
vec1i v = {1,2,3,4};
vec1u id = {1,2,3,0};

for (uint_t i : range(v)) {
    v[id[i]] = v[i];
}

// v = {1,1,1,1}, aliasing *did* occur

v = {1,2,3,4};

for (uint_t i : range(v)) {
    v[i] += v[0];
}

// v = {2,4,5,6}, aliasing *did* occur
```

While this may cause confusion, experience has shown that aliasing is more often an unwanted nuisance than a feature. Furthermore, with the explicit loop it is immediately apparent that `v[i]`, `v[id[i]]`, or `v[0]` will be re-evaluated on each iteration, therefore that the corresponding value may change.

CHAPTER 5

Known issues, problems, and limitations

This section describes issues and limitations in vif. Some of these are design issues which should be solved by me, the author of the library, but which I actually *cannot* solve, or haven't found the time to solve yet. Some are not issues but *features*, namely, conscious choices that were made when designing the library, and that will thus never be "solved".

In any case, the aspects listed in this page may require you, the user, to pay special attention to some corner cases, and you should therefore make sure you are familiar with them.

5.1 Dangling views

The problem 1.

A "dangling" reference is a invalid reference that points to an object that no longer exists. Creating such dangling references is a common programming mistake not specific to vif, which compilers are fortunately well equipped to detect:

```
int& foo() {
    int i = 0;
    return i; // warning: reference to local variable 'i' returned
}
```

vif views suffer from the same problem: a dangling view can be created that points to a vector that no longer exists. Unfortunately, compilers are not aware of it:

```
vec<1,int*> foo() {
    vec<1,int> v = {0, 1, 2, 3};
    return v[_];
}
```

This code will unfortunately compile without warning, and there is no (efficient) programmatic way to identify it at run time. Calling `foo()` will therefore not throw any immediate error, but create an "undefined behavior"; it may do anything, and that's very bad.

The case above can be spotted by looking carefully at the function itself: 1) the function returns a view, 2) this view points to a vector that is created in the function, 3) therefore it's a dangling view. However there are more subtle cases where the issue is not as obvious...

The problem 2.

One such case where dangling views are hard to spot is when returning a view from a lambda, rather than from a function with a specified return type. In this case, the return type is *inferred* from the returned expression, and this causes some surprising results. To avoid this, C++ has some special rules for references:

```
auto foo = []() {
    int i = 0;
    return i;
}
```

This lambda actually returns *a copy* of `i`, to avoid silently creating a dangling reference. Again, unfortunately this special behavior does not apply to vif views:

```
auto foo = []() {
    vec1i v = {0, 1, 2, 3};
    return v[_];
}
```

This lambda will return a dangling view, and because it would not do that for normal types and references like `int` and `int&`, it is *not* obvious to spot.

The solution.

If you are a seasoned C++ programmer, you know how to avoid dangling references within the current C++ language rules. Use the same caution with views, and you will avoid most instances of "Problem 1" (as described above) without relying on the compiler to throw warnings at you.

To solve "Problem 2", take extra precautions whenever you write a lambda function (or a function with `auto` return type in C++14) to ensure you are not accidentally returning a view. If you do not trust yourself to do this, then make sure you *always* specify the expected return type of your lambda functions:

```
auto foo = []() -> vec1i {
    vec1u v = {0, 1, 2, 3};
    return v[_];
}
```

Note that smart people are currently thinking of adding new C++ rules that will allow me (and other library authors who experience similar problems) to modify the view class such that it will benefit from all the good magic that C++ currently applies to references. This will fix "Problem 2", and some cases of "Problem 1". In the mean time, just be careful!

5.2 Invalid views

The problem.

With `std::vector<T>`, any operation that modifies the size of the vector *invalidates* all the iterators that point to this vector:

```
std::vector<int> v;
auto b = v.begin();
```

(continues on next page)

(continued from previous page)

```
v.resize(10);
// b is now invalid!
```

The same is true for views: if a view points to a vector and this vector is later resized or re-assigned, the view becomes invalid and *must not* be used any more.

```
vec1u vec = {1,2,3,4};
vec<1,int*> view = vec[_];

v = {1,2,3,4,5,6};
// the view is now invalid!
```

The reason why is that the view stores *pointers* to the values in `vec`, not indices. These pointers may become invalid themselves if the values of `vec` are moved to another spot in the computer's memory.

The solution.

There is a reason why shortcut types are provided for vectors (`vec1i` instead of `vec<1,int>`) and not for views: *views are only meant to be temporaries*, they should not be saved into named variables like in the above. If you feel it is necessary to do this for performance reasons, simply avoid using views altogether and manipulate indices explicitly, this will be faster.

CHAPTER 6

Guidelines for writing generic functions

This section contains more advanced technical details about the implementation of vectors and views in vif. It also includes tips and tricks for writing correct, efficient, and generic functions.

6.1 What is a generic function?

A "generic" function can operate on vectors regardless of the precise type of their elements. For example, a function to shuffle the values inside a vector does not care whether these values are integers, strings, or potatoes, it just needs to know how many values there are.

In C++, such generic functions are written using *template metaprogramming*:

```
template<typename T>
void foo(const T& v) {
    print(v(1,_)*2.5);
}

vec2f v1 = {{1.0}, {5.0}, {-1.0}}; // float, shape 3x1
foo(v1); // prints {12.5}

vec2i v2 = {{1, 5}, {-1, 1}}; // int, shape 2x2
foo(v2); // prints {-2.5, 2.5}
```

Note: This example uses the `print()` function from the vif support library, which simply displays its arguments on the terminal.

6.2 Summary of guidelines

The functions we use as examples here can be somewhat silly, but they will serve to illustrate a number of important "rules" which one should follow when writing generic functions. These rules are explained in detail below, and can be summarized as follows:

- Use the most specific type possible for the function arguments (e.g., `vec<1,T>` instead of just `T`).
- Even when the function is supposed to work on only one specific data type, leave the data type of vectors unconstrained in order to support both vectors and view (e.g., `vec<D,T>` instead of `vec<D,int>`).
- Use `std::enable_if<>` to express any remaining constraints on the type.
- Do not use the `T` in `vec<D,T>` to form new variables, use `meta::rtype_t<T>` instead.
- Provide default values for template arguments whenever it makes sense, to enable support for initializer lists.
- Avoid output or input/output parameters whenever possible, else use universal references `T&&` and `std::enable_if<>` as described in the guidelines below.
- Use the available helper tools to vectorize existing functions.
- Use `vif_check()` to express any constraints on the data that can only be checked at run time (number of elements, value ranges, etc.).

6.3 Expressing constraints on function arguments

In the example above, the type of the function's argument `v` is `T`, and is totally unconstrained. It could be anything. However this specific function has an *implicit* constraint on the type `T`: it must be possible to write `v(1,_)*2.5`. This means `v` *must* be a 2D vector, and the elements must be of arithmetic type. If you try to use this function on a value which does not satisfy this implicit constraint, the compiler will generate an error:

```
vec1f v4 = {1.0, 5.0, 6.0};
foo(v4); // error! 1D vector
```

The program will not compile, which is good. However the error message will be *nasty* (180 lines of errors with Clang), and the error will point to code *inside* the function. This is far from ideal, because the user of the function should not need to understand what the function does internally to fix the error. This can be fixed by adding constraints on the type `T`:

```
template<typename T>
void foo(const vec<2,T>& v) {
    print(v(1,_)*2.5);
}
```

Here, we state that `v` must be a 2D vector, and we leave the type of the elements unconstrained. Using this new definition of `foo()`, the error message in the case above becomes much smaller (4 lines of errors), and explicitly says that there is no matching function call for `foo(v4)`. This is much better. Therefore you should always make sure to specify as many constraints as possible in the *signature* of the function (i.e., the type of its arguments).

We are not done though. Indeed, we still left the type of the elements unconstrained, while we need elements of arithmetic types to be able to write `v(1,_)*2.5`. For example, using a vector of strings would be an error:

```
vec2s v5 = {{"I", "am"}, {"a", "string"}};
foo(v5); // error! vector of strings
```

Again, this emits a lengthy error from inside the function (20 lines of errors). We can fix this by adding extra constraints on the type `T` of the elements. One possibility is to force it to be some "common" type, like `double`:

```
// Note: parameter is fully constrained, it is not generic anymore
void foo(const vec<2,double>& v) {
```

(continues on next page)

(continued from previous page)

```
    print(v(1,_)*2.5);
}
```

This makes the error much easier to understand (7 lines of errors), but it has the important downside that the function is no longer generic: it *needs* a vector of `double`. If you try to call it on a vector of `float`, it will first have to make a copy of that vector and convert all values to `double` before calling the function, which is not optimal. It will also fail to work on *views* (see below). So unless you know the function should only be used with `double` values, this is not the right solution. Instead, we can leave the type of elements to be generic, and use `std::enable_if<>` to express a constraint on this type, in this case `std::is_arithmetic<T>`:

```
template<typename T,
    typename enable = typename std::enable_if<std::is_arithmetic<T>::value>::type>
void foo(const vec<2,T>& v) {
    print(v(1,_)*2.5);
}
```

With this version of the function, the error when called on vectors of strings becomes much clearer (4 lines of errors) and says that you cannot call the function on strings. Again, much better!

So, that's it? Not quite. There is one last implicit requirement when we write `v(1,_)`: the first dimension of `v` must have at least two elements. There is no way to check this at the time of compilation, so the faulty program below will compile:

```
vec2i v6;
foo(v6); // compiles, but runtime error! empty vector
```

It will fail at runtime though. The backtrace will show that the error happened in `foo()`, but with a rather cryptic error message:

```
error: operator(): index out of bounds (1 vs. 0)
```

The solution here is to perform an explicit check inside the function, and emit a clearer error message using the `vif_check()` function:

```
template<typename T,
    typename enable = typename std::enable_if<std::is_arithmetic<T>::value>::type>
void foo(const vec<2,T>& v) {
    vif_check(v.dims[0] >= 2, "vector must have at least two elements along first␣
↪dimension ",
        "(got ", v.dims[0], ")");
    print(v(1,_)*2.5);
}
```

The error shown to the user then becomes clear:

```
error: vector must have at least two elements along first dimension (got 0)
```

Now that we do an explicit check that the index `1` is valid before accessing the vector, we no longer need the vector to perform automatic bounds checking. Therefore we can use the "safe" indexing interface:

```
template<typename T,
    typename enable = typename std::enable_if<std::is_arithmetic<T>::value>::type>
void foo(const vec<2,T>& v) {
    vif_check(v.dims[0] >= 2, "vector must have at least two elements along first␣
↪dimension ",
        "(got ", v.dims[0], ")");
```

(continues on next page)

(continued from previous page)

```
    print(v.safe(1,_)*2.5);
}
```

This is the optimal way to write this function, and it is clearly not as pretty as the very first version. This shows that, while writing generic functions is easy, writing them *well* is much harder. For this reason, always check in the support library if a function already exists before writing your own.

It should be said, however, that the very first version we wrote actually does the work we expect it to do (save for the fact that it does not support initializer lists, see below). It is not "incorrect"; its only defect is that it will not be very helpful when things go wrong.

6.4 Supporting initializer lists

There is one last modification we can do to make the `foo()` function "as good as it gets". Indeed, even with the last version, we cannot use initializer lists directly as function arguments:

```
foo({{1, 5}, {-1, 1}});
```

This generates an error because the compiler is not smart enough to infer the type `T` of the vector from this initializer list. Unfortunately, in general we cannot do this *perfectly* and support any type in the initializer list.

But we can still make it work. The trick is to specify a default value for the template parameter `T`, for example `double`. This way, the initializer list will automatically be used to form a vector of `double`, and the code will compile and run. This is not a perfect solution because the *true* type of the values in the initializer list is lost, but in most cases it is possible to identify a "safe" type (such as `double`) that will be able to do the job anyway.

In this particular case, `double` is actually a perfect choice because we multiply the values of the vector by `2.5`, which requires a conversion to `double` anyway. So converting the values of the initializer list to `double` will not change the final result. The definite, final version of our function is thus:

```
template<typename T = double, // use a default value here
    typename enable = typename std::enable_if<std::is_arithmetic<T>::value>::type>
void foo(const vec<2,T>& v) {
    vif_check(v.dims[0] >= 2, "vector must have at least two elements along first␣
↪dimension ",
        "(got ", v.dims[0], ")");
    print(v.safe(1,_)*2.5);
}
```

6.5 Supporting both vectors and views

For a vector of type `vec<D1,T>`, a view will have a type `vec<D2,T*>` or `vec<D2,const T*>`. The number of dimensions can be different, and the data type is a pointer to the type of elements in the vector. The `const` qualifier is used to propagate const-correctness if the original vector was declared `const`.

This makes it relatively easy to write function that work on both vectors and view, but this distinction means that there are a number a details to keep in mind. Consider this generic function that computes the sum of all the elements in a vector:

```
template<std::size_t D = 1, typename T = double>
T sum_it_all(const vec<D,T>& v) {
    T ret = 0;
```

(continues on next page)

(continued from previous page)

```
    for (const T& val : v) {
        ret += val;
    }

    return ret;
}
```

Note: Such a function already exists in the vif support library, and is called `total()` (for integers and floating point values) or `count()` (for boolean values). Their return type is determined in a smarter way than we discuss here, to prevent overflow and underflow.

This implementation works for all vectors, but it will fail for views. Indeed, if called on a view of type `vec<1, int*>`, then `T = int*`, and the return value is not an integer but an (invalid!) pointer to an integer. Fortunately, it will not even compile because the loop will try to assign the values of `v` to a `const int*&`, which will fail. Therefore, the type `T` should never be used directly like this.

Instead, you should apply the transform `meta::rtype_t<T>`, which essentially transforms `T*` into `T` and removes const qualifiers, and use `auto` whenever possible to let the type system make the right decisions for you:

```
template<std::size_t D = 1, typename T = double>
meta::rtype_t<T> sum_it_all(const vec<D,T>& v) {
    meta::rtype_t<T> ret = 0;
    for (const auto& val : v) {
        ret += val;
    }

    return ret;
}
```

There are a few, rarer corner cases to keep in mind when both view and vectors need to be supported. The case of output parameters, in particular, is described further below.

6.6 Vectorizing scalar functions

Most function created in C++ thus far, including those in the C++ standard library, are *scalar* functions which operate on one single value. The best example of this are all the mathematical functions, `sqrt()`, `pow()`, `ceil()`, etc. These functions can be *vectorized* to operate directly on vector data without having to write a loop. The vif support library contains a large number of such vectorized functions:

```
double v1 = 2.0;
sqrt(v1); // 1.41...
vec1d v2 = {2.0, 4.0, 6.0};
sqrt(v2); // {1.41..., 2.0, 2.45...}
```

However the vif support library cannot contain *all* functions that ever existed, and you may create your own scalar functions that you wish to vectorize. This can be achieved using the preprocessor macro `VIF_VECTORIZE()`:

```
float myfunc(float v) {
    return sqrt(3*v + 5.0); // whatever you wish to do
}

VIF_VECTORIZE(myfunc)
```

The macro must be called in the global scope, inside a namespace, or a the root scope of a class. It *cannot* be called inside a function. This macro emits two additional functions with the same name. The first function is the most generic vectorized version of the scalar version, which will get used most of the time.

The second version offers an interesting optimization opportunity when the return type is the same as the argument type (as is the case for `myfunc`), and when the function is called on a temporary vector (not views). This optimized version reuses the memory of the temporary vector instead of returning a brand new vector. This offers important optimizations in case of chained calls:

```
vec1d v1 = {1.0, 1.2, 1.5};
vec1d v2 = myfunc(sqrt(v1));
```

In this example, `sqrt(v1)` creates a temporary vector, and `myfunc()` applies `myfunc()` in-place on the values of the temporary vector. It is equivalent to this:

```
vec1d v1 = {1.0, 1.2, 1.5};

vec1d tmp(v1.dims);
for (uint_t i : range(v1)) {
    tmp[i] = sqrt(v1[i]);
}
for (double& v : tmp) {
    v = myfunc(v);
}

vec1d v2 = std::move(tmp);
```

Without this optimization, the chained call would have created two temporaries:

```
vec1d v1 = {1.0, 1.2, 1.5};

vec1d tmp1(v1.dims);
for (uint_t i : range(v1)) {
    tmp1[i] = sqrt(v1[i]);
}
vec1d tmp2(tmp1.dims);
for (uint_t i : range(tmp1)) {
    tmp2[i] = myfunc(tmp1[i]);
}

vec1d v2 = std::move(tmp2);
```

The optimal version would avoid the extra loop:

```
vec1d v1 = {1.0, 1.2, 1.5};

vec1d tmp(v1.dims);
for (uint_t i : range(v1)) {
    tmp[i] = myfunc(sqrt(v1[i]));
}

vec1d v2 = std::move(tmp);
```

This is only possible using expression templates, which currently vif does not support for the sake of simplicity. Therefore, if performances are critical you may want to write the loop explicitly (following the guidelines in *Indexing* for optimal performance).

A cleaner alternative is to use `vectorize_lambda_first()`, which transforms a lambda function into a functor with overloaded call operator that works on both vector and scalar values. It also supports the optimization for chained calls. Contrary to the `VIF_VECTORIZE()` macro, `vectorize_lambda_first()` can be called in any scope, including inside other functions:

```
auto chained = vectorize_lambda_first([](float f) { return myfunc(sqrt(f)); });

vec1d v1 = {1.0, 1.2, 1.5};
vec1d v2 = chained(v1);
```

Both `VIF_VECTORIZE()` and `vectorize_lambda_first()` will vectorize the function/lambda on the *first* argument only. Other arguments will simply be forwarded to all the calls, so `foo(v,w)` will call `foo(v[i],w)` for each index `i` in `v`.

If instead you need to call `foo(v[i],w[i])`, you should use `vectorize_lambda()`. This is an alternative implementation that will support vector or scalars for *all* its arguments, and will assume that the vectors all have the same size and should be jointly iterated. The downside of this implementation is that the chaining optimization is not available.

6.7 Output arguments and views

In general, the only output of a function must be its return value. Output arguments should only be used when: a) the function must return multiple values, and b) it would be inefficient or impractical to return them by value. Otherwise, one may wish to use input/output arguments for functions that have no return value but only modify the content of an existing vector. As you will see below, writing functions with output or input/output vector arguments is possible but a bit nasty, so make sure you really need them before diving in.

The typical example where output arguments are needed is the following function which converts a string to a value of another type (e.g., an integer):

```
template<typename T>
bool from_string(const std::string& s, T& v) {
    std::istringstream ss(s);
    return ss >> v;
}
```

Note: In C++ there is no difference between purely output parameters (only used to store a result) and input/output parameters (used to read data and write results back). As a result, even though the discussion here is centered on output parameters, the same principles apply to input/output parameters as well.

This function returns a flag to let the user know whether the conversion was successful, and the output value is stored in the argument `v`, which is a reference (`T&`). The function is then used as follows:

```
int v;
if (from_string("42", v)) {
    // do whatever with 'v'
} else {
    error("could not convert the string");
}
```

Note: In C++17, one may wish to return an `std::optional<T>` instead, which would be the optimal solution for the scalar case. However this solution does not vectorize well. Currently, `vec<D,std::optional<T>>` is not supported; it may work, but use it at your own risk.

The vectorization of such functions cannot be done with the automatic vectorization tools described above, so we will have to do it manually. It is rather simple, right? We only need to use a reference to an output vector `vec<D,T>&`:

```
template<std::size_t D = 1, typename U = std::string,
    typename T, typename enable = typename std::enable_if<
    std::is_same<meta::rtype_t<U>, std::string>::value
>::type>
vec<D,bool> from_string(const vec<D,U>& s, vec<D,T>& v) {
    vec<Dim,bool> res(s.dims);
    v.resize(s.dims);
    for (uint_t i : range(s)) {
        res.safe[i] = from_string(s.safe[i], v.safe[i]);
    }

    return res;
}
```

In this particular case, we use `std::enable_if<>` to make sure the input type is either a vector of strings or a view on such vector. We then return a vector of `bool` so the user can check the success of the conversion for each individual value separately. The function is then used as follows:

