# Current Users of *DIPlib* and *DIPimage*

The following people use *DIPlib* and *DIPimage* in their work, and might write a letter of support for the current project:

* Laura Kegelmeyer, National Ignition Facility, LLNL, California
* Sylvain Costes, Life Sciences Division, LBNL, California
* Stephen Lockett, National Cancer Institute, Maryland
* Rodrigo Fernandez-Gonzalez, Institute of Biomaterials & Biomedical Engineering, University of Toronto, Canada
* Carlos Ortiz de Solorzano, Centre for Applied Medical Research, University of Navarra