```
vec1s s = {"5", "-6", "9", "42"};

vec1i v;
vec1b r = from_string(s, v);

for (uint_t i : range(s)) {
    if (r[i]) {
        // do whatever with 'v[i]'
    } else {
        error("could not convert the string");
    }
}
```

The catch here is to support *views* as output arguments. Indeed, one may want to convert only part of a string vector with `from_string()` and store the result in a view, in which case our current implementation fails:

```
vec1s s = {"5", "-6", "9", "42"};

// Say we only want to convert values with 2 characters
vec1u id = where(length(s) == 2);

// This does *not* work:
vec1i v(s.dims); // resize output vector beforehand
vec1b r = from_string(s[id], v[id]);

// error: 'v[id]' is an r-value, cannot bind it to a reference 'vec<D,T>&'

// But this works:
vec1i v(s.dims); // resize output vector beforehand
vec1i tmp;       // create a temporary
vec1b r = from_string(s[id], tmp);
v[id] = tmp;     // assign temporary values to 'v'
```

Note: This issue also affects IDL, in a nastier way since IDL will not throw any error. It will store the output values in a automatically generated temporary vector, which is then discarded, so the values of the view are not modified... Oops! In IDL, this can only be solved by explicitly introducing a temporary vector and assigning it back to the view, as done in the example above. But C++ is smarter, and we can make this work! Read on.

Such type of problem arises whenever you write a function that takes a non-constant reference to a vector in order to modify its values. To support this type of usage with views, we need an argument type that can be either an "l-value" (a reference to a vector) or an "r-value" (a temporary view). This is exactly what the "universal reference" is for: `T&&`. Unfortunately, this universal reference requires an unconstrained type `T`. This means we loose all the implicit constraints on the type: it is no longer `vec<D,T>`, but simply `T`. Therefore we will have to specify these constraints explicitly using `std::enable_if<>`. And there are a lot of constraints! We want to make sure:

- that `T` is a vector or a view,
- that the number of dimensions of `T` is the same as the input vector of strings,
- that if `T` is an r-value, it must be a non-constant view,
- that if `T` is an l-value, it must be a non-constant reference (to a vector or a view).

Since these basic requirements will be the same for every vectorized function with output parameters, a specific trait is provided in vif to express all these constraints: `meta::is_compatible_output_type<In,Out>`. It is used in the following way:

```
template<std::size_t D = 1, typename U = std::string, typename T,
    typename enable = typename std::enable_if<
    std::is_same<meta::rtype_t<U>, std::string>::value && // this was there before
    meta::is_compatible_output_type<vec<D,U>,T>::value    // this is the new trait
>::type>
vec<D,bool> from_string(const vec<D,U>& s, T&& v) {
    // ...
}
```

In addition, here we need to differentiate the behavior of the function for the two cases: we want the "vector" version to automatically resize the output vector to the dimensions of the input vector, and the "view" version to simply check that the view has the same dimensions as the input vector. This is expected to be a common behavior for functions with output parameters, therefore a helper function is provided in vif to do just that: `meta::resize_or_check(v, d)`. This function resizes the vector `v` to the dimensions `d`, or, if `v` is a view, checks that its dimensions match `d`. The final, fully generic, vectorized function is therefore:

```
template<std::size_t D = 1, typename U = std::string, typename T,
    typename enable = typename std::enable_if<
    std::is_same<meta::rtype_t<U>, std::string>::value &&
    meta::is_compatible_output_type<vec<D,U>,T>::value
>::type>
vec<D,bool> from_string(const vec<D,U>& s, T&& v) {
    vec<Dim,bool> res(s.dims);
    meta::resize_or_check(v, s.dims);
    for (uint_t i : range(s)) {
        res.safe[i] = from_string(s.safe[i], v.safe[i]);
    }

    return res;
}
```

If you are in a case where there is no "input" vector to consider, and you simply want to write a function that modifies an existing vector's values (i.e., an input/output parameters), use the simpler `meta::is_output_type<T>` trait:

```
template<typename T, typename enable = typename std::enable_if<
    std::is_vec<T>::value && meta::is_output_type<T>::value
>::type>
```

(continues on next page)

(continued from previous page)

```
void twice(T&& v) {
    v[_] *= 2;
}
```

This traits only checks the last two conditions of `meta::is_compatible_output_type`, namely:

- that if `T` is an r-value, it must be a non-constant view,
- that if `T` is an l-value, it must be a non-constant reference (to a vector or a view).

6.8 Creating views

TODO

6.9 Metaprogramming helpers

TODO

CHAPTER 7

IDL equivalents

The interface of vif was designed to facilitate the migration from IDL, an interpreted language that, wile dated, is still commonly used in astronomy. Although IDL as a language suffers from a number of design issues, there is much good in its interface and API that one may wish to emulate in C++. But not all of it.

This page lists common language constructs in IDL and their C++ equivalent with vif. The following table is inspired from the xtensor documentation.

7.1 Crucial differences to always keep in mind

- C++ statements must end with a semicolon `;`. Line breaks are treated as white spaces.
- C++ is a statically typed language: a variable, once created, can never change its type.
- C++ variables must be *declared* before being used, and are destroyed automatically once the program exits the scope in which the variables were declared.
- C++ is a row-major language, IDL is a column-major language: for the same layout in memory (or on the disk), the dimensions of a vector in C++ are reversed compared to IDL. In particular, an image is accessed as `img[x,y]` in IDL, and `img(y,x)` in C++.
- C++ is case-sensitive, so `a` and `A` are different objects. Keywords and functions in C++ are always lowercase by convention.

7.2 Other notable differences

- C++ does not have a mechanism for finding where a function's code is based only on its name. If you put the function in a different file (a "header"), you must `#include "..."` this header explicitly in all other files that use this function.
- C++ has no procedures, but functions are allowed to have no return values.

- vif vectors can be empty, while IDL vectors cannot. It is possible to do operations with an empty vector (which have no cost and do nothing) if all the other vectors involved, if any, are also empty.
- C++ loops are *much* faster than in IDL, so there is no need to avoid them except to make the code shorter and more readable.
- C++ does not support keywords for functions, only normal arguments. A structure with named member values can be used instead.
- C++ does not have an equivalent for IDL's `common` variables (shared between functions), but you can use `static` to declare variables that survive between different calls to the same function. They just cannot be shared with another function.

7.3 Basics

IDL	C++ 11 - vif
`; a comment`	`// a comment`

`a = 1`
`s = 'foo'`
	`int_t a = 1;`
	`std::string s = "foo";`

`v1 = 1 & v2 = 'bar'`	`int_t v1 = 1; std::string v2 = "bar";`

`v = 1 + $`
`   2 + 3`
	`int_t v = 1 +`
	`   2 + 3;`

`a = 5`
`print, 'a=', a`
	`int_t a = 5;`
	`print("a=", a);`

`print, a, format='(F5.3)'`	No equivalent with `print()`, use `std::cout` or other formatting library.
`stop`	No equivalent. C++ programs either run or crash, but they cannot be stopped and resumed.
`r = execute('a = b')`	No equivalent.

`a = 1`
`delvar, a`
 	
`; 'a' no longer exist`
	`{`
	`   int_t a = 1;`
	`}`
	`// 'a' no longer exist`

7.4 Control flow

IDL	C++ 11 - vif

`if x lt y then begin`
`   ; ...`
`endif else begin`
`   ; ...`
`endelse`
	`if (x < y) {`
	`   // ...`
	`} else {`
	`   // ...`
	`}`

`for i=0, n-1 do begin`
`   ; ...`
`   break`
`   ; ...`
`   continue`
`   ; ...`
`endfor`
	`for (uint_t i : range(n)) {`
	`   // ...`
	`   break;`
	`   // ...`
	`   continue;`
	`   // ...`
	`}`

`array = ['foo','bar','blob']`
`foreach val, array do begin`
`   ; ...`
`endforeach`
	`vec1s array = {"foo","bar","blob"};`
	`for (std::string val : array) {`
	`   // ...`
	`}`

`while a gt b do begin`
`   ; ...`
`endfor`
	`while (a > b) {`
	`   // ...`
	`}`

`repeat begin`
`   ; ...`
`endrep until a gt b`
	`do {`
	`   // ...`
	`} while (a > b);`

`switch i of`
`1: print, 'one'`
`2: print, 'two'`
`3: print, 'three'`
`4: begin`
`   print, 'four'`
`   break`
`  end`
`else: print, 'other'`
`endswitch`
	`switch (i) {`
	`case 1: print("one");`
	`case 2: print("two");`
	`case 3: print("three");`
	`case 4:`
	`   print("four");`
	`   break;`
	
	`default: print("other");`
	`}`
	Note: only works with integers, no strings.

`case i of`
`   ; ...`
	No direct equivalent. Use `switch()` and be sure to call `break;` at the end of

7.5 Creating, accessing, modifying vectors

IDL	C++ 11 - vif

`v = fltarr(10)`
`v = fltarr(20)`
	`vec1f v(10);`
	`v.resize(20);`

`v = intarr(5)`
`d = double(v)`
	`vec1i v(10);`
	`vec1d d = v;`

`v = intarr(5)`
`v = double(v)`
	No equivalent. Types in C++ are *static*, cannot change `int` to `double`.

`v = intarr(6)`
`d = reform(v, 3, 2)`
	`vec1i v(6);`
	`vec2i d = reform(v, 2, 3);`

`v = intarr(2, 3)`
`v = reform(v, 3, 2)`
	`vec2i v(3, 2);`
	`v = reform(v, 2, 3);`

`v = intarr(6)`
`v = reform(v, 3, 2)`
	No equivalent. The number of dimensions of a vector is part of its type, and cannot change.

`v = [1,2,5,7]`
`v = [1,2,3]`
	`vec1i v = {1,2,5,7};`
	`v = {1,2,3};`

`n_elements(v)`	`v.size();`
`v = dindgen(5)`	`vec1d v = dindgen(5);`

`v = indgen(2,3)`
`v[0] = 1`
`v[0,2] = 2`
`v[0,*] = [2,5,6]`
`v[0,*:1] = [5,6]`
`v[0,1:*] = [5,6]`
`v[0,1:2] = [5,6]`
	`vec2i v = indgen(3,2);`
	`v[0] = 1;`
	`v(2,0) = 2;`
	`v(_,0) = {2,5,6};`
	`v(_-1,0) = {5,6};`
	`v(1-_,0) = {5,6};`
	`v(1-_-2,0) = {5,6};`

`v = intarr(5)`
`w = intarr(5)`
`id = [1,3,4]`
`v[id] = 1`
`v[id] = [-1,0,1]`
`w[id] = v[id]`
	`vec1i v(5);`
	`vec1i w(5);`
	`vec1u id = {1,3,4};`
	`v[id] = 1;`
	`v[id] = {-1,0,1};`
	`w[id] = v[id];`

7.6 Vector operations

IDL	C++ 11 - vif

Arithmetic:
`x = v + w`
`x = v - w`
`x = v * w`
`x = v / w`
`x = v ^ w`
`x = v mod w`
`x = v mod w`
	`x = v + w;`
	`x = v - w;`
	`x = v * w;`
	`x = v / w;`
	`x = pow(v, w);`
	`x = v % w;` (for integers)
	`x = fmod(v, w);` (for floats)

Comparison:
`x = v gt w`
`x = v ge w`
`x = v lt w`
`x = v le w`
`x = v && w`
`x = v || w`
`x = ~w`
`x = v > w`
`x = v < w`
	`x = v > w;`
	`x = v >= w;`
	`x = v < w;`
	`x = v <= w;`
	`x = v && w;`
	`x = v || w;`
	`x = !w;`
	`x = max(v, w);`
	`x = min(v, w);`

Bitwise:
`x = v and w`
`x = v or w`
`x = v xor w`
`x = not w`
	`x = v & w;`
	`x = v | w;`
	`x = v ^ w;`
	`x = ~w;`

Matrix:
`x = v # w`
	`matrix::mat<T> w, v;`
	`x = w * v;`
	or
	`vec<2,T> w, v;`
	`x = matrix::wrap(w)*matrix::wrap(v);`

`x = v ## w`	No direct equivalent. Do the operation explicitly with indices in a loop.

7.7 Finding values

IDL	C++ 11 - vif
`v = [1,2,3,4,5]` `id = where(v gt 3, cnt)` `if cnt ne 0 then v[id] = 0`	`vec1f v = {1,2,3,4,5};` `vec1u id = where(v > 3);` `v[id] = 0;` Note: empty vectors are allowed in vif, so the check for `cnt` is not needed.

CHAPTER 8

Introduction

Here we describe the set of helper functions that are part of the vif support library. These functions are not *essential* to the use of the library, in that the vector and view classes can be used on their own, but they are definitely *useful*. The functions in the support library are arranged inside modular components, which one may choose to use or not. This is the first repository in which one should look for existing, extensively tested functions, which are written to be as generic and efficient as possible.

In this documentation, all these functions are sorted into categories to help you discover new functions and algorithm. Alternatively, if you know the name of a function and would like to read its documentation, you may use the search feature.

In all the following sections, each function is presented and described separately, with its multiple overloads (if any) or "sibling" functions which share a similar role. Usage examples are also provided.

The signature of each function is given in a simplified form, both for conciseness and readability. In particular, we do not display the various template meta-programming tricks used to build the function's interface (`std::enable_if<>`), only the basic type of the arguments. The documention should therefore make it clear what the function's interface is, and specify what type of arguments are allowed and rejected (i.e., are views allowed? are non-arithmetic vector types rejected?). In that, we roughly follow the conventions of cppreference.com.

8.1 Interface conventions

To preserve consistency and help users `guess` the correct name or behavior of a function without having to look at the documentation all the time, there are a few interface conventions that all functions and types in vif should adhere to. Not all of these conventions are strictly enforced, and exceptions are allowed when there is a good justification.

- **Case.** All names must use lower case letters, with words separated by underscores _. For example `to_string()` is good, but not `ToString()` or `TO_STRING()`. Capital letters are reserved for macros only. This rule will be strictly enforced.
- **Language.** All names must be written in US English. This rule will be strictly enforced.
- **Spelling.** Whenever reasonable, words should not be abbreviated or truncated. Acronyms should be used sparingly. For example `to_string()` is good, but not `to_str()`. Counter example:

`sexagesimal_to_degrees()` is long to type and one has to remember it is `degrees` and not `degree`; this could be reasonably abbreviated to `sex_to_deg()`. But `sex2deg()` should be avoided.

- **Actions vs. tests.** Functions performing an "action" (to modify or create things) should include the verb of that action as the first word. For example `make_mask()`, `mask_circular_area()`, or `begin_calculation()`. Functions performing "tests" (by returning a `bool`) must start with the testing verb conjugated to the 3rd person. For example `is_finite()` or `begins_with_prefix()`, and not `finite()` or `begin_with_prefix()`, which could both indicate an "action" function instead. The counter example here is `vec::empty()`, which is a test that should have been spelled `vec::is_empty()`. Unfortunately the spelling `empty()` is used in the C++ standard library for *all* the containers, and therefore we choose to follow this spelling since most C++ programmers will be expecting it.
- **Classes vs. functions.** The naming convention is the same for classes, structs, or functions. Class names are not required to start with `C` (as in `CObject`), in fact this style is discouraged.
- **Class vs. struct.** Whenever possible and reasonable, use a `struct` over a `class`. The underlying semantic is: keep member values public for easy inspection by the user, make it clear what their purpose is and if/how/when they should be modified, and avoid excessive abstraction (no getters/setters). This may not apply to complex classes, or classes which must handle an insecure resource (like a raw pointer from a C library), in which case a `class` semantic (with private interface, etc.) is more adequate.
- **Member values.** Member values of a class or struct should follow the same naming conventions as functions. If part of the private interface of a class, they may end with an underscore to highlight that they are not part of the public interface (for example: `cached_`).
- **Free functions vs. member functions.** There is no definite convention, but favor free functions for implementing elaborate algorithms and data processing that use this class/struct, and prefer member functions for simple modifications or queries of the state of that class/struct.
- **Variables and function arguments.** There is no convention for the names of local variables, although lower case should usually be preferred.
- **Namespaces.** All functions and classes must reside in the `vif` namespace. Nested namespaces are allowed and encouraged.

CHAPTER 9

Generic functions

The vector and view classes are useful on their own. However, there are a number of tasks that one may needs to do quite often, like generating a sequence of indices, or sorting a vector, which would be tedious to rewrite each time they are needed. For this reason, the vif support library comes with a large set of utility functions to traverse, sort, rearrange, and select data inside vectors. In this section we list these various functions and describe the corresponding algorithms.

The support library introduces a global constant called `npos`. This is an unsigned integer whose value is the largest possible integer that the `uint_t` type can hold. It is meant to be used as an error value, or the value to return if no valid integer value would make sense. It is very similar in concept to the `std::string::npos` provided by the C++ standard library. In particular, it is worth noting that converting `npos` to a *signed* integer produces the value `-1`.

We now describe the functions provided by this module, sorted by categories.

9.1 Range-based iteration

Defined in header `<vif/core/range.hpp>`.

9.1.1 range

```
template<std::size_t D, typename T>
/*...*/ range(const vec<D,T>& v); // [1]

/*...*/ range(uint_t n); // [2]

/*...*/ range(uint_t i0, uint_t n); // [3]
```

This function returns a C++ *range*, that is, an object that can be used inside the C++ range-based `for` loop. This range will generate integer values starting from `0` (in [1], [2]) or `i0` (in [3]) to `v.size()` (in [1]) or `n` (in [2], [3]), that last value being *excluded* from the range. This nice way of writing an integer `for` loop actually runs as fast as (if not faster than) the classical way, and is less error prone.

The return value is a proxy class that holds the starting and ending point of the range, and offers `begin()` and `end()` function for iteration. Its type is of little importance.

Example:

```
vec1i v = {4,5,6,8};

// First version
for (uint_t i : range(v)) { // [1]
    // 'i' goes from 0 to 3
    v[i] = ...;
}

// Note that the loop above generates
// *indices* inside the vector, while:
for (int i : v) { /* ... */ }
// ... generates *values* from the vector.

// Second version
for (uint_t i : range(3)) { // [2]
    // 'i' goes from 0 to 2
    v[i] = ...;
}

// Third version
for (uint_t i : range(1,3)) { // [3]
    // 'i' goes from 1 to 3
    v[i] = ...;
}
```

9.2 Index manipulation

Defined in header `<vif/utility/generic.hpp>`.

9.2.1 mult_ids

```
template<std::size_t D>
vec1u mult_ids(const std::array<uint_t,D>& dims, uint_t i); // [1]

template<std::size_t D>
vec2u mult_ids(const std::array<uint_t,D>& dims, vec1u i); // [2]

template<std::size_t D, typename T>
vec1u mult_ids(const vec<D,T>& v, uint_t i); // [3]

template<std::size_t D, typename T>
vec2u mult_ids(const vec<D,T>& v, vec1u i); // [4]
```

This function converts a “flat” index `i` into an array of multidimensional indices, following the provided dimensions `dims` ([1] and [2]) or the provided vector `v` ([3] and [4]). The `flat_id` function does the inverse job.

[2] and [4] are the vectorized version of [1] and [3], respectively. The return value is a 2D vector of indices: the first dimension contains as many elements as `dims` ([2]) or the dimensions of `v` ([4]), and the second dimension contains as many elements as the provided index vector `i`.

Example:

```
vec2i v(2,3);
mult_ids(v,0); // {0,0}
mult_ids(v,1); // {0,1}
mult_ids(v,2); // {0,3}
mult_ids(v,3); // {1,0}
v[3] == v(1,0); // true
```

9.2.2 flat_id

```
template<std::size_t D, typename ... Args>
uint_t flat_id(const std::array<uint_t,D>& dims, Args&& ... args); // [1]

template<std::size_t D, typename TI>
uint_t flat_id(const std::array<uint_t,D>& dims, const vec<1,TI>& ids); // [2]

template<std::size_t D, typename T, typename ... Args>
uint_t flat_id(const vec<D,T>& v, Args&& ... args); // [3]

template<std::size_t D, typename T, typename TI>
uint_t flat_id(const vec<D,T>& v, const vec<1,TI>& ids); // [4]
```

This function converts a set of multidimensional indices into a "flat" index, following the provided dimensions `dims` ([1] and [2]) or the provided vector `v` ([3] and [4]). The `mult_ids` function does the inverse job.

In [1] and [3], the multidimensional indices are provided as separate arguments to the function. In [2] and [4] they are grouped inside an index vector.

Example:

```
vec2i v(2,3);
flat_id(v,0,0); // 0
flat_id(v,0,1); // 1
flat_id(v,0,2); // 2
flat_id(v,1,0); // 3
v(1,0) == v[3]; // true
```

9.2.3 increment_index_list

```
void increment_index_list(vec1u& ids, const uint_t& n); // [1]

void increment_index_list(vec1u& ids, const vec1u& n); // [2]
```

These functions perform *one* increment on the set of multidimensional indices `ids`, following the order in memory (i.e., last dimension is incremented first). In [1], each dimension has the same size `n`, while in [2] the dimensions may be different and are provided as a vector `n`. These functions allow the full traversal of the multidimensional space in a single loop, and are typically used to iterate on a multidimensional data set when the number of dimensions is either too large or unknown at compile time.

If called on the last allowed index, the function will set `ids` to zero, hence come back to the first index.

Example:

```
// Say we got some multidimensional data from a file
vec1d data;
vec1u dims = /* read from a file */;

uint_t nelem = 1;
for (uint_t d : dims) nelem *= d;

data.resize(nelem);

// Initialize the index vector to zero (first index)
vec1u ids(dims.size());

// Iterate in one single loop
for (uint_t i : range(nelem)) {
    // data[i] is the element at index (ids[0],ids[1],...)

    // Increment using [2]
    increment_index_list(ids, dims);
}
```

9.3 Integer sequences

Defined in header `<vif/utility/generic.hpp>`.

9.3.1 indgen

```
template<typename T = uint_t, typename ... Dims>
vec</*...*/,T> indgen(Dims&& ... ds);
```

This functions will create a new vector with values starting at `0` and increment linearly by steps of `1` until the end of the vector. Internally, the values are generated with the standard function `std::iota``. The number of dimensions of the resulting vector depends on the types `Args` of the arguments:

- Each argument of type `uint_t` increases the number of dimensions by one.
- Each argument of type `std::array<uint_t,D>` increases the number of dimensions by `D`.

The type of the values in the resulting vector is determined by the template parameter `T`, which defaults to `uint_t` if none is provided.

Example:

```
vec1u v = indgen(5);   // {0,1,2,3,4}
vec2u w = indgen(3,2); // {{0,1}, {2,3}, {4,5}}
```

9.4 Rearranging elements and dimensions

Defined in header `<vif/utility/generic.hpp>`.

9.4.1 flatten

```
template<std::size_t Dim, typename Type>
vec<1,Type> flatten(const vec<Dim,Type>& v); // [1]

template<std::size_t Dim, typename Type>
vec<1,Type> flatten(vec<Dim,Type>&& v); // [2]

template<typename T>
T flatten(T&& v); // [3]
```

This function transforms a multidimensional vector into a 1D vector ([1] and [2]; [3] is a no-op overload for scalars). The content in memory remains exactly the same, so the operation is fast. In particular, if the argument of this function is a temporary ([2]), this function is extremely cheap as it produces no copy. The `reform()` function does the inverse job, and more.

The provided argument `v` is unchanged ([1] and [3]).

Example:

```
vec2i v = {{1,2,3}, {4,5,6}};
vec1i w = flatten(v); // {1,2,3,4,5,6}
```

9.4.2 reform

```
template<std::size_t Dim, typename Type, typename ... Args>
vec</*...*/,Type> reform(const vec<Dim,Type>& v, Args&& ... args); // [1]

template<std::size_t Dim, typename Type, typename ... Args>
vec</*...*/,Type> reform(vec<Dim,Type>&& v, Args&& ... args); // [2]
```

This function transforms a vector into another vector, simply changing its dimensions. The content in memory remains exactly the same, so the operation is fast. In particular, if the argument of this function is a temporary ([2]), this function is extremely cheap as it produces no copy. However, the new dimensions have to sum up to the same number of elements as that in the provided vector. The `flatten()` function is a special case of `reform()` where all dimension are reformed into one, resulting in a 1D vector.

The provided argument `v` is unchanged in [1], but not [2]. The number of dimensions of the resulting vector depends on the types `Args` of the arguments:

- The starting dimension is `0`.
- Each argument of type `uint_t` increases the final dimension by one.
- Each argument of type `std::array<uint_t,D>` increases the final dimension by `D`.

Example:

```
vec1i v = {1,2,3,4,5,6};
vec2i w = reform(v, 2, 3); // {{1,2,3}, {4,5,6}}
```

9.4.3 reverse

```
template<typename Type>
vec<1,Type> reverse(vec<1,Type> v);
```

This function will return a copy of the provided vector, in which the order of all the elements is reversed. The original vector is unchanged. Only works with 1D vectors or views.

Example:

```
vec1i v = {1,2,3,4,5,6};
vec1i w = reverse(v); // {6,5,4,3,2,1}
```

9.4.4 shift, inplace_shift

```
template<typename Type>
vec<1,Type> shift(vec<1,Type> v, int_t n); // [1]

template<typename Type>
void inplace_shift(vec<1,Type>& v, int_t n); // [2]
```

`shift()` ([1]) returns a copy of the provided vector `v` where the elements are moved by circular shift of `n` elements. If `n` is positive, elements that would go beyond the bounds of the vector after the shift are moved to the beginning, with their order preserved. If `n` is negative, elements that would go beyond the beginning of the vector are placed at the end, with their order preserved. This function calls `std::rotate()`. The original vector is unchanged. Only works with 1D vectors or views.

`inplace_shift()` ([2]) performs the same operation as `shift()` but operates directly on the provided vector, which is therefore modified, but no copy is made so the operation is faster.

Example:

```
vec1i v = {1,2,3,4,5};

// [1]
vec1i sr1 = shift(v, 2);  // {4,5,1,2,3}
vec1i sr2 = shift(v, -2); // {3,4,5,1,2}

// [2]
inplace_shift(v, 2);
// v = {4,5,1,2,3}
```

9.4.5 transpose

```
template<typename Type>
vec<2,Type> transpose(const vec<2,Type>& v);
```

This function will transpose the provided 2D vector so that its dimensions are swapped. In other words, `v(i,j)` becomes `v(j,i)`. This is a matrix transposition. The original vector is unchanged.

Example:

```
vec2i v = {{1,2}, {3,4}, {5,6}};
vec2i w = transpose(v); // {{1,3,5}, {2,4,6}}
// now w(i,j) == v(j,i)
```

9.4.6 replicate

```
template<typename Type, typename ... Args>
vec</*...*/, meta::vtype_t<Type>> replicate(const Type& t, Args&& ... args); // [1]

template<std::size_t Dim, typename Type, typename ... Args>
vec</*...*/, meta::rtype_t<Type>> replicate(const vec<Dim,Type>& t, Args&& ... args);␣
↪// [2]
```

This function will take the provided scalar ([1]) or vector ([2]), and replicate it multiple times according to the provided additional parameters, to generate additional dimensions.

The number of dimensions of the resulting vector depends on the types `Args` of the arguments:

- The starting dimension is `0` ([1]) or `Dim` ([2]).
- Each argument of type `uint_t` increases the final dimension by one.
- Each argument of type `std::array<uint_t,D>` increases the final dimension by `D`.

Example:

```
// [1]
vec1i v = replicate(2, 5);
// v = {2,2,2,2,2}, or 5 times 2

vec2i w = replicate(2, 3, 2);
// w = {{2,2},{2,2},{2,2}}, or 3 x 2 times 2

vec3u x = replicate(1u, w.dims, 5);
// equivalent to:
// x = replicate(1u, 3, 2, 5);

// [2]
vec2i z = replicate(vec1i{1,2}, 3);
// z = {{1,2},{1,2},{1,2}}, or 3 times {1,2}

// Note that it is not possible to just use a plain initializer list
// since its type cannot be deduced with current C++ rules
vec2i z = replicate({1,2}, 3); // error
```

9.4.7 sort, inplace_sort

```
template<std::size_t Dim, typename Type>
vec1u sort(const vec<Dim,Type>& v); // [1]

template<std::size_t Dim, typename Type, typename F>
vec1u sort(const vec<Dim,Type>& v, F&& comp); // [2]

template<std::size_t Dim, typename Type>
void inplace_sort(vec<Dim,Type>& v); // [3]

template<std::size_t Dim, typename Type, typename F>
void inplace_sort(vec<Dim,Type>& v, F&& comp); // [4]
```

`sort()` returns a vector of indices for the provided vector `v`, ordered such that the pointed values are sorted by increasing value ([1]) or following the provided comparison function ([2]). The number of returned indices is the same as the number of values in `v`. The original vector is not modified. If two elements of `v` compare equal, their respective order in the vector will be unchanged (this function uses `std::stable_sort()`).

`inplace_sort()` directly modifies the order of the values inside the vector, and returns nothing. It is fastest, but less powerful.

In [2] and [4], the comparator function `comp(x,y)` must return `true` if `x` should be placed after `y` after the sort.

Warning: The comparison function `comp` must provide a *strict total ordering*, otherwise the behavior of the function is undefined. See cppreference.com for more information. In brief, this means that any value can only be "equal", "lesser", or "greater" than any other value. With a comparison function returning simply `x < y`, this requirement is not met for `float` and `double` because of the special value "not-a-number", `NaN`, which is neither. [1] and [3] use the default comparator for vif vectors, in which this issue is solved by considering `NaN` as "greater than" positive infinity. `NaN` values will thus be placed at the end of a sorted vector. To take advantage of this implementation, use `vec<Dim,Type>::comparator_less{}(x,y)` and `vec<Dim, Type>::comparator_greater{}(x,y)` instead of `x < y` and `x > y` inside your custom comparison functions. This is unnecessary for integer types and strings.

Example:

```
// [1]
vec1i v = {1,5,6,3,7};
vec1u id = sort(v); // {0,3,1,2,4}
// v[id] = {1,3,5,6,7} is sorted

// Now, 'id' can also be used to modify the order of
// another vector of the same dimensions.

// [3]
inplace_sort(v);
v; // {1,3,5,6,7} is sorted

// [4]
vec1f v1 = {1.0,2.0,3.0,4.0, 5.0,6.0};
vec1f v2 = {3.0,0.5,1.0,fnan,0.0,0.0};

// Sort 'v1+v2'
vec1u id = uindgen(v1.size());
inplace_sort(id, [&](uint_t i1, uint_t i2) {
    return vec1f::comparator_less{}(v1[i1]+v2[i1], v1[i2]+v2[i2]);
});

// (v1+v2)[id] = {2.5,4,4,5,6,nan}
// v1[id]      = {2.0,1,3,5,6,4}
// v2[id]      = {0.5,3,1,0,0,nan}

// Sort first by 'v2', then 'v1'
id = uindgen(v1.size());
inplace_sort(id, [&](uint_t i1, uint_t i2) {
    if (vec1f::comparator_less{}(v2[i1], v2[i2])) {
        return true;
    } else if (vec1f::comparator_greater{}(v2[i1], v2[i2])) {
        return false;
    } else {
        return vec1f::comparator_less{}(v1[i1], v1[i2]);
    }
```

(continues on next page)

(continued from previous page)

```
});

// v1[id] = {5,6,2.0,3,1,4}
// v2[id] = {0,0,0.5,1,3,nan}
```

9.4.8 is_sorted

```
template<std::size_t Dim, typename Type>
bool is_sorted(const vec<Dim,Type>& v);
```

This function just traverses the whole input vector and checks if its elements are sorted by increasing value.

Example:

```
// First version
vec1i v = {1,5,6,3,7};
is_sorted(v); // false
inplace_sort(v);
// v = {1,3,5,6,7}
is_sorted(v); // true
```

9.4.9 append, prepend

```
template<std::size_t N, std::size_t Dim, typename Type1, typename Type2>
void append(vec<Dim,Type1>& v, const vec<Dim,Type2>& t); // [1]

template<std::size_t N, std::size_t Dim, typename Type1, typename Type2>
void prepend(vec<Dim,Type1>& v, const vec<Dim,Type2>& t); // [2]
```

These functions behave similarly to `vec::push_back()`, in that they will add new elements at the end ([1]), but also at the beginning ([2]) of the provided vector `v`. However, while `vec::push_back()` can only add new elements from a vector that is one dimension *less* than the original vector (or a scalar, for 1D vectors), these functions will add new elements from a vector of the *same* dimension. These functions are also more powerful than `vec::push_back`, because they allow you to choose along which dimension the new elements will be added using the template parameter `N` (note that this parameter is useless and therefore does not exist for 1D vectors). The other dimensions must be otherwise identical.

The first argument `v` cannot be a view.

Example:

```
// For 1D vectors
vec1i v = {1,2,3};
vec1i w = {4,5,6};
append(v, w);
// v = {1,2,3,4,5,6}
prepend(v, w);
// v = {4,5,6,1,2,3,4,5,6}

// For multidimensional vectors
vec2i x = {{1,2}, {3,4}};          // x is (2x2)
vec2i y = {{0}, {0}};              // y is (2x1)
vec2i z = {{5,6,7}};               // z is (1x3)
```

(continues on next page)

(continued from previous page)

```
append<1>(x, y);
// x = {{1,2,0}, {3,4,0}}          // x is (2x3)
prepend<0>(x, z);
// x = {{5,6,7}, {1,2,0}, {3,4,0}} // x is (3x3)
```

9.4.10 remove, inplace_remove

```
template<std::size_t Dim, typename Type>
vec<Dim,Type> remove(vec<Dim,Type> v, const vec1u& ids); // [1]

template<std::size_t Dim, typename Type>
void inplace_remove(vec<Dim,Type>& v, vec1u ids); // [2]
```

`remove()` ([1]) will return a copy of the provided vector `v` with the elements at the indices provided in `id` removed. `inplace_remove()` ([2]) removes values directly from the provided vector, and is therefore faster.

The first argument `v` cannot be a view. The values in `ids` are checked to ensure they represent valid indices in `v`; if not, a run time error is raised.

Example:

```
// [1]
vec1i v = {4,5,2,8,1};
vec1i w = remove(v, {1,3}); // {4,2,1}

// [2]
inplace_remove(v, {1,3});
// v = {4,2,1}
```

9.5 Finding elements

Defined in header `<vif/utility/generic.hpp>`.

9.5.1 where

```
template<std::size_t Dim, typename Type>
vec1u where(const vec<Dim,Type>& v);
```

This function will scan the `bool` vector (or view) provided in argument, will store the “flat” indices of all the elements which are `true`, and will return all these indices in a vector. This is a very useful tool to filter and selectively modify vectors, and probably one of the most used function of the whole library.

Example:

```
vec1i v = {4,8,6,7,5,2,3,9,0};

// We want to select all the elements which are greater than 3.
// We use where() to get their indices:
vec1u id = where(v > 3); // {0,1,2,3,4,7}

// Now we can check:
```

(continues on next page)

(continued from previous page)

```
v[id]; // {4,8,6,7,5,9}, good!

// The argument of where() can be quite complex:
id = where(v < 3 || (v > 3 && v % 6 < 2)); // now guess

// It can also involve multiple vectors, as long as they have
// the same dimensions.
vec1i w = {9,8,6,1,-2,0,8,5,1};
id = where(v > w || (v + w) % 5 == 0);

// The returned indices are then valid for both v and w.
v[id]; // {8,6,7,5,2,9}
w[id]; // {8,6,1,-2,0,5}
```

Note that, when called on a view, `where()` will return indices inside the view itself, and *not* inside the viewed vector:

Example:

```
vec1i v = {4,8,6,7,5,2,3,9,0};

// Select all the elements greater than 3.
vec1u id1 = where(v > 3);

for (uint_t i : range(1)) {
    // We want to apply another selection on top of the first one.
    // Say we now want only those elements which are even (when i=0)
    // or odd (when i=1):
    vec1u id2 = where(v[id1] % 2 == i);

    // Here 'id2' points inside 'v[id1]', not 'v'!
    // To access the correspond values in 'v', one must write:
    v[id1[id2]] = /* ... */;

    // This is dangerous, because 'v[id2]' is perfectly valid,
    // yet makes absolutely no sense.
}
```

In general, such situations can be avoided by only calling `where()` at the last possible moment, as shown in the example below.

Example:

```
vec1b base = v > 3; // only a 'bool' vector for now

for (uint_t i : range(1)) {
    vec1u id = where(base && v % 2 == i);

    // Now we can access 'v' directly:
    v[id] = /* ... */;
}
```

This is often more readable and less error prone, however it may also be less efficient, particularly if the first `where()` reduced significantly the number of elements to work with (e.g., if `v` contained millions of elements, and only a few are greater than `3`). A typical case where these situations arise is when binning multidimensional data, for example to build a 2D histogram. Then, a much more efficient approach is to use the `histogram()` function and its siblings. Despite the name, these powerful function can be used for purposes other than histograms.

9.5.2 where_first, where_last

```
template<std::size_t Dim, typename Type>
uint_t where_first(const vec<Dim,Type>& v); // [1]

template<std::size_t Dim, typename Type>
uint_t where_last(const vec<Dim,Type>& v); // [2]
```

These functions will scan the `bool` vector (or view) provided in argument, and return the "flat" index of the first ([1]) or last ([2]) element which is `true`, or `npos` if all are `false`.

When used with views, the same caution applies as for `where()`: the returned index points inside the view itself, not inside the viewed vector.

Example:

```
vec1i v = {4,8,6,7,5,2,3,9,0};
// We want to select the first element which is greater than 3
uint_t id;
id = where_first(v > 3); // 0
v[id];                   // 4
id = where_last(v > 3);  // 7
v[id];                   // 9
```

9.5.3 complement

```
template<std::size_t Dim, typename Type>
vec1u complement(const vec<Dim,Type>& v, const vec1u& ids);
```

This function works in parallel with `where()`. Given a vector `v` and a set of "flat" indices `id`, it will return the complementary set of indices inside this vector, i.e., all the indices of `v` that are *not* present in `id`. The values of `v` are unused, only its dimensions are read.

Example:

```
vec1i v = {1,5,6,3,7};
vec1u id = where(v > 4); // {1,2,4}
vec1u cid = complement(v, id); // {0,3}
```

9.5.4 match

```
template<std::size_t D1, std::size_t D2, typename Type1, typename Type2>
void match(const vec<D1,Type1>& v1, const vec<D2,Type2>& v2, vec1u& id1, vec1u& id2);
```

This function returns the indices of the elements with equal values in `v1` and `v2`. In practice, it traverses `v1` and, for each value in `v1`, looks for elements in `v2` that have the same value. If one is found, the index of the element of `v1` is added to `id1`, and the index of the element of `v2` is added to `id2`. If other matches are found in `v2` for this same value, they are ignored, therefore only the *first* match is returned. Then the function goes on to the next value in `v1`. The two vectors `v1` and `v2` need not be the same size.

When used with views, the same caution applies as for `where()`: the returned indices point inside the views themselves, not inside the viewed vectors.

Example:

```
vec1i v = {7,6,2,1,6};
vec1i w = {2,6,5,3};
vec1u id1, id2;
match(v, w, id1, id2);
id1; // {1,2,4}
id2; // {1,0,1}
v[id1] == w[id2]; // always true
```

9.5.5 set_intersection, set_intersection_sorted, set_union, set_union_sorted

```
template<std::size_t D1, std::size_t D2, typename Type1, typename Type2>
vec<1,/*...*/> set_intersection(vec<D1,Type1> v1, vec<D2,Type2> v2); // [1]

template<std::size_t D1, std::size_t D2, typename Type1, typename Type2>
vec<1,/*...*/> set_intersection_sorted(const vec<D1,Type1>& v1, const vec<D2,Type2>&␣
↪v2); // [2]

template<std::size_t D1, std::size_t D2, typename Type1, typename Type2>
vec<1,/*...*/> set_union(vec<D1,Type1> v1, vec<D2,Type2> v2); // [3]

template<std::size_t D1, std::size_t D2, typename Type1, typename Type2>
vec<1,/*...*/> set_union_sorted(const vec<D1,Type1>& v1, const vec<D2,Type2>& v2); //␣
↪[4]
```

These functions return a 1D vector containing all the values that exist in both ([1], [2]) or either of ([3], [4]) `v1` and `v2`. These values are returned sorted. If a value is found `n1` times in `v1` and `n2` times in `v2`, [1] and [2] will return this value `min(n1,n2)` times, while [3] and [4] will return this value `max(n1,n2)` times. The two vectors do not need to have the same size, and the functions are symmetric: returned values are the same if `v1` and `v2` are swapped.

The algorithms used internally (`std::set_intersection()` and `std::set_union()`) operate on sorted vectors. [1] and [3] will automatically sort the input vectors, so there are no pre-requirement on their ordering. [2] and [4] will assume that the input vectors are already sorted, all will thus be faster.

The type of the returned vector is the common type between `v1` and `v2`, namely, whatever type results of an operation like `v1[0]+v2[0]`.

Example:

```
vec1i v = {1,2,3,3,3,4,5};
vec1i w = {2,3,3,4,6};

set_insersection(v, w); // {2,3,3,4}
set_union(v, w);        // {1,2,3,3,3,4,5,6}
```

9.5.6 unique_ids, unique_ids_sorted, unique_values, unique_values_sorted

```
template<std::size_t Dim, typename Type>
vec1u unique_ids(const vec<Dim,Type>& v); // [1]

template<std::size_t Dim, typename Type>
vec1u unique_ids(const vec<Dim,Type>& v, const vec1u& sid); // [2]

template<std::size_t Dim, typename Type>
vec1u unique_ids_sorted(const vec<Dim,Type>& v); // [3]
```

(continues on next page)

(continued from previous page)

```
template<std::size_t Dim, typename Type>
vec<1,meta::rtype_t<Type>> unique_values(const vec<Dim,Type>& v); // [4]

template<std::size_t Dim, typename Type>
vec<1,meta::rtype_t<Type>> unique_values(const vec<Dim,Type>& v, const vec1u& sid); //
↪ [5]

template<std::size_t Dim, typename Type>
vec<1,meta::rtype_t<Type>> unique_values_sorted(const vec<Dim,Type>& v); // [6]
```

These functions will traverse the provided vector `v` and find all the unique values. Functions [1] to [3] store the *indices* of these values and return them inside an index vector. If a value is present more than once, the index of the first one will be returned. Functions [4] to [6] directly return the values themselves, rather than indices.

Internally, the algorithm needs to sort `v`. To optimize execution, several versions of these functions are provided which handle the sorting differently. Functions [1] and [4] will automatically sort the vector, so there is no pre-requirement on `v`. Functions [2] and [5] takes a second argument `id` that contains indices that will sort `v`. In particular, `id` can be the return value of `sort(v)`. Lastly, functions [3] and [6] assume that `v` is already sorted, and are thus the fastest of the three.

When used with views, the same caution applies for functions [1] to [3] as for `where()`: the returned indices point inside the views themselves, not inside the viewed vectors.

Example:

```
// For an non-sorted vector [1]
vec1i w = {5,6,7,8,6,5,4,1,2,5};
vec1u u = unique_ids(w); // {7,8,6,0,1,2,3}
w[u]; // {1,2,4,5,6,7,8} only unique values

// Providing a sorting vector [2]
vec1u s = sort(w);
vec1u u = unique_ids(w, s); // {7,8,6,0,1,2,3}
w[u]; // {1,2,4,5,6,7,8} only unique values

// For a sorted vector [3]
vec1i v = {1,1,2,5,5,6,9,9,10};
vec1u u = unique_ids_sorted(v); // {0,2,3,5,6,8}
v[u]; // {1,2,5,6,9,10} only unique values
```

9.5.7 is_any_of

```
template<typename Type1, std::size_t Dim2, typename Type2>
bool is_any_of(const Type1& v1, const vec<Dim2,Type2>& v2); // [1]

template<std::size_t Dim1, typename Type1, std::size_t Dim2, typename Type2>
vec<Dim1,bool> is_any_of(const vec<Dim1,Type1>& v1, const vec<Dim2,Type2>& v2); // [2]
```

Function [1] looks inside `v2` if there is any value that is equal to `v1`. If so, it returns `true`, else it returns `false`. Function [2] is the vectorized version of [1], and executes this search for each value of `v1`, then returns a `bool` vector containing the results.

There are no pre-requirements on `v1` or `v2`.

Example:

```
vec1i v = {7,4,2,1,6};
vec1i d = {5,6,7};
vec1b b = is_any_of(v, d); // {true, false, false, false, true}
```

9.5.8 bounds, lower_bound, upper_bound

```
template<typename T, std::size_t Dim, typename Type>
uint_t lower_bound(const vec<Dim,Type>& v, T x); // [1]

template<typename T, std::size_t Dim, typename Type>
uint_t upper_bound(const vec<Dim,Type>& v, T x); // [2]

template<typename T, std::size_t Dim, typename Type>
std::array<uint_t,2> bounds(const vec<Dim,Type>& v, T x); // [3]

template<typename T, typename U, std::size_t Dim, typename Type>
std::array<uint_t,2> bounds(const vec<Dim,Type>& v, T x1, U x2); // [4]
```

These functions use a binary search algorithm to locate the element in the input vector `v` that is equal to or closest to the provided value `x`, which must be a scalar. The binary search assumes that the elements in the input vector are *sorted* by increasing value. This algorithm also assumes that, if the input vector contains floating point numbers, none of them is `NaN`.

`lower_bound()` ([1]) locates the last element in `v` that is *less or equal* to `x`. If no such element is found, `npos` is returned.

`upper_bound()` ([2]) locates the first element in `v` that is *greater than* `x`. If no such element is found, `npos` is returned.

`bounds()` ([3]) combines what `lower_bound()` and `upper_bound()` do, and returns both indices in an array. The second overload of `bounds()` ([4]) calls `lower_bound()` to look for `x1`, and `upper_bound()` to look for `x2`.

Example:

```
vec1i v = {2,5,9,12,50};
bounds(v, 0);   // {npos,0}
bounds(v, 9);   // {2,3}
bounds(v, 100); // {4,npos}
```

9.5.9 equal_range

```
template<typename T, std::size_t Dim, typename Type>
std::array<uint_t,2> equal_range(const vec<Dim,Type>& v, T x);
```

This function uses a binary search algorithm to locate all the values in the input vector `v` that are equal to `x`. The binary search assumes that the elements in the input vector are *sorted* by increasing value. This algorithm also assumes that, if the input vector contains floating point numbers, none of them is `NaN`.

The function returns the first and last indices of the values equal to `x`. If no such value is found, the two indices will be set to `npos`.

If `v` is not sorted, an alternative is to call `where(v == x)`; this should be faster than sorting `v` and then using `equal_range()`.

Example:

```
vec1i v = {2,2,5,9,9,9,12,50};
equal_range(v, 9); // {3,5}

// The above is a faster version of:
where(v == 9);
```

9.5.10 astar_find

```
bool astar_find(const vec2b& map, uint_t& x, uint_t& y);
```

This function uses the A* ("A star") algorithm to look inside a 2D boolean map `m` and, starting from the position `x` and `y` (i.e. `m(x,y)`), find the closest point that has a value of `true`. Once this position is found, its indices are stored inside `x` and `y`, and the function returns `true`. If no element inside `m` is `true`, then the function returns `false`.

Example:

```
// Using 'o' for true and '.' for false, assume we have the following boolean map,
// and that we start at the position indicated by 'S', the closest point whose␣
↪coordinates
// will be returned by astar_find() is indicated by an 'X'

vec2b m;

//   0123456789  13
// 0 .................
// 1 .................
// 2 .................
// 3 ...ooooo.........
// 4 ...ooooo.........
// 5 ...ooooo.........
// 6 ...ooooo.........
// 7 ...ooooX.........
// 8 .............S...
// 9 .................
//   .................

uint_t x = 13, y = 8;
astar_find(m, x, y);
x; // 7
y; // 7
```

9.6 Error checking

Defined in header `<vif/core/error.hpp>`.

9.6.1 vif_check

```
template<typename ... Args>
void vif_check(bool b, Args&& ... args);
```

This function (in fact a C++ macro) makes error checking easier. When called, it checks the value of `b`. If `b` is `true`, then nothing happens and the values of `args` are not evaluated. However if `b` is `false`, then the current file and line are printed to the standard output, followed by the other arguments of this function, which are supposed to compose an error message explaining what went wrong, and the program is immediately stopped. This function is used everywhere in the vif library to ensure that certain pre-conditions are met before doing a calculation, and it is *essential* to make the program stop in case something is unexpected rather than letting it run hoping for the best (and often getting the worst).

Since this function is actually implemented by a preprocessor macro, one should not worry about its performance impact beside the cost of evaluating `b`. Performances will only be affected when something goes wrong and the program is about to stop anyway.

In addition, if the vif library was configured accordingly, this function can print the "backtrace" of the program that lead to the error. This backtrace lists which functions or lines of code the program was executing when the error occured, tracing back the offending line all the way up from the `main()` function. This can be very useful to identify the source of the problem, but is only available if debugging informations are stored inside the compiled program. Note that this only affects the size of the executable on disk: debugging informations do not alter performances. Lastly, the backtrace may not be sufficient to understand what went wrong, and one may need to use the debugger.

```
// Suppose 'v' is read from the command line arguments.
vec1i v = /* read from somewhere unsafe */;

// The rest of the code needs at least 3 elements in the
// vector 'v', so we need to check that first.
vif_check(v.size() >= 3, "this algorithm needs at least 3 values in the input vector,
↪"
    "but only ", v.size(), " were found");

// If we get past this point, we can proceed safely to use
// the first three elements
print(v[0]+v[1]+v[3]);

// ... and we can safely disable index checking using the
// 'safe' interface of the vector
print(v.safe[0]+v.safe[1]+v.safe[3]);
```

CHAPTER 10

Command line arguments

Defined in header `<vif/utility/argv.hpp>`.

10.1 read_args

The biggest problem of C++ programs is that, while they are fast to execute, they can take a long time to *compile*. For this reason, C++ is generally not a productive language in situations where the code has to be written by *trial and error*, a process that involves frequently changing the behavior or starting point of a program.

There are ways around this issue. One in particular is called *data driven* programming: the behavior of a program depends on the data that are fed to it. The simplest way to use this paradigm is to control the program through "command line arguments". The C++ language provides the basic bricks to use command line arguments, but the interface is inherited from C and lacks severely in usability.

For this reason we introduce in vif a single function, `read_args()`, that uses these bricks to provide a simple and concise interface to implement command line arguments in a program. The first two arguments of the function (`argc, argv`) must be the arguments of the `main()` function, in the same order. The following argument must be `arg_list(...)`, inside of which one must list all the variables that can be modified through the command line interface. These variables can be of any type, as long as it is possible to convert a string into a variable of this type.

10.2 Usage examples

Let us illustrate this with an example. Assume that we want to build a simple program that will print to the terminal the first `n` powers of two, with `n` being specified by the user of the program. Here is how this would be done with `read_args()`:

```
# include <vif.hpp>

// This is the standard entry point of every C++ program.
// The signature of the main function is imposed by the C++ standard
int vif_main(int argc, char* argv[]) {
```

(continues on next page)

(continued from previous page)

```
    // Declare the main parameters of the program, in this case
    // the number of powers of two to display, 'n'.
    uint_t n = 1;

    // Then read command line arguments...
    read_args(argc, argv, arg_list(n));

    // Now we can go on with the program, using 'n' normally
    print(pow(2.0, findgen(n)+1));

    // And quit gracefully
    return 0;
}
```

By just adding this line

```
read_args(argc, argv, arg_list(n));
```

we exposed the variable `n` to the public: everyone that runs this program can modify the value of `n` to suit their need. Simple! Assuming the name of the compiled program is `show_pow2`, then the program is ran the following way:

```
# First try with no parameter.
# 'n' is not modified, and keeps its default value of 1.
./show_pow2
# output:
# 2

# Then we change 'n' to 5.
./show_pow2 n=5
# output:
# 2 4 8 16 32
```

The advantages of this approach are immediate. Instead of recompiling the whole program just to change `n`, we exposed it in the program arguments. We then compiled the program *once*, and changed its behavior without ever recompiling. This can save a lot of time, for example when trying to figure out what is the best value of a parameter in a given problem (i.e., tweaking parameters of an algorithm). And of course, this is most useful when writing *tools* with configurable options.

Within the `arg_list()`, one can put as many C++ variables as needed. The function will recognize their name, and when the program is executed it will try to find a command line argument that matches. If it finds one, it tries to convert its value into the type of the variable, and if successful, store this new value inside the variable. In all other cases, the variable is not modified. It is therefore important to give a meaningful default value to each variable!

In the example above, we chose to expose a simple integer. But in fact, the interface can be used to expose any type, provided that there is a way to convert a string into a value of this type. In particular, this is the case for vectors. The values of the vector must be separated by commas, (*without any space*, unless you put the whole argument inside quotes), and surrounded by brackets `[...]`. Again, let us illustrate this with an example. We will modify the previous program to allow it to show not only powers of `2`, but the powers of multiple, arbitrary numbers. Note: in the following, we will not repeat the whole `main()` function, just the important bits.

```
// The number of powers of two to display
uint_t n = 1;
// The powers to display
vec1f p = {2};

// Read command line arguments
```

(continues on next page)

(continued from previous page)

```
read_args(argc, argv, arg_list(n, p));

// Go on with the program
for (float v : p) {
    print(pow(v, findgen(n)+1));
}
```

The program can now change the powers it displays, for example:

```
# We keep 'n' equal to 5, and we show the powers of 2, 3 and 5.
./show_pow2 n=5 p=[2,3,5]
# output:
# 2 4 8 16 32
# 3 9 27 81 243
# 5 25 125 625 3125

# It is possible to use spaces inside the [...], but then you must add quotes:
./show_pow2 n=5 p="[2, 3, 5]"
```

Now, you may think that `p` is not a very explicit name for this last parameter. It would be clearer if we could call it `pow`. Unfortunately, `pow` is already the name of a function in C++, so we cannot give this name to the variable. However, the `read_args()` interface allows you to manually give a name to any parameter using the `name()` function. Let us do that and modify the previous example.

```
// The number of powers of two to display
uint_t n = 1;
// The powers to display, we still call it 'p' in the program
vec1f p = {2};

// Read command line arguments
read_args(argc, argv, arg_list(n, name(p, "pow")));

// Go on with the program
for (float v : p) {
    print(pow(v, findgen(n)+1));
}
```

Now we will write instead:

```
./show_pow2 n=5 pow=[2,3,5]
# output:
# 2 4 8 16 32
# 3 9 27 81 243
# 5 25 125 625 3125
```

10.3 Flags

Often, command line options are "flags". These are boolean variables that are `false` by default, but can be changed to `true` to enable some specific functionality. For example, setting `verbose=1` can be used to tell the program to display information in the terminal about its progress. To simplify usage of these flags, `read_args()` allows an alternative syntax where specifying `verbose` without any equal sign in the arguments is equivalent to `verbose=1`:

```
./my_program verbose
# ... is equivalent to:
./my_program verbose=1
```

There is no shortcut for `var=0`.

10.4 Alternative syntax

In the examples above, command lines arguments are specified as `variable=value`. This is the tersest available syntax. However, most linux programs tend to use dashes (`-`) to identify command line arguments; for example `-variable=value` or `--variable=value`. To avoid confusing users, `read_args()` supports both ways of writing command line arguments; dashes can be used but are not mandatory.

CHAPTER 11

String manipulation

11.1 String conversions

Defined in header <vif/core/string_conversion.hpp>.

11.1.1 to_string, to_string_vector

```
template<typename Type>
std::string to_string(const Type& v); // [1]

template<std::size_t Dim, typename Type>
vec<Dim,std::string> to_string_vector(const vec<Dim,Type>& v); // [2]
```

The function [1] will convert the value v into a string. This value can be of any type, as long as it is convertible to string. In particular, v can be a vector, in which case the output string will contain all the values of the vector, separated by commas, and enclosed inside curly braces: "{a,b,c,d,....}".

Example:

```
to_string(2);             // "2"
to_string(true);          // "1"
to_string("foo");         // "foo"
to_string(vec1i{2,5,9}); // "{2, 5, 9}"
```

The function [2] allows you to convert *each value* of a vector into a separate string, and store them in a string vector.

```
to_string(vec1i{2,5,9});        // "{2, 5, 9}"
to_string_vector(vec1i{2,5,9}); // {"2", "5", "9"}
```

Internally, the argument is converted to a string using the std::ostream operator<<. This means that most types from the standard C++ or external C++ libraries will be convertible to a string out of the box. If you encounter some errors for a particular type, this probably means that the operator<< is missing and you have to write it yourself. Here is how you would do that:

```
// We want to make this structure printable
struct test {
    std::string name;
    int i, j;
};

// We just need to write this function
std::ostream& operator<< (std::ostream& o, const test& t) {
    o << t.name << ":{i=" << i << " j=" << j << "}";
    return o; // do not forget to always return the std::ostream!
}

// The idea is always to rely on the existence of an operator<<
// function for the types that are contained by your structure
// or class. In our case, 'std::string' and 'int' are already
// conertible to string, This is the standard C++ way of doing
// string conversions, but it can be annoying to use regularly
// because the "<<" are taking a lot of screen space.
// 'to_string()' makes it much more convenient.

// Now we can convert!
test t = {"toto", 5, 12};
std::string s = to_string(t);
s; // "toto:{i=5 j=12}"
```

11.1.2 format::precision, format::scientific, format::fixed

```
template<typename Type>
/* ... */ format::scientific(const Type& v); // [1]

template<typename Type>
/* ... */ format::precision(const Type& v, uint_t ndigit); // [2]

template<typename Type>
/* ... */ format::fixed(const Type& v); // [3]
```

The `to_string()` and `to_string_vector()` functions adopt a default format for converting numbers into strings. While integers have a unique and natural string representation, floating point numbers often require a choice regarding the number of significant digits, and whether scientific notation should be used. By default, these functions follow the behavior of `std::ostream`, which is to only use scientific notation when the number would be "too big" (or "too small").

Function [1], `format::scientific()`, will specify that it's argument `v` *must* be formated using the scientific notation.

Example:

```
double v = 0.15;
to_string(v);                     // "0.15"
to_string(format::scientific(v)); // "1.500000e-01"
```

Function [2], `format::precision()`, will specify that it's first argument `v` *must* be formated using `ndigit` digits. Normally, "digits" include numbers on either side of the decimal separator, so `"3.15"`, `"31.5"`, and `"315"` are all three digits. When not in scientific format, trailing zeros after the decimal separator will still be removed, so the total number of digits may still be less than `ndigit`.

Example:

```
double v = 0.15;
to_string(v);                          // "0.15"
to_string(format::precision(v, 8)); // "0.15"

v = 0.123456789123456789;
to_string(v);                          // "0.123457"
to_string(format::precision(v, 8)); // "0.12345679"
```

Function [3], `format::fixed()`, will format the value with a fixed number of digits after the decimal separator. Trailing zeroes will not be removed. This is best used in combination with `format::precision`, which then specifies how many digits to keep after the decimal separator (digits before the separator do not count).

Example:

```
double v = 0.15;
to_string(v);                                        // "0.15"
to_string(format::fixed(format::precision(v, 8))); // "0.15000000"

v = 3150.15;
to_string(v);                                        // "3150.15"
to_string(format::fixed(format::precision(v, 8))); // "3150.15000000"

v = 0.123456789123456789;
to_string(v);                                        // "0.123457"
to_string(format::fixed(format::precision(v, 8))); // "0.12345679"
```

Note that all these functions can be used in other contexts than just `to_string()` and `to_string_vector()`, essentially whenever a conversion to string is performed. See for example `ascii::write_table()`.

11.1.3 from_string

```
template<typename Type>
bool from_string(const std::string& s, const Type& v); // [1]

template<std::size_t D, typename Type>
vec<D,bool> from_string(const vec<D,std::string>& v, vec<D,Type>& v); // [2]
```

The function [1] tries to convert the string `s` into a C++ value `v` and returns `true` in case of success. If the string cannot be converted into this value, for example if the string contains letters and the value has an arithmetic type, or if the number inside the string is too big to fit inside the C++ value, the function will return `false`. In this case, the value of `v` is undefined.

The version [2] will try to convert each value inside the string vector `s`, and will store the converted values inside the vector `v`. It will automatically take care or resizing the vector `v`, so you can pass an empty vector in input. The return value is an array of boolean values, corresponding to the success or failure of conversion for each individual value inside `s`. If an element of `s` failed to convert, the corresponding value in `v` will be undefined.

Example:

```
float f;
bool b = from_string("3.1415", f);
b; // true
f; // 3.1415
```

(continues on next page)

(continued from previous page)

```
b = from_string("abcdef", f);
b; // false;
f; // ??? could be 3.1415, or NaN, or anything else

vec1f fs;
vec1b bs = from_string({"1", "1.00e5", "abc", "1e128", "2.5"}, fs);
bs; // {true, true, false, false, true}
fs; // {1,    1e5,  ???,   ???,   2.5}
```

11.2 Basic string operations

Defined in header `<vif/utility/string.hpp>`.

11.2.1 empty

```
bool empty(const std::string& s); // [1]

template<std::size_t D>
vec<D,bool> empty(const vec<D,std::string>& s); // [2]
```

The function [1] will return `true` if the provided string does not contain *any* character (including spaces), and `false` otherwise. This is a synonym for `s.empty()`. The function [2] is the vectorized version of [1].

Example:

```
vec1s str = {"", "abc", "   "};
vec1b b = empty(str);
b; // {true, false, false}
// Not to be confused with the vec::empty() function
str.empty(); // false
str = {""};
str.empty(); // false
str = {};
str.empty(); // true
```

11.2.2 length

```
uint_t length(const std::string& s); // [1]

template<std::size_t D>
vec<D,uint_t> length(const vec<D,std::string>& s); // [2]
```

The function [1] will return the length of the provided string, i.e., the number of character it contains (including spaces). If the string is empty, the function will return zero. The function [2] is the vectorized version of [1].

Example:

```
vec1s str = {"", "abc", " a b"};
vec1u n = length(str);
n; // {0, 3, 4}
```

11.2.3 keep_first, keep_last

```
std::string keep_first(std::string s, uint_t n = 1); // [1]

std::string keep_last(std::string s, uint_t n = 1); // [2]

template<std::size_t D>
vec<D,std::string> keep_first(vec<D,std::string> s, uint_t n = 1); // [3]

template<std::size_t D>
vec<D,std::string> keep_last(vec<D,std::string> s, uint_t n = 1); // [4]
```

These functions will return the first ([1]) or last ([2]) `n` characters of the string `s` and discard the rest. If `n` is larger than the size of `s`, the whole string is returned untouched. Functions [3] and [4] are the vectorized versions of [1] and [2], respectively.

Example:

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.fits"};
vec1s s = keep_first(v, 2);
s; // {"p1", "p3", "p1"}
s = keep_last(v, 4);
s; // {".txt", "fits", "fits"}
```

11.2.4 distance

```
uint_t distance(const std::string& s1, const std::string& s2); // [1]

template<std::size_t D>
vec<D,uint_t> distance(const vec<D,std::string>& s1, const std::string& s2); // [2]
```

The function [1] computes the *lexicographic distance* between two strings. The definition of this distance is the following. If the two strings are exactly identical, the distance is zero. Else, each character of the shortest string are compared to the corresponding character at the same position in the other string: if they are different, the distance is increase by one. Finally, the distance is increased by the difference of size between the two strings.

The goal of this function is to identify *near* matches in case a string could not be found in a pre-defined list. This is useful to suggest corrections to the user, who may have misspelled it.

Example:

```
vec1s s = {"wircam_K", "hawki_Ks", "subaru_B"};
vec1u d = distance(s, "wirkam_Ks");
d; // {2, 8, 8}

// Nearest match
std::string m = s[min_id(d)];
m; // "wircam_K"
```

11.3 Split and combine

Defined in header `<vif/utility/string.hpp>`.

11.3.1 split, split_any_of

```
vec1s split(const std::string& s, const std::string& p); // [1]

vec1s split_if_any_of(const std::string string& s, const std::string& c); //[2]
```

Function [1] will split the string `s` into a vector of sub-strings each time the pattern `p` occurs. If no such pattern is found in `s`, the function returns a vector containing a single element which is the whole string `s`. It should be used to parse lists of values separated by a fixed pattern, like a coma (`','`).

Function [2] will split the string `s` into a vector of sub-strings each time any of the characters listed in `c` occurs. If no such character is found in `s`, the function returns a vector containing a single element which is the whole string `s`. It should be used to isolate values that are separated by a variable amount of characters, such as spaces.

Example:

```
vec1s str = split("foo,bar,bob", ",");
str; // {"foo", "bar", "bob"};

// Difference between split and split_any_of:
std::string s;

s = "  this is the   end";
str = split(s, " ");
str; // {"", "", this", "is", "the", "", "", "end"};
str = split_any_of(s, " ");
str; // {"this", "is", "the", "end"};

s = "foo, bar ,  bob";
str = split(s, ",");
str; // {"foo", " bar ", "  bob"};
str = split_any_of(s, ", ");
str; // {"foo", "bar", "bob"};

// Use case: split a line of text into words
str = split_any_of(/* ... */, " \t\n\r");
```

11.3.2 cut

```
vec1s cut(const std::string& s, uint_t n); // [1]
```

This function will split the string `s` into a vector of sub-strings (or "lines") that are exactly `n` characters long (except possibly the last line, which may be shorter). Contrary to the function `wrap()`, this function does not care about spaces and word boundaries.

Example:

```
vec1s str = cut("this is the end", 5);
str; // {"this ", "is th", "e end"};
```

11.3.3 wrap

```
vec1s wrap(const std::string& s, uint_t w, const std::string& i = "", bool e = false);
↪ // [1]
```

This function will split the string `s` into a vector of sub-strings (or "lines") that are at most `w` characters long. Contrary to the function `cut()`, this function takes care of not splitting the line in the middle of a word. When this would occur, the cut is shifted back to just before the beginning of the word, and that word is flushed to the next line. If a word is larger than `w`, then it will be left alone on its own line. Alternatively, if `e` is set to `true`, the word is truncated and the last characters are lost and replaced by an ellipsis (`"..."`) to notify that the word has been truncated. Finally, the parameter `i` can be used to add indentation: these characters are added at the beginning of each line and are taken into account when calculating the line length. In this case, the first line is *not* automatically indented, to allow using a different header. This function is useful to display multi-line messages on the terminal.

Example:

```
std::string str = "This is an example text with many words. Just "
    " for the sake of the example, we are going to write a "
    "veryyyyyyyyyyyyyyyyyyyyyyyyyy long word.";

vec1s s = wrap(str, 23);
s[0]; // "This is an example text"
s[1]; // "with many words. Just "
s[2]; // "for the sake of the"
s[3]; // "example, we are going"
s[4]; // "to write a"
s[5]; // "veryyyyyyyyyyyyyyyyyyyyyyyyy"
s[6]; // "long word."

vec1s s = wrap(str, 23, "", true);
s[0]; // "This is an example text"
s[1]; // "with many words. Just "
s[2]; // "for the sake of the"
s[3]; // "example, we are going"
s[4]; // "to write a"
s[5]; // "veryyyyyyyyyyyyyyyyy..."
s[6]; // "long word."

vec1s s = wrap(str, 23, "  ", true);
s[0]; // "This is an example text"
s[1]; // "  with many words. Just"
s[2]; // "  for the sake of the"
s[3]; // "  example, we are going"
s[4]; // "  to write a"
s[5]; // "  veryyyyyyyyyyyyyyy..."
s[6]; // "  long word."
```

11.3.4 collapse

```
template<std::size_t D>
std::string collapse(vec<D,std::string> v, const std::string& s = ""); // [1]
```

This function will concatenate together all the strings present in the vector `v` to form a single string. A separator can be provided using the argument `s`, in which case the string `s` will be inserted between each pair of strings of `v` before concatenation.

Example:

```
vec1s v = {"a", "b", "c"};
std::string s = collapse(v);
s; // "abc"
```

(continues on next page)

(continued from previous page)

```
s = collapse(v, ", ");
s; // "a, b, c"
```

11.4 Formatting

Defined in header `<vif/utility/string.hpp>`.

11.4.1 trim

```
std::string trim(std::string s, const std::string& c = " \t"); // [1]

template<std::size_t D>
vec<D,std::string> trim(vec<D,std::string> s, const std::string& c = " \t"); // [2]
```

The function [1] will look at the *beginning* and *end* of the string `s` for any of the characters that is present in `c` (order is irrelevant), and remove them. This procedure is repeated until no such character is found. The net effect of this function is that the provided string `s` is *trimmed* from any of the characters listed in `c`. This is useful for example to remove leading and trailing spaces of a string (which is what the default value of `c` does), or to removes quotes, leading zeroes, etc. The function [2] is the vectorized version of [1].

Example:

```
vec1s str = {"", "abc", " a b", " a b c  "};
vec1s t = trim(str, " "); // trim spaces
t; // {"", "abc", "a b", "a b c"}

str = {"", "(a,b)", "((a,b),c)"};
t = trim(str, "()"); // trim parentheses
t; // {"", "a,b", "a,b),c"}
```

11.4.2 to_upper, to_lower

```
std::string to_upper(std::string s); // [1]

std::string to_lower(std::string s); // [2]

template<typename T>
vec<D,std::string> to_upper(vec<D,std::string> s); // [3]

template<typename T>
vec<D,std::string> to_lower(vec<D,std::string> s); // [4]
```

These functions will transform all characters of the string to be upper case ([1]) or lower case ([2]). It has no effect on non-alphabetic characters such as numbers, punctuation, of special characters. Functions [3] and [4] are the vectorized versions of [1] and [2], respectively.

Example:

```
vec1s str = {"", "abc", "AbCdE", "No, thanks!"};
vec1s t = to_upper(str);
t; // {"", "ABC", "ABCDE", "NO, THANKS!"}
t = to_lower(str);
t; // {"", "abc", "abcde", "no, thanks!"}
```

11.4.3 align_left, align_right, align_center

```
std::string align_left(std::string s, uint_t w, char f = ' '); // [1]

std::string align_right(std::string s, uint_t w, char f = ' '); // [2]

std::string align_center(std::string s, uint_t w, char f = ' '); // [3]
```

These functions will pad the provided string with the character `f` (default to a space) so that the total width the returned string is equal to `w`. If the provided string is larger than `w`, it is returned untouched. Padding characters will be appended at the end of the string ([1]), at the beginning of the string ([2]), or equally to both ([3]), so the string will be aligned left, right, and centered, respectively.

Example:

```
std::string s = "5.0";
std::string n = align_left(s, 6);
n; // "5.0   "
n = align_right(s, 6);
n; // "   5.0"
n = align_center(s, 6);
n; // " 5.0  "

// Another padding character can be used
n = align_left(s, 6, '0');
n; // "5.0000"
```

11.5 Find/replace

Defined in header `<vif/utility/string.hpp>`.

11.5.1 find

```
uint_t find(const std::string& s, const std::string& p); // [1]

template<std::size_t D>
vec<D,uint_t> find(const vec<D,std::string>& s, const std::string& p); // [2]
```

The function [1] returns the position in the string `s` of the first occurrence of the sub-string `p`. If no such occurrence is found, the function returns `npos`. The function [2] is the vectorized version.

Example:

```
vec1s v = {"Apple", "please", "complementary", "huh?"};
vec1u p = find(v, "ple");
p; // {2, 0, 3, npos}
```

11.5.2 replace

```
std::string replace(std::string s, const std::string& p, const std::string& r); // [1]

template<std::size_t D>
vec<D,std::string> replace(vec<D,std::string> s, const std::string& p, const
↪std::string& r); // [2]
```

The function [1] will look inside the string `s` for occurrences of the pattern `p` and replace each of them with the replacement `r`. The string is unchanged if no occurrence is found. In particular, this function can also be used to remove all the occurrences of `p` simply by setting `r` equal to an empty string. The function [2] is the vectorized version.

Example:

```
vec1s str = {"I eat apples", "The apple is red"};
vec1s r = replace(str, "apple", "pear");
r; // {"I eat pears", "The pear is red"}

str = {"a:b:c", "g::p"};
r = replace(str, ":", ",");
r; // {"a,b,c", "g,,p"};
```

11.5.3 begins_with, ends_with

```
bool begins_with(const std::string& s, const std::string& p); // [1]

bool ends_with(const std::string& s, const std::string& p); // [2]

vec<D,bool> begins_with(const vec<D,std::string>& s, const std::string& p); // [3]

vec<D,bool> ends_with(const vec<D,std::string>& s, const std::string& p); // [4]
```

These functions will return `true` if the beginning ([1]) or the end ([2]) of the string `s` exactly matches the string `p`. The functions [3] and [4] are the vectorized versions.

Example:

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.fits"};
vec1b b = begins_with(v, "p1");
b; // {true, false, true}

b = ends_with(v, ".fits");
b; // {false, true, true}
```

11.5.4 erase_begin, erase_end

```
std::string erase_begin(std::string s, uint_t n); // [1]

std::string erase_begin(std::string s, const std::string& p); // [2]

std::string erase_end(std::string s, uint_t n); // [3]
```

(continues on next page)

(continued from previous page)

```
std::string erase_end(std::string s, const std::string& p); // [4]

vec<D,std::string> erase_begin(vec<D,std::string> s, uint_t n); // [5]

vec<D,std::string> erase_begin(vec<D,std::string> s, const std::string& p); // [6]

vec<D,std::string> erase_end(vec<D,std::string> s, uint_t n); // [7]

vec<D,std::string> erase_end(vec<D,std::string> s, const std::string& p); // [8]
```

These functions will erase characters from the beginning ([1] and [2]) or the end ([3] and [4]) of the string `s`.

Functions [1] and [3] will remove `n` characters. If `n` is larger than the size of `s`, the returned string will be empty. Functions [2] and [4] first check that the string begins or ends with the other string `p` provided as second argument: if it does, it removes this substring from `s`; if it does not, an error is reported and the program stops.

Functions [5] to [8] are the vectorized versions of functions [1] to [4], respectively.

Example:

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.fits"};
std::string s = erase_begin(v[0], "p1_");
s; // "m2.txt"
s = erase_begin(v[1], "p1_");
// will trigger an error
s = erase_begin(v[2], "p1_");
s; // "t8.fits"

s = erase_end(v[0], ".fits");
// will trigger an error
s = erase_end(v[1], ".fits");
s; // "p3_c4"
s = erase_end(v[2], ".fits");
s; // "p1_t8"

vec1s t = erase_begin(v, 3);
t; // {"m2.txt", "c4.fits", "t8.fits"}
t = erase_end(v, 5);
t; // {"p1_m", "p3_c4", "p1_t8"}
```

11.5.5 replace_block, replace_blocks

```
template<typename T>
std::string replace_block(std::string v, const std::string& b, const std::string& e,␣
↪T f); // [1]

template<typename T>
std::string replace_blocks(std::string v, const std::string& b, const std::string& s,␣
↪const std::string& e, T f); // [2]
```

Function [1] looks in the string `v` and identifies all "blocks" that start with `b` and end with `e`. The content of each block is fed to the user-supplied function `f` which does any kind of conversion or operation on that content, and must returns a new string as a replacement. This new string is then inserted in `v` and replaces the entire block.

Function [2] does the same thing, except that each block can have multiple "components" that are separed with the separator `s`. In this case, the function extracts all these "components" and stores them inside a string vector, and feeds this vector to the conversion function `f`.

See `regex_replace` for a more powerful (but also more complex and possibly slower) alternative.

Example:

```
// We want to modify the content inside <b>...</b> to be upper case
std::string s = "This is a <b>whole</b> lot of <b>money</b>";
std::string ns = replace_block(s, "<b>", "</b>", [](std::string t) {
    return to_upper(t);
});

ns; // "This is a WHOLE lot of MONEY"

// We want to convert this LaTeX link into HTML
s = "Look at \url{http://www.google.com}{this} link.";
ns = replace_blocks(s, "\url{", "}{", "}", [](vec1s t) {
    return "<a href=\""+t[0]+"\">"+t[1]+"</a>";
});

ns; // "Look at <a href="http://www.google.com">this</a> link."
```

11.6 Regular expressions (regex)

Defined in header `<vif/utility/string.hpp>`.

11.6.1 regex_match

```
bool regex_match(const std::string& s, const std::string& r); // [1]

template<std::size_t D>
bool regex_match(const vec<D,std::string>& s, const std::string& r); // [2]
```

Function [1] will return `true` if the string `s` matches the regular expression (or "regex" for short) `r`. This regular expression can be used to identify complex patterns in a string, far more advanced than just matching the existence of a sub-string. This particular implementation uses POSIX regular expressions. The syntax is complex and not particularly intuitive, but it has become a well known standard and is therefore understood by most programmers. A gentle tutorial can be found here. If the regular expression is invalid, an error will be reported to diagnose the problem, and the program will stop. Function [2] is the vectorized version.

Note: For regular expression we advise to use C++ *raw string literals*. Indeed, in these expressions one often needs to use the backslash character (\), but this character is also used in C++ to form special characters, such as `'\n'` (new line). For this reason, to feed the backslash to the regex compiler one actually needs to escape it: `"\\"`. But the POSIX rules also use \ as an escape character, and so this can quickly cause head aches… (to have a regex matching the backslash character itself, one would have to write `"\\\\"`!) To avoid this inconvenience, one can enclose the regular expression in `R"(...)"`, and not need to worry about escaping characters from the C++ compiler.

Example:

```
vec1s v = {"abc,def", "abc:def", "956,fgt", "9g5,hij", "ghji,abc"};

// We want to find which strings have the "XXX,YYY" format
```

(continues on next page)

(continued from previous page)

```
// where "XXX" can be any combination of three letters or numbers
// and "YYY" can be a combination of three letters
vec1b b = regex_match(v, R"(^[a-z0-9]{3},[a-z]{3}$)");

// This regular expression can be read like:
//  '^'   : at the beginning of the string, match...
//  '['   : any character among...
//  'a-z' : letters from 'a' to 'z'
//  '0-9' : numbers from 0 to 9
//  ']'
//  '{3}' : three times
//  ','   : followed by a coma, then...
//  '['   : any character among...
//  'a-z' : letters from 'a' to 'z'
//  ']'
//  '{3}' : three times
//  '$'   : and then the end of the string

b[0]; // true
b[1]; // false, there is no ","
b[2]; // true
b[3]; // true
b[4]; // false, too many characters before the ","
```

11.6.2 regex_extract

```
vec2s regex_extract(const std::string& s, const std::string& r); // [1]
```

This function will analyze the string `s`, perform regular expression matching (see `regex_match` above) using the regular expression `r`, and will return a vector containing all the extracted substrings. To extract one or more substrings in the regular expression, just enclose the associated patterns in parentheses. The returned vector is two dimensional: the first dimension corresponds to the number of times the regular exception was matched in the provided string, the second dimension corresponds to each extracted substring.

Example:

```
std::string s = "array{5,6.25,7,28}end45.6ddfk 3.1415";

// We want to find all floating point numbers in this mess, and
// extract their integer and fractional parts separately.
vec2s sub = regex_extract(s, R"(([0-9]+)\.([0-9]+))");

// The regular expression can be read like:
// '('   : open a new sub-expression containing...
// '['   : any character among...
// '0-9' : the numbers 0 to 9
// ']'
// '+'   : with at least one such character
// ')'   : end of the sub-expression
// '\.'  : followed by a dot (has to be escaped with '\')
// '('   : open a new sub-expression containing...
//          ... exactly the same pattern as the first one
// ')'   : end of the sub-expression

// So we are looking for two sub-expressions, the first is the
```

(continues on next page)

(continued from previous page)

```
// integral part of the floating point number, and the second is the
// fractional part.

// It turns out that there are three locations in the input string
// that match this pattern:
sub(0,_); // {"6",  "25"}
sub(1,_); // {"45", "6"}
sub(2,_); // {"3",  "1415"}
```

11.6.3 regex_replace

```
template<typename T>
std::string regex_replace(std::string s, const std::string& reg, T fun); // [1]
```

This function will search in the string `s` using the regular expression `reg` (see `regex_match` above) to locate some expressions. Parts of these expressions can be *captured* by enclosing them in parenthesis. These captured sub-expressions are extracted from `s`, stored inside a string vector, and fed to the user-supplied "replacement function" `fun`. In turn, this function analyzes and/or modifies the captured sub-expressions to produce a new replacement string that will be inserted in place of the matched expression. The function is call for each match of the regular expression in `s`.

If no sub-expression is captured, then the string vector that is fed to `fun` will be empty. Expressions are found and replaced in the order in which they appear in the input string `s`.

This function is very similar to the `sed` program.

Example:

```
std::string s = "a, b, c=5, d=9, e, f=2, g";

// First a simple example.
// We want to find all the "X=Y" expressions in this string and
// add parentheses around "Y".
std::string n = regex_replace(
    s,                      // the string to analyze
    R"(([a-z])=([0-9]))", // the regular expression
    // the replacement function
    [](vec1s m) {
        // "X" is in m[0], "Y" is in m[1].
        return m[0]+"=("+m[1]+")";
    }
);

// The regular expression can be read like:
// '('   : open a new sub-expression containing...
// '['   : any character among...
// 'a-z' : the letters 'a' to 'z'
// ']'
// ')'   : end of the sub-expression
// '='   : followed by the equal sign
// '('   : open a new sub-expression containing...
// '['   : any character among...
// '0-9' : the numbers 0 to 9
// ']'
// ')'   : end of the sub-expression
```

(continues on next page)

(continued from previous page)

```
// The result is:
n; // "a, b, c=(5), d=(9), e, f=(2), g"

// A second, more complex example.
// We take the same example as above, but this time we want to
// change "X" to upper case and increment "Y" by one.
std::string n = regex_replace(
    s,                      // the string to analyze
    R"(([a-z])=([0-9]))", // the regular expression
    // the replacement function
    [](vec1s m) {
        // Again, "X" is in m[0], "Y" is in m[1].

        // Read the integer "Y" and increment it
        uint_t k;
        from_string(m[1], k);
        ++k;

        // Build the replacement string
        return to_upper(m[0])+"="+to_string(k);
    }
);

// The result is:
n; // "a, b, C=6, D=10, e, F=3, g"
```

11.7 Hash

Defined in header `<vif/utility/string.hpp>`.

11.7.1 hash

```
template<typename ... Args>
std::string hash(const Args& ... args); // [1]
```

This function scans all the arguments that are provided, and returns the hexadecimal representation of the SHA-1 "hash" of this argument list. The hash is a string such that: 1) all further calls of `hash(...)` with arguments that have the exact same value (perhaps when the program is executed a second time later) will always return the same string, and 2) the probability is very small that the function returns the same string for another set of arguments, or arguments with different values. Although this algorithm was created in 1995, the first "collision" (two different data sets producing the same hash) was found in 2017.

This is useful for example to cache the result of some heavy computation: once the computation is done, the *input* parameters of the computation can be fed to `hash()` to give a "sort-of-unique" identifier to the "input+result" pair. The result of the computation can then be saved somewhere with the hash as an identifier. Later on, if the computation is requested with a new set of parameters, these parameters are fed to `hash()` and the resulting string is compared to all the identifiers of the cached results: if a match is found, then the associated pre-computed result can be re-used, else the computation must be executed anew.

Example:

```
std::string s;

// With a single argument
s = hash("hello world!");
s; // "da52a1357f3c973e1ffc1b694d5308d0abcd9845"
s = hash("hello world?")
s; // "793e673d04e555f8f0b38033d5223c525a040719"
// Notice how changing a single character gives a completely
// different hash string

// With multiple arguments
s = hash(1, 2, 3);
s; // "570331ab965721aae8a8b3c628cae57a21a37560"
s = hash("123");
s; // "0e898437b29ec20c39ca48243e676bcb177d4632"
s = hash(1.0, 2.0, 3.0);
s; // "9c45014f7c7943cb7860f3db4b885fb44b510ec8"
// Notice how the hash is different even though we would
// consider these different sets of values to be equivalent.
```

CHAPTER 12

Printing to the terminal

Defined in header <vif/core/print.hpp>.

12.1 print

```
template<typename ... Args>
void print(Args&& ...); // [1]
```

This function will "print" (or display) the content of all its arguments into the standard output (i.e., the terminal). This has *nothing* to do with printing things on paper (unless you have set *that* as your standard output). Arguments are printed one after the other on the same line, without any spacing or separator, and the "end of line" character is printer after the last argument so that the next call of `print()` will display on a new line. This function should only be used for debugging purposes, or to inform the user of the progress of the program (but see also below and `print_progress()`).

Example:

```
uint_t a = 5;
vec1u x = {1,2,3,4,9,10};

print("the program reached this location, a=", a, ", and x=", x);
```

The above code will display on the terminal:

```
the program reached this location, a=5, and x={1,2,3,4,9,10}
```

A few words about formatting.

First, since the text is meant to be sent to a simple terminal window, there are very few options to affect the look and feel of the output. In fact there is currently no option to change fonts, add boldface or italic, or change the color (some of this may be implemented in the future, but it will always remain fairly limited). The only thing you can actually control is when to start a new line. This is done with the special character `'\n'`, and such a line break is always made at the end of each printed message. If you need a finer control of line jumps, you will have to come back to the standard C++ functions for output (i.e., `std::cout` and the likes).

Second, if you are used to C functions like `prinft()`, note from the above example how printing in vif behaves quite differently. The `print()` function does not work with “format strings”, hence the character `'%'` is not special and can be used safely without escaping. One can print a value that is not a string simply by adding it to the list of the function arguments, which will do the conversion to string automatically based on that argument’s type. See also *String conversions* for more information on that process. The format of this conversion is defined by the C++ standard library, and is usually fine for most cases. If you need a finer control on the display format (for example for floating point numbers), look at *Formatting* and `format::` functions.

12.2 error, warning, note

```
template<typename ... Args>
void error(...); // [1]

template<typename ... Args>
void warning(...); // [2]

template<typename ... Args>
void note(...); // [3]
```

These functions behave exactly like `print()`. The only difference is that they automatically append a “prefix” to the message, namely: `"error:  "` ([1]), `"warning:  "` ([2]), or `"note:  "` ([3]). In some operating systems, these prefixes can be colored to make them stand out better from the regular `print()` output. These three functions are indeed much more useful than `print()`, since they are meant to talk to the *user* of a program rather than the *developer*. They can be used, for example, to tell the user that something went wrong ([1]), or could go wrong ([2]), or simply to inform them of what’s going on ([3]).

More specifically, function [1] (`error()`) should be used to print *unrecoverable* and *serious* errors. In other words, this is a situation the program was not designed to handle, and it has to stop and tell the user why. Function [2] (`warning()`) is for *recoverable* errors (unusual situations for which a workaround is implemented), or situations where the user is *likely* to have made a mistake (but maybe not). Lastly, function [3] (`note()`) is for everything else which is not critical and does not require immediate attention. Typically this will be logging, displaying the progress of a calculation, etc. Since these types of messages are not critical, the user should be able to quickly glance over them and ignore them. Is it therefore good practice to offer an option to completely disable this non-critical output, so only errors and warnings are displayed.

Example:

```
bool verbose = false; // let the user enable extra output only if needed

std::string datafile1 = "toto1.txt";
if (!file::exist(datafile1)) {
    // We cannot work without 'datafile1', print an error
    error("cannot open '", datafile1, "'");
    error("make sure the program is run inside the data directory");
    return 1; // typically, exit the program
}

std::string datafile2 = "toto2.txt";
if (!file::exist(datafile2)) {
    // It's better is we have 'datafile2', but we can work without it.
    // So we print a warning and let the user know what are the
    // consequences of this non-critical issue.
```

(continues on next page)

(continued from previous page)

```
    warning("cannot open '", datafile2, "', so calculation will be less accurate");
    warning("will use '", datafile1, "' as a fallback");
    datafile2 = datafile1;
}

if (verbose) {
    // Things are going fine, inform the user of what we are about to do
    note("analysing the data, please wait...");
}

// Do the stuff...
```

The above code, if the first file does not exist, will display:

```
error: cannot open 'toto1.txt'
error: make sure the program is run inside the data directory
```

12.3 prompt

```
template<typename T>
bool prompt(const std::string& msg, T& v, const std::string& err = ""); // [1]
```

This function interacts with the user of the program through the standard output and input (i.e., the terminal). It first prints `msg`, waits for the user to enter a value and press the Enter key, then try to load this value inside `v`. If the value entered by the user is invalid and cannot be converted into the type `T`, the program asks again and optionally writes an error message `err` to clarify the situation.

Currently, the function can only return after successfully reading a value, and always returns `true`. In the future, it may fail and return `false`, for example after the user has failed a given number of times. If possible, try to keep the possibility of failure into account.

Example:

```
uint_t age;
if (prompt("please enter your age: ", age,
    "it better just be an integral number...")) {
    print("your age is: ", age);
} else {
    print("you will do better next time");
}
```

Here is a possible interaction scenario with a (naive) user:

```
please enter your age: 15.5
error: it better just be an integral number...
please enter your age: what?
error: it better just be an integral number...
please enter your age: oh I see, it is 15
error: it better just be an integral number...
please enter your age: ok...
error: it better just be an integral number...
please enter your age: 15
your age is: 15
```

CHAPTER 13

Interacting with the operating system

13.1 Environment variables

Defined in header <vif/utility/os.hpp>.

13.1.1 system_var

```
std::string system_var(const std::string& v, const std::string& d); // [1]

template<typename T>
T system_var(string v, T d); // [2]
```

These functions looks inside the operating system environment for a variable named v (using the C function getenv()). If this variable exists, its value is returned as a string ([1]), or converted into a value of type T ([2]). If the conversion fails, or if the environment variable does not exist, the default value d is returned instead.

Environment variables are complementary to *Command line arguments*. They are mostly used to store constant, system-specific configurations that will typically change only between one machine or one user to another. Because they seldom change, it would be tedious to have to specify these configurations as command line arguments and provide them for *each* call of a given program. Instead, environment variables are set "once and for all" at the beginning of the user's session (on Linux this is usually done in the ~/.bashrc file, or equivalent), and are read on demand by each program that needs them.

By convention and for portability issues, it is recommended to specify environment variable names in full upper case (i.e., "PATH" and not "Path" or "path").

Example:

```
// One typical use case is to get the path of some external component
std::string sed_library_dir = system_var("SUPERFIT_LIB_PATH");
if (!sed_library_dir.empty()) {
    // The directory has been provided, look what is inside...
} else {
```

(continues on next page)

(continued from previous page)

```
    // This component is missing, try to do without or print an error
}

// It can also be used to modify generic behaviors, for example
// configure how many threads we want to use by default in all the
// programs of a given tool suite.
uint_t nthread = system_var<uint_t>("MYTOOLS_NTHREADS", 1);
```

13.2 Processes

Defined in header `<vif/utility/os.hpp>`.

13.2.1 spawn

```
bool spawn(const std::string& cmd);
```

This function executes the shell command passed in argument (on UNIX systems, using `/bin/sh`). It puts execution of the current program on hold until the shell command terminates. This can be used to launch another application, for example an image viewer that the user can use to inspect some temporary calculations.

If you wish to run this command in parallel with the current program, see `fork()`.

Example:

```
// Calculate some things to make an image
vec2d img = /* ... */;

// Write this image to the disk in FITS format
fits::write("tmp.fits", img);

// Call the FITS viewer DS9 on this data and let user inspect it.
// The program will be paused until the user closes DS9.
spawn("ds9 tmp.fits");

// Now execution comes back to us.
// We can, for example, ask the user if the data was satisfactory.
```

13.2.2 fork

```
bool fork(const std::string& cmd);
```

This function creates a new `child` process to execute the shell command passed in argument (on UNIX systems, using `/bin/sh`). The child process runs in the background, while execution continues in the main process. Note that any child process will be automatically killed when the main process terminates: child processes cannot survive longer than the main process.

If you wish simply to run a command and wait for it to finish, see `spawn()`.

Example:

```
// Calculate some things to make an image
vec2d img = /* ... */;

// Write this image to the disk in FITS format
fits::write("tmp.fits", img);

// Call the FITS viewer DS9 on this data to let user inspect it.
// The DS9 window will open in the background.
fork("ds9 tmp.fits");

// Execution comes back to us immediately, so the user can keep
// inspecting the image while we do some more calculations.
// Just be careful not to open too many windows in this way!
```

CHAPTER 14

File system

14.1 File paths manipulation

Defined in header `<vif/io/filesystem.hpp>`.

14.1.1 file::directorize

```
std::string file::directorize(const std::string& p); // [1]

template<std::size_t D>
vec<D,std::string> file::directorize(const vec<D,std::string>& p); // [2]
```

The function [1] modifies the path given in argument to make sure that a file name can be appended to it and form a valid file path. In UNIX systems, for example, the function ensures that the path ends with a forward slash `/`.

The function [2] is the vectorized version of [1].

Example:

```
std::string p;
p = file::directorize("/some/path");     // "/some/path/"
p = file::directorize("/another/path/"); // "/another/path/"
```

14.1.2 file::is_absolute_path

```
std::string file::is_absolute_path(const std::string& p); // [1]

template<std::size_t D>
vec<,D,std::string> file::is_absolute_path(const vec<,D,std::string>& p); // [2]
```

The function [1] returns `true` if its argument describes an absolute file path, and `false` otherwise. In UNIX systems, for example, this is equivalent to checking that the path starts with a forward slash `/` (referencing the root directory).

The function [2] is the vectorized version of [1].

Example:

```
bool b = file::is_absolute_path("/some/path");           // true
b = file::is_absolute_path("../sub/directory/file.txt"); // false
```

14.1.3 file::get_basename

```
std::string file::get_basename(const std::string& p); // [1]

template<std::size_t D>
vec<,D,std::string> file::get_basename(const vec<,D,std::string>& p); // [2]
```

The function [1] extracts the name of a file from its full path given in argument. If this path is that of a directory, the function returns the name of this directory. This behavior is similar to the bahs function `basename`.

The function [2] is the vectorized version of [1].

Example:

```
std::string n = file::get_basename("/some/path");       // "path"
n = file::get_basename("/another/path/to/a/file.txt"); // "file.txt"
```

14.1.4 file::get_extension

```
std::string file::get_extension(const std::string& f); // [1]

template<std::size_t D>
vec<,D,std::string> file::get_extension(const vec<,D,std::string>& f); // [2]
```

The function [1] scans the provided string to look for a file extension. The "extension" is whatever is found at the end the string after the *last* dot (and including this dot), for example `".cpp"`. If an extension is found, this function returns it (including the leading dot), else it returns an empty string.

The function [2] is the vectorized version of [1].

Example:

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.dat.fits", "readme"};
vec1s s = file::get_extension(v); // {".txt", ".fits", ".fits", ""}
```

14.1.5 file::remove_extension

```
std::string file::remove_extension(const std::string& f); // [1]

template<std::size_t D>
vec<,D,std::string> file::remove_extension(const vec<,D,std::string>& f); // [2]
```

The function [1] scans the provided string to look for a file extension. The "extension" is whatever is found at the end the string after the *last* dot (and including this dot), for example `".cpp"`. If an extension is found, this function returns the input string with this extension removed. If no extension is found, the input string returned unchanged.

The function [2] is the vectorized version of [1].

Example:

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.dat.fits", "readme"};
vec1s s = file::remove_extension(v); // {"p1_m2", "p3_c4", "p1_t8.dat", "readme"}
```

14.1.6 file::split_extension

```
std::pair<std::string> file::split_extension(const std::string& f); // [1]

template<std::size_t D>
vec<D,std::pair<std::string>> file::split_extension(const vec<,D,std::string>& f); //␣
↪[2]
```

The function [1] scans the provided string to look for a file extension. The "extension" is whatever is found at the end the string after the *last* dot (and including this dot), for example `".cpp"`. If an extension is found, this function splits the input string into two substrings, the first being the string with the extension removed (see `file::remove_extension()`), and the second being the extension itself (see `file::get_extension()`).

The function [2] is the vectorized version of [1].

Example:

```
auto p = file::split_extension("p1_m2.txt");
p.first; // "p1_m2"
p.second; // ".txt"
```

14.1.7 file::get_directory

```
string file::get_directory(const std::string& p); // [1]

template<std::size_t D>
vec<,D,std::string> file::get_directory(const vec<,D,std::string>& p); // [2]
```

The function [1] scans the path given in argument and returns the path to the parent directory. This behavior is similar to the bash function `dirname`, except that here the returned path always ends with a forward slash `/`.

The function [2] is the vectorized version of [1].

Example:

```
std::string n;
n = file::get_directory("/some/path");                // "/some/"
n = file::get_directory("/another/path/to/a/file.txt"); // "/another/path/to/a/"
```

14.2 File system

Defined in header `<vif/io/filesystem.hpp>`.

14.2.1 file::exists

```
bool file::exists(const std::string& f); // [1]

template<std::size_t D>
vec<D,bool> file::exists(const vec<D,std::string>& f); // [2]
```

The function [1] returns `true` if a file (or directory) exists at the location given in `f`, and `false` otherwise.

The function [2] is the vectorized version of [1].

Example:

```
bool b;
b = file::exists("~/.vifrc");       // hopefully true
b = file::exists("/i/do/not/exist"); // probably false
```

14.2.2 file::is_older

```
bool file::is_older(const std::string& f1, const std::string& f2); // [1]

template<std::size_t D>
vec<D,bool> file::is_older(const vec<D,std::string>& f1, const std::string& f2); //␣
↪[2]
```

The function [1] returns `true` if the file (or directory) `f1` is *older* than the file (or directory) `f2`. The “age” of a file corresponds to the time spent since that file was last modified. If one of the two files does not exists, the function returns `false`.

The function [2] is the vectorized version of [1], where all files in `f1` are compared against the same file `f2`.

Example:

```
bool b = file::is_older("~/.vifrc", "/usr/bin/cp"); // maybe false?
```

14.2.3 file::list_directories

```
vec1s file::list_directories(const std::string& d, const std::string& p = "");
```

This function returns the list of all the subdirectories inside the directory `d` that match the search pattern `p`. Only the names of the directories are returned, not their full paths. When the search pattern is empty (default), all the directories are returned. Otherwise, the pattern can contain any number of “wildcard” character `*` to filter the output, as when listing files in bash (see examples below).

Hidden directories are ignored. An empty list is returned if there is no subdirectory matching the pattern, or if the operation could not be performed (that is, if `d` doest not exist, or is not a directory, or if you do not have read access to it). The function does not look inside subdirectories recursively. The *order* of the directories in the output list is undefined and should not be relied upon: if you need a sorted list, you have to sort it yourself.

Example:

```
vec1s d = file::list_directories("./");
d; // subdirectories of the working directory

```

(continues on next page)

(continued from previous page)

```
d = file::list_directories("/path/to/vif/");
d; // {"cmake", "test", "doc", "bin", "include", "tools"}

d = file::list_directories("/path/to/vif/", "t*");
d; // {"test", "tools"}
```

14.2.4 file::list_files

```
vec1s file::list_files(const std::string& d, const std::string& p = "");
```

This function returns the list of all the files inside the directory `d` that match the search pattern `p`. Only the names of the files are returned, not their full paths. When the search pattern is empty (default), all the files are returned. Otherwise, the pattern can contain any number of "wildcard" character `*` to filter the output, as when listing files in bash (see examples below).

Hidden files are ignored. An empty list is returned if there is no file matching the pattern, or if the operation could not be performed (that is, if `d` doest not exist, or is not a directory, or if you do not have read access to it). The function does not look inside subdirectories recursively. The *order* of the files in the output list is undefined and should not be relied upon: if you need a sorted list, you have to sort it yourself.

Example:

```
vec1s d = file::list_files("./");
d; // files in the working directory

d = file::list_files("/path/to/vif/doc");
d; // {"vif.pdf", "compile.sh", "vif.tex"}

d = file::list_files("/path/to/vif/doc", "*.tex");
d; // {"vif.tex"}
```

14.2.5 file::explorer

```
class file::explorer {
public:
    struct file_data {
        std::string full_path;
        std::string name;
        uint_t size;
        bool is_hidden = false;
        bool is_dir = false;
    };

    // Constructors
    explorer(); // [1]
    explorer(const std::string& d, const std::string& p = ""); // [2]

    void open(const std::string& d, const std::string& p = ""); // [3]
    bool find_next(file_data& f); // [4]
    void close(); // [5]
};
```

This class allows you to browse through the content of a directory, to list the files and other directories it contains. Its interface is similar to `std::ifstream`: it can be default-constructed ([1]) then initialized with `open()` ([3]), or this can be achieved in a single step using the constructor [2], which takes the same arguments as `open()`.

To use this class, you must first open a directory, either with [2] or [3]: the class will attempt to open the directory `d` and initialize a new search, optionally with a search pattern `p`. If the directory does not exist or is not readable, `open()` will return `false`, and the search will be aborted (subsequent calls to `find_next()` will return `false`). The search pattern must constain at least one wildcard character `*` to indicate which part of the files (or directories) name is allowed to vary, like when listing files in bash.

Once the directory is open, you can iterate over its content using `find_next()` ([4]). This function take a pre-constructed `file_data` in argument, in which it will fill the details of the next file it found. If no more file is found (i.e., if the previous call to `find_next()` returned the last file), this function returns `false` and the `file_data` is not modified (should not be used).

The `file_data` object is a simple structure holding basic informations about the file (or directory): `name` is the name the file, `full_path` is the name appended to the search directory `d`, `size` is the size of the file (in bytes), `is_hidden` is `true` for hidden files or directories, and `is_dir` is `true` for directories and `false` for files.

Once you are done with a search, you can let the `explorer` instance be destroyed at the end of its scope. This will call `close()` ([5]) automatically. If you need to close the access to the directory immediately, or if you wish to start another search, you can also call `close()` explicitly.

Example:

```
// Create explorer
file::explorer e;

// Try to open directory
if (e.open("some/dir", "*.cpp")) {
    // Success, now list the files/directories
    file::explorer::file_data f;
    while (e.find_next(f)) {
        // We found a file/directory, do something with it:
        print("found ",
            (f.is_hidden ? "hidden " : ""),
            (f.is_dir ?    "directory " : "file "),
            f.name, " (size: ", f.size, ")");
    }
} else {
    // Failed
    error("could not open directory some/dir");
}
```

14.2.6 file::mkdir

```
bool file::mkdir(const std::string& d); // [1]

template<std::size_t D>
vec<D,bool> file::mkdir(const vec<D,std::string>& d); // [2]
```

The function [1] creates a new directory at the path given in argument (including all the parent directories, if necessary), and returns `true`. If the directory could not be created (e.g., because of permission issues), the function returns `false`. If the directory already exists, the function does nothing and returns `true`. This function is equivalent to the bash function `mkdir -p`.

The function [2] is the vectorized version of [1].

Example:

```
bool b = file::mkdir("/path/to/vif/a/new/directory");
// Will most likely create the directories:
//  - /path/to/vif/a
//  - /path/to/vif/a/new
//  - /path/to/vif/a/new/directory
b; // maybe true or false, depending on your permissions
```

14.2.7 file::copy

```
bool file::copy(const std::string& from, const std::string& to);
```

This function creates a copy of the file `from` at the location given in `to` and returns `true`. If the new file could not be created (e.g., because of permission issues or because its parent directory does not exist), or if the file `from` could not be found or read, the function returns `false`. If the file `to` already exists, it will be overwritten without warning. Copying directories is not presently supported. This function is equivalent to the bash function `cp -f`.

Example:

```
bool b = file::copy("/home/joe/.vifrc", "/home/bob/.vifrc");
b; // maybe true or false, depending on your permissions
```

14.2.8 file::remove

```
bool file::remove(const std::string& f);
```

This function will delete the file (or directory) given in argument and return `true` on success, or if the file did not exist. It will return `false` only if the file exists but could not be removed (i.e., because you are lacking the right permissions). This function is equivalent to the bash function `rm -rf`.

Example:

```
// That's a bad idea, but for the sake of the example...
bool b = file::remove("/home/joe/.vifrc");
b; // probably true
```

14.2.9 file::to_string

```
std::string file::to_string(const std::string& f);
```

This function reads the content of the file whose path is given in argument, stores all the characters (including line break characters and spaces) inside a string and returns it. If the file does not exist, the function returns an empty string.

Warning: This is a very sub-optimal way of reading the content of a file, and it should only be attempted on short files. If you need to read a file line by line, use `std::getline()` instead. If you need to read a data table, use the dedicated functions in *ASCII tables*.

Example:

```
std::string r = file::to_string("/etc/lsb-release");
// 'r' now contains all the lines of the file, each
// separated by a newline character '\n'.
```

CHAPTER 15

ASCII tables

Defined in header `<vif/io/ascii.hpp>`.

15.1 ascii::read_table

```
template<typename ... Args>
void ascii::read_table(std::string f, Args&& ... args); // [1]

template<typename ... Args>
void ascii::read_table(std::string f, const ascii::input_format& o, Args&& ... args);␣
↪// [2]
```

These functions read the content of the ASCII table whose path is given in `f`, and stores the data inside the vectors listed in `args`. Each column of the file will be stored in a separate vector, in the order in which they are provided to the function. If there is more columns than vectors, the extra columns are ignored. If there is more vectors than columns, the program will stop and report an error.

Function [1] assumes a number of default options regarding the layout of the table. In particular, it assumes the columns are separated by white spaces, and that there may be a header at the beginning of the table (lines starting with the character `'#'`) that must be skipped before reading the data. See below for more detail on how the table data is read. Below is an example of such a table.

Example:

```
# my_table.dat
# id    x     y
   0   10    20
   5   -1   3.5
   6    0    20
   8    5     1
  22  6.5    -5
```

We can read this in C++ with the following code:

```
// Declare the vectors that will receive the data
vec1u id;
vec1f x, y;

// Read the data, asking for three columns.
ascii::read_table("my_table.dat", id, x, y);

// Use the vectors
print(id); // {0, 5, 6, 8, 22}
```

Warning: Beware that, with these functions, the *names* of the C++ vectors are not used to identify the data in the file, and the information contained in the table header is plainly ignored. Only the *order* of the columns and vectors matters.

Function [2] allows you to fine tune how the table data is read using the option structure `o`. This structure has the following members:

```
struct input_format {
    bool        auto_skip    = true;
    std::string skip_pattern = "#";
    uint_t      skip_first   = 0;
    std::string delim        = " \t";
    bool        delim_single = false;
};
```

- `auto_skip` and `skip_pattern`. When `auto_skip` is set to `true`, the function will automatically ignore all the lines starting with `skip_pattern` (typically, the header).
- `skip_first`. This is an alternative way to skip a header, when the header has always the same number of lines (one or two, typically), but when the lines do not start with a specific character. By setting this option to a positive number, the function will skip the first `skip_first` lines before reading the data.
- `delim` and `delim_single`. The string `delim` determines what characters are used to separate the columns in the file. When `delim_single` is `false`, `delim` is interpreted as a list of characters that can be expected in between columns, in any number and order. For example, `delim = " \t"; delim_single = false;` states that columns can be separated by any number of white spaces and tabulations. On the other hand, when `delim_single` is `true`, `delim` is interpreted as a fixed string that must be found between each column, and any other character is considered part of the column data itself. For example, `delim = ","; delim_single = true;` would specify a comma-separated table.

Some pre-defined sets of options are made available for simplicity:

- `ascii::input_format::standard()`. This is the default behavior of `read_table()`, when no options are given (see default values above). With this setup, columns in the file can be separated by any number of spaces and tabulations. The data does not need to be perfectly aligned to be read correctly, even though it is recommended for better human readability. The table may also contain empty lines, they will simply be ignored.

  However, "holes" in the table are not supported (i.e., rows that only have data for some columns, but not all). For example:

  ```
  # my_table.dat
  # id    x     y
     0   10    20
     5   -1   3.5
     6         20  # <-- not OK!
  ```

(continues on next page)

(continued from previous page)

```
  8    5     1
 22  6.5    -5
```

In such cases (see how `x` is missing a value on the third row), the "hole" would be considered as whitespace and ignored, and the data from this column would actually be read from the next one (so `x` would have a value of `20` for this row). This would eventually trigger an error when trying to read the last columns, because there won't be enough data on this line (there is no value for `y`). Therefore, *every* row must have a value for *every* column. If data is missing, use special values such as `-1` or `NaN` to indicate it.

This also means that string columns cannot contain spaces in them, since they would otherwise be understood as column separators. Adding quotes `"..."` will *not* change that. If you need to read strings containing spaces, you should use another table format (such as CSV, see below).

Using this format is done as follows:

```
// Declare the vectors that will receive the data
vec1u id;
vec1f x, y;

// Read the data, asking for three columns.
ascii::read_table("my_table.dat", ascii::input_format::standard(), id, x, y);
// This is equivalent to
ascii::read_table("my_table.dat", id, x, y);
```

You can also use it as a starting point to create customized options:

```
ascii::input_format opts = ascii::input_format::standard();
opts.skip_pattern = "%"; // skip lines starting with '%'
ascii::read_table("my_table.dat", opts, id, x, y);
```

- `ascii::input_format::csv()`. This preset enables loading comma-separated values (CSV) tables. In these tables, columns are separated by a single comma (`','`). Contrary to the `standard` format, spaces are considered to be a significant part of the data, and will not be trimmed.

The information below applies to any type of table.

Data type. Values in ASCII tables are not explicitly typed, so a column containing integers can be read as a vector of integers, floats, or even strings. As long as the data in the table can be converted to a value of the corresponding C++ vector using `from_string()` (see *String conversions*), this function will be able to read it. Note that, for all numeric columns, if the value to be read is too large to fit in the corresponding C++ variable, the program will stop and report an error. This will happen for example when trying to read a number like `1e128` inside a `float` vector. In such cases, use a larger data type to fix this (e.g., `double` in this particular case).

Skipping columns. If you want to ignore a specific column, you can use the "placeholder" symbol `_` instead of providing an actual vector. The corresponding data in the table will not be read. If you want to ignore `n` columns, you can use `ascii::columns(n,_)`. With the example table above:

```
// Declare the vectors that will receive the data
vec1u id;
vec1f x, y;

// Read the data, ignoring the 'x' column
ascii::read_table("my_table.dat", 2, id, _, y);

// Read the data, ignoring the 'id' and 'x' columns.
// This can be done with two '_':
ascii::read_table("my_table.dat", 2, _, _, y);
```

(continues on next page)

(continued from previous page)

```
// ... or with the function ascii::columns():
ascii::read_table("my_table.dat", 2, ascii::columns(2,_), y);
```

Reading 2D columns. With the interface describe above, if you need to read `N` columns, you need to list `N` vectors when calling the function. This can be cumbersome for tables with a large number of columns. In the cases where it makes sense, you can choose to combine `n` adjacent columns of the ASCII table into a single 2D vector. The first dimension (`v.dims[0]`) will be the number of rows, and the second dimension (`v.dims[1]`) will be the number of columns (`n`). This can be done by specifying `ascii::columns(n,v)`. With the example table above:

```
// Declare the vectors that will receive the data
vec1u id;
vec2f v;

// Read the data, merging the 'x' and 'y' columns into the 2D vector 'v'
ascii::read_table("my_table.dat", 1, id, ascii::columns(2,v));

// The data in 'v' is stored such that
vec1f x = v(_,0);
vec1f y = v(_,1);
```

Multiple 2D columns. The trick of reading 2D columns can be extended to read several columns into multiple 2D vectors by following a pattern. A typical case is when you have, say, three quantities ‘A’, ‘B’, and ‘C’ listed in the table, each with their values and uncertainties:

```
# abc_data.dat
# id    A   Aerr     B   Berr     C   Cerr
   0   10    1.0     1    0.1    -1      1
   5   -1    3.5     2    0.2     1      1
   6    0      6     3    0.2     1      2
   ...
```

This table can be read easily into two 2D vectors `value` and `uncertainty` by using `ascii::columns(3, value,uncertainty)`. This is interpreted as “read 3 sets of columns, each containing value and uncertainty”:

```
// Declare the vectors that will receive the data
vec1u id;
vec2f value, uncertainty;

// Read the data
ascii::read_table("abc_data.dat", 2, id, ascii::columns(3,value,uncertainty));

// The data is stored in 'value' and 'uncertainty' such that
vec1f a    = value(_,0);
vec1f aerr = uncertainty(_,0);
vec1f b    = value(_,1);
vec1f berr = uncertainty(_,1);
vec1f c    = value(_,2);
vec1f cerr = uncertainty(_,2);
```

This can also be mixed with the placeholder symbol _ to skip column (see above):

```
// If we only wanted to read the values, and not the uncertainties, we could write:
ascii::read_table("abc_data.dat", 2, id, ascii::columns(3,value,_));
```

15.2 ascii::write_table

```
void ascii::write_table(std::string f, const Args& ... args); // [1]

void ascii::write_table(std::string f, const ascii::output_format& o, const Args& ...␣
↪args); // [2]

void ascii::write_table(std::string f, ftable(...)); // [3]

void ascii::write_table(std::string f, const ascii::output_format& o, ftable(...)); //
↪ [4]
```

These functions will write the data of the vectors listed in `args` into the file whose path is given in `f`. The data will be formated in a human-readable form, colloquially called "ASCII" format. In all cases, all columns must have the same number of rows, otherwise the function will report an error.

Function [1] uses a "standard" format, where the data is written in separate columns, separated and automatically aligned by white spaces. See below for more detail on how the table data is written. Here is a simple example.

Example:

```
// We have some data
vec1u id = {1,2,3,4,5};
vec1i x = {125,568,9852,12,-51};
vec1i y = {-56,157,2,99,1024};

// Write these in a simple ASCII file
ascii::write_table("my_table.dat", id, x, y);
```

The content of `my_table.dat` will be:

```
1  125  -56
2  568  157
3 9852    2
4   12   99
5  -51 1024
```

Note: Human-readable formats are simple, and quite convenient for small files. But if the volume of data is huge, consider instead using *C interface* instead. This will be faster to read and write, and will also occupy less space on your disk.

Function [2] allows you to change the output format by specifying a number of options in the option structure `o`. This structure has the following members:

```
struct output_format {
    bool        auto_width   = true;
    uint_t      min_width    = 0;
    std::string delim        = " ";
    std::string header_chars = "# ";
    vec1s       header;
};
```

- `auto_width`. When set to `true` (the default), the function will compute the maximum width (in characters) of each column before writing the data to the disk. It will then use this maximum width to nicely align the data in each column (always aligned to the right). Note that it also takes into account the width of the header string (see below). This two-step process reduces performances a bit, and for large data sets you may want to disable it by setting this option to `false`. In this case, either the data is written without alignment (still readable by a machine, but not really by a human), or with a fixed common width if `min_width` is set to a positive value.

- `min_width`. This defines the minimum width allowed for a column, in characters. The default is zero, which means columns can be as narrow as one single character if that is all the space they require.
- `delim`. This string defines which character(s) should be used to separate columns in the file. The default is to use a single white space (plus any alignment coming from adjusting the column widths).
- `header` and `header_chars`. These variables can be used to print a header at the beginning of the file, before the data. This header can be used by a human (or, possibly, a machine) to understand what kind of data is contained in the table. The header will be written on a single line, starting with `header_chars` (the header starting string). Then, each column written in the file must have its name listed in the `header` array, in the same order as given in `args`.

Some pre-defined sets of options are made available for simplicity:

- `ascii::output_format::standard()`. This is the default behavior of `write_table()`, when no options are given (see default values above). With this setup, columns in the file are separated by at least one white space character (and possibly more, for alignment). The data in a given column is automatically aligned.

  Using this format is done as follows:

```
// We have some data
vec1u id = {1,2,3,4,5};
vec1i x = {125,568,9852,12,-51};
vec1i y = {-56,157,2,99,1024};

// Write these in a simple ASCII file
ascii::write_table("my_table.dat", output_options::standard(), id, x, y);
// This is equivalent to
ascii::write_table("my_table.dat", id, x, y);
```

  This would result in the table:

```
1  125  -56
2  568  157
3 9852    2
4   12   99
5  -51 1024
```

  You can also use it as a starting point to create customized options:

```
ascii::output_format opts = ascii::output_format::standard();
opts.header_chars = "% ";          // begin header line with '% ' instead of '# '
opts.header = {"index", "X", "Y"}; // specify column names
ascii::write_table("my_table.dat", opts, id, x, y);
```

  This would produce instead:

```
% index    X    Y
      1  125  -56
      2  568  157
      3 9852    2
      4   12   99
      5  -51 1024
```

- `ascii::output_format::csv()`. This preset enables writing comma-separated values (CSV) tables. In these tables, columns are separated by a single comma (`','`), and the data is not aligned at all:

```
1,125,-56
2,568,157
3,9852,2
4,12,99
5,-51,1024
```

The information below applies to any type of table.

Output format for numbers. When providing vectors of floats or doubles, these functions will convert the values to strings using the default C++ format. See discussion in *String conversions*. When this is not appropriate, you can use the `format::...` functions to modify the output format, as you would with `to_string()`. For example:

```
// Data
vec1i x = {0, 1, 2, 3, 4};
vec1f y = {1e-5, 0.0, 1e5, 1.2, 100.5};

// Standard format
ascii::write_table("my_table.dat", x, y);
```

The default format produces the following table:

```
0   1e-05
1       0
2 100000
3     1.2
4   100.5
```

Using a custom format:

```
// Custom format
ascii::write_table("my_table.dat", x, format::scientific(y));
```

This produces instead:

```
0 1.000000e-05
1 0.000000e+00
2 1.000000e+05
3 1.200000e+00
4 1.005000e+02
```

Writing 2D vectors. These functions support writing 2D vectors as well. They are interpreted as containing multiple columns: the number of rows is the first dimension of the vector (`v.dims[0]`), and the number of columns is the second dimension (`v.dims[1]`). For them to be recognized by the function, you must wrap them in `ascii::columns(n,v)`, where `n` is the number of columns. For example:

```
// We have some data
vec1u id = {1,2,3,4,5};
vec2f value(5,2);
value(_,0) = {0.0, 1.2, 5.6, 9.5, 1.5};
value(_,1) = {9.6, 0.0, 4.5, 0.0, 0.0};

// Write these in a simple ASCII file
ascii::write_table("my_table.dat", id, ascii::columns(2,value));
```

The content of `my_table.dat` will be:

```
1   0 9.6
2 1.2   0
3 5.6 4.5
4 9.5   0
5 1.5   0
```

Multiple 2D vectors. As for `ascii::read_table()`, you can use the above mechanism to write multiple 2D columns following a pattern by listing them in the `ascii::columns()`. For example, `ascii::columns(n, value, uncertainty)` will write `n` pairs of columns, with `value` and `uncertainty` in each pair.

```
// We have some data
vec1u id = {1,2,3,4,5};
vec2f value(5,2);
value(_,0) = {0.0, 1.2, 5.6, 9.5, 1.5};
value(_,1) = {9.6, 0.0, 4.5, 0.0, 0.0};
vec2f uncertainty(5,2);
uncertainty(_,0) = {0.01, 0.03, 0.05, 0.09, 0.01};
uncertainty(_,1) = {0.02, 0.05, 0.01, 0.21, 0.04};

// Write these in a simple ASCII file
ascii::write_table("my_table.dat", id, ascii::columns(2,value,uncertainty));
```

The content of `my_table.dat` will be:

```
1   0 0.01 9.6 0.02
2 1.2 0.03   0 0.05
3 5.6 0.05 4.5 0.01
4 9.5 0.09   0 0.21
5 1.5 0.01   0 0.04
```

Easy headers. Functions [3] and [4] will adopt the same output format as functions [1] and [2]. The only difference is that they will automatically create the header based on the names of the C++ variables that are listed in argument. To do so, you must use the `ftable()` macro to list the data to be written. For example:

```
// Write these in a simple ASCII file
ascii::write_table("my_table.dat", ftable(id, x, y));
```

This also works for 2D vectors. In such cases, `_i` is appended to the name of the vector for each column `i`. If you need better looking headers, you can always write them manually using function [2].

Function [4] allows combining this convenient shortcut with other output options:

```
// Write these in a CSV file
ascii::write_table("my_table.dat", ascii::output_format::csv(), ftable(id, x, y));
```

In this case the `header` vector is overriden by the values produced by `ftable()`.

CHAPTER 16

FITS files

16.1 Overview

Defined in header `<vif/io/fits.hpp>`.

16.1.1 A few words on the FITS format

The FITS (Flexible Image Transport System) format is a general purpose file format originally developed for astrophysics data. In particular, FITS files can store images with integer or floating point pixel values, image cubes (with more than two dimensions), but also binary data tables with an arbitrary number of columns and rows. Using a meta-data system (keywords in a header), FITS files usually carry a number of important additional informations about their content. For example, for images files, this can be the mapping between image pixels and sky coordinates (WCS coordinates), or the physical unit of the pixel values.

A single file can contain any number of Header-Data Units (HDUs), which can be seen as independent "extensions" of the file. Each such extension can contain either image data or table data, and has its own header and keywords for storing meta-data. The first HDU of a FITS file has a special status and is called the "primary HDU"; it can only contain image data. For this reason FITS tables always have an empty primary HDU, and the table data located in the first extension.

Writing tabulated data in a binary FITS file is a space-efficient and fast way to store and read non-image data, vastly superior to using human-readable ASCII tables. FITS tables come in two fashions: row-oriented and column-oriented tables. In row-oriented tables, all the data about one row (e.g., about one galaxy in the table) is stored contiguously on disk. This means that it is very fast to retrieve all the information about a given object. In column-oriented tables however, a whole column is stored contiguously in memory. This means that it is very fast to read a given column for all the objects in the table. This distinction is analogous to the dilemma of choosing between a structure-of-array (column-oriented) or an array-of-structures (row-oriented).

Since vif vectors are also contiguous in memory and are used to store data from a given column, the column-oriented format is the most efficient, and is therefore the default format in vif. An additional benefit of this format is that it allows storing columns of different lengths, which is particularly useful to carry meta-data that would be hard to store in FITS keywords. The column-oriented format is not well known, but most softwares and libraries do support it.

Topcat does, and in IDL column-oriented FITS files are supported naturally by the mrdfits and mwrfits procedures. But since row-oriented files are nevertheless very common, vif is capable of reading and writing in both formats.

16.1.2 The FITS implementation in vif

Because FITS files can have a complex structure, vif offers *three* different interfaces to interact with them, which offer different levels of abstraction, capabilities, and ease of use.

The lowest level interface is the raw C interface of the CFITSIO library. This interface allows the finest control over the FITS file structure, however it is more cumbersome to use and more error prone. Unless you need to do something very specific that the other interfaces cannot achieve, it is not recommended to use the CFITSIO interface directly. If you do need to use it, please refer to the official CFITSIO documentation.

The second interface already has a significantly higher abstraction level. It is available through the classes `fits::input_table`, `fits::output_table`, `fits::table`, `fits::input_image`, `fits::output_image`, and `fits::image`. This is an object-oriented interface, where an instance of one of the classes above represents an open FITS file, through which you can navigate to read and/or write data. This is versatile enough to allow you to create multiple table or image extensions, modify data inside the file (i.e., one pixel, or one cell in a table), and edit individual header keywords.

The last and simplest interface is provided through the free functions `fits::read_table()`, `fits::write_table()`, `fits::update_table()`, `fits::read_image()`, `fits::write_image()`, and `fits::update_image()`. These allow you to read/write/update data from a single extension in a FITS file, all with a single function call. They are most convenient when dealing with simple FITS files, however they are less powerful than the object-oriented interface described above.

The object-oriented interface is implemented directly on top of the CFITSIO C API, and the simple free-function interface is implemented on top of the object-oriented interface. These interfaces are not based on the official CFITSIO C++ wrapper, CCFITS, mostly because its level of abstraction is too high such that it was not possible to implement all the required features with it. This official wrapper could have simply been adopted as the default FITS implementation in vif, however its design choices conflicted too blatantly with the vif "mindset" (different choice of container types, class hierarchy too deep, and notion of HDU too central).

16.1.3 General information on the FITS classes

The class hierarchy is roughly modeled around the `std::iostream` interface:

A file can be opened directly by providing the file name in the constructor, or using the `open()` member function. Memory and other resources are freed automatically in the destructor of the object, however it is possible to close the file early if needed using the `close()` member function. When the file is open, classes `fits::input_image`, `fits::output_image`, and `fits::image` will automatically go to the first extension of the file containing image data, likewise with `fits::input_table`, `fits::output_table`, and `fits::table` and table data.

Any invalid operation will raise an exception of the type `fits::exception` (with `fits::exception::what()` providing a text message describing the error). It is safe to recover from exceptions raised when attempting invalid data read operations (incorrect image format, unknown table columns, . . .). However, write operations are not exception-safe; if they happen, the only safe course of action is to close the file and consider its content as corrupted.

If an instance of any of these classes is shared by multiple execution threads, all operations (even simple queries about the state of the file) must be protected by a mutex lock. In contrast, multiple instances can be used in multiple threads without locking as long as:

- each instance is used by a single thread only,

- all instances are pointing to different files, or instances pointing to the same file are all performing read operations only.

16.2 Open/close files

Defined in header `<vif/io/fits.hpp>`.

16.2.1 open

```
void fits::file_base::open(std::string f);
```

This function will open a new FITS file for read or write access (depending on the actual class that is used).

Example:

```
// Open a file directly in the constructor
fits::input_image img1("my_image.fits");

// Open a file later using open()
fits::input_image img2;
img2.open("my_image.fits");
```

If there is already a file open when `open()` is called, that file is closed before the new file is opened.

When requesting only read access (i.e., with the classes `fits::input_image` or `fits::input_table`), an exception will be raised if the file does not exist or cannot be accessed with read permission. When requesting only write access (i.e., with the classes `fits::output_image` or `fits::output_table`), a new file will be created regardless of whether a file with the provided name already exists or not, and an exception will be raised if the file cannot be created. When requesting read/write access (i.e., with the classes `fits::image` or `fits::table`), an exception will be raised if the file does not exist or cannot be accessed with read/write permission.

This function is partially exception-safe: if a file was previously open before the call and an exception is raised, that file will be closed and no data will be lost. Aside from this minor point, the instance can be used safely after recovering from an exception raised by `open()`.

It is possible to open the same file multiple times as different objects, but this is not safe when performing write operations. It is, however, perfectly safe to read data from the same file through two objects:

```
// Open the same file twice for reading data
fits::input_image img1, img2;
img1.open("my_image.fits");
img2.open("my_image.fits");
// Perform read operations (safe)
vec2d image1, image2;
img1.read(image1);
img1.read(image2);
```

16.2.2 close

```
void fits::file_base::close() noexcept;
```

This function will close the currently opened FITS file (if any). If data was written to the file, it will be force-flushed to the disk to ensure no data is lost before the file is closed.

This function is called automatically in the destructor, so you do not need to call it explicitly unless you want to close the file before the end of the object's lifetime.

If the file cannot be properly closed for any reason, this function will not raise an exception and simply consider the file as closed.

Example:

```
// Open a file
fits::input_image img("my_image.fits");
// Perform some operations
// ...
// Close the file early
img.close();
// A new file must now be opened before doing further operations
```

16.2.3 is_open

```
bool fits::file_base::is_open() const noexcept;
```

This function checks if a file is currently open.

Example:

```
// Create a FITS image object with no opened file yet
fits::input_image img;
img.is_open(); // false
// Open a file
img.open("my_image.fits");
img.is_open(); // true
```

16.2.4 filename

```
const std::string& fits::file_base::filename() const noexcept;
```

This function returns the name of the currently opened file (or blank if no file is opened).

Example:

```
fits::input_image img("my_image.fits");
img.filename(); // "my_image.fits"
```

16.2.5 flush

```
void fits::output_file_base::flush();
```

This function will perform any pending write operation to the disk and only return when all the data has been written. It will perform a full update of the file, including binary data and header data. Only available for output files. Will throw an exception if no file is currently open.

Indeed, as with any disk write operation in the C++ standard library, CFITSIO write operations use a write buffer which is only written to the disk occasionally, rather than on any write operation. This is done for performance reasons. The downside of this approach is that the data is not always immediately written to the disk, even after a call to `write()`

has returned. This usually is not an issue, except when one wants to access the content of the file while it is being written, or if the program crashed while data was not yet written to the file.

Example:

```
// Open a FITS image for writing
fits::output_image img("my_image.fits");
// Write some data
img.write(data);
// Force writing data to disk now
img.flush();
```

16.2.6 flush_buffer

```
void fits::output_file_base::flush_buffer();
```

This function will perform any pending write operation to the disk and only return when all the data has been written. Contrary to `flush()`, it will only flush the binary data, and not the header data. This will be faster but less complete; only use this if you know the header data is likely to already be up-to-date. See `flush()` for more information. Only available for output files. Will throw an exception if no file is currently open.

16.3 Managing HDUs

Defined in header `<vif/io/fits.hpp>`.

16.3.1 hdu_count

```
uint_t fits::file_base::hdu_count() const;
```

This function returns the number of HDUs (or extensions) currently present in the file. This includes the "primary HDU" (extension with ID `0`), and therefore should always be larger or equal to one. Will throw an exception if no file is currently open.

Example:

```
// Open a FITS image for writing
fits::output_image img("my_image.fits");
img.hdu_count(); // 1 (only the primary HDU)
// Reach some other HDU
img.reach_hdu(1);
img.hdu_count(); // 2
```

16.3.2 current_hdu

```
uint_t fits::file_base::current_hdu() const;
```

This function returns the ID of the current HDU (or extension). The "primary HDU" has ID of `0`, and every following HDU has its ID incremented by one. Will throw an exception if no file is currently open.

Example:

```
// Open a FITS image for writing
fits::output_image img("my_image.fits");
img.current_hdu(); // 0 (the primary HDU)
// Reach some other HDU
img.reach_hdu(1);
img.current_hdu(); // 1
```

16.3.3 hdu_type

```
fits::hdu_type fits::file_base::hdu_type() const;
```

This function attempts to identify the content in the current HDU, determining whether it is an image (`fits::image_hdu`), a table (`fits::table_hdu`), or an empty HDU (`fits::empty_hdu`). If it could not decide, it returns `fits::null_hdu`. The function will throw an exception if the header contains keywords with invalid values, or if no file is currently open.

Example:

```
// Open a FITS image for writing
fits::output_image img("my_image.fits");
img.hdu_type(); // fits::empty_hdu (the primary HDU is initially empty)
// Write some data
img.write(data);
img.hdu_type(); // fits::image_hdu
```

16.3.4 reach_hdu

```
void fits::file_base::reach_hdu(uint_t hdu);
```

This function attempts to reach the requested HDU to start reading/writing data from/to it. If this HDU does not exist and the file was opened only with read access, the function will throw an exception. If the file was opened with write access, the function will insert as many empty HDUs as required so that the requested HDU exists, and then reach it for read/write operations. Will throw an exception if no file is currently open.

Example:

```
// Open a FITS image for writing; we start at the primary HDU (ID 0)
fits::output_image img("my_image.fits");
// Reach some other HDU
img.reach_hdu(2);
// Write data there
vec2d data(10,10);
img.write(data);
// The file now contains:
//  - an empty primary HDU (ID 0)
//  - an empty first extension (ID 1)
//  - the image data in the second extension (ID 2)
```

16.3.5 remove_hdu

```
void fits::file_base::remove_hdu();
```

This function removes the current HDU from the file. If other HDUs existed after the current HDU, their IDs are decreased by one, to fill the gap. This function will throw an exception when attempting to remove the primary HDU, as by definition it cannot be removed. Will throw an exception if no file is currently open. Only available for output files.

Example:

```
// Open a FITS image for writing; we start at the primary HDU (ID 0)
fits::output_image img("my_image.fits");
// Reach some other HDU
img.reach_hdu(2);
// Write some data
vec2d data(10,10);
img.write(data)
// The file now contains:
//  - an empty primary HDU (ID 0)
//  - an empty first extension (ID 1)
//  - the image data in the second extension (ID 2)

// Move to the HDU 1
img.reach_hdu(1);
// Remove it
img.remove_hdu();
// The file now contains:
//  - an empty primary HDU (ID 0)
//  - the image data in the first extension (ID 1)
```

16.3.6 axis_count

```
uint_t fits::file_base::axis_count() const;
```

This function returns the number of axes of the data located in the current HDU. For image data, this is the number of axes (1 for 1D data, 2 for images, 3 for cubes, etc.). For table data and empty HDUs, the function returns zero. Will throw an exception if no file is currently open.

Example:

```
// Open a FITS image for writing
fits::output_image img("my_image.fits");
img.axis_count(); // 0 (the primary HDU is initially empty)
// Write some data
vec2d data(10,10);
img.write(data);
img.axis_count(); // 2
```

16.3.7 image_dims

```
vec1u fits::file_base::image_dims() const;
```

This function returns the dimensions of the image in the current HDU. If the current HDU is empty or contains table data, this returns an empty vector. Will throw an exception if no file is currently open.

Example:

```
// Open a FITS image for writing
fits::output_image img("my_image.fits");
img.image_dims(); // {} (the primary HDU is initially empty)
// Write some data
vec2d data(8,10);
img.write(data);
img.image_dims(); // {8,10}
```

16.4 Header & keywords

Defined in header `<vif/io/fits.hpp>`.

16.4.1 has_keyword

```
bool fits::file_base::has_keyword(std::string name) const;
```

This function checks if a given keyword exists in the header of the current HDU. This check is not case-sensitive, and the function automatically supports long keyword names specified with the `HIERARCH` convention; it is not necessary to specify the `HIERARCH` explicitly. Will throw an exception if no file is currently open.

Example:

```
// Open a FITS image
fits::input_image img("my_image.fits");
img.has_keyword("BUNIT"); // does this image have a unit?
```

16.4.2 read_keyword

```
template<typename T>
bool fits::file_base::read_keyword(std::string name, T& value) const;
```

This function checks if a given keyword exists in the header of the current HDU, and if the keyword exits, attempts to read its value and store it into the variable `value`. This check is not case-sensitive, and the function automatically supports long keyword names specified with the `HIERARCH` convention; it is not necessary to specify the `HIERARCH` explicitly. If any of these steps fail, the content of `value` is unchanged and the function returns `false`. Will throw an exception if no file is currently open.

Example:

```
// Open a FITS image
fits::input_image img("my_image.fits");
std::string unit;
if (img.read_keyword("BUNIT", unit)) {
    // We know the unit of the image
}
double frequency;
if (img.read_keyword("FREQ", frequency)) {
    // We know the frequency at which the image was obtained
}
```

16.4.3 write_keyword, add_keyword

```
template<typename T>
void fits::output_file_base::write_keyword(std::string name, const T& value); // [1]
template<typename T>
void fits::output_file_base::add_keyword(std::string name, const T& value); // [2]
```

These functions write the given keyword into the header of the current HDU, setting its value to the provided `value`. If a keyword with this name already exist, function [1] will update its value, while function [2] will simply ignore it and add a new keyword with the same name at the end of the header (it is indeed possible to have multiple keywords with the same name). If the keyword name is longer than 8 characters, CFITSIO will automatically write the keyword with the `HIERARCH` convention; it is not necessary to specify the `HIERARCH` explicitly. Will throw an exception if no file is currently open.

Example:

```
// Open a FITS image
fits::output_image img("my_image.fits");
vec2d data(10,10);
img.write(data);
img.write_keyword("BUNIT", "W/m2/sr"); // write a string
img.write_keyword("FREQ", 1.4e9);       // write a number
```

16.4.4 remove_keyword

```
void fits::output_file_base::remove_keyword(std::string name);
```

This function will remove the first keyword in the header whose name matches the provided string. No error is generated if no such keyword exists. If the keyword name is longer than 8 characters, CFITSIO will automatically write the keyword with the `HIERARCH` convention; it is not necessary to specify the `HIERARCH` explicitly. Will throw an exception if no file is currently open.

Example:

```
// Open a FITS image
fits::output_image img("my_image.fits");
vec2d data(10,10);
img.write(data);
img.write_keyword("BUNIT", "W/m2/sr"); // write a string
img.remove_keyword("BUNIT");            // we changed our mind, remove it
```

16.5 FITS images

Defined in header `<vif/io/fits.hpp>`.

16.6 FITS tables

Defined in header `<vif/io/fits.hpp>`.

16.7 C interface

Defined in header `<vif/io/fits.hpp>`.

16.7.1 cfitstio_status

```
int fits::file_base::cfitstio_status() const noexcept;
```

This function returns the current CFITSIO error code. Only useful for debugging purposes. If no file is currently open, it will return zero.

Example:

```
fits::input_image img("my_image.fits");
img.cfitsio_status(); // most likely 0
```

16.7.2 cfitsio_ptr

```
fitsfile*       fits::file_base::cfitsio_ptr()       noexcept;
const fitsfile* fits::file_base::cfitsio_ptr() const noexcept;
```

These functions returns the underlying CFITSIO file pointer. This is useful if you need to perform an operation that is not available as part of the C++ interface. It is safe to perform any operation with this pointer and then fall back to the C++ interface, however if you do so you must call the `update_state()` function before using any function of the C++ interface.

If no file is currently open, it will return a null pointer.

Example:

```
// Open a FITS image
fits::input_image img("my_image.fits");
// Get the underlying CFITSIO pointer
fitsptr* fptr = img.cfitsio_ptr();
// Use the pointer with the raw C interface
// ...
// Update the internal state
img.update_internal_state();
// Continue using the C++ interface
```

16.7.3 update_internal_state

```
void fits::file_base::update_internal_state();
```

This function is called internally by `open()` and `reach_hdu()`, and is used to update the internal state of the C++ wrapper based on the current content of the file. You only need to use this function if you perform operations on the file using the raw CFITSIO interface. See `cfitsio_ptr()` for more information. Will throw an exception if no file is currently open.

CHAPTER 17

Measuring time

CHAPTER 18

Mathematics

CHAPTER 19

Multi threading

CHAPTER 20

Image manipulation

CHAPTER 21

Astronomy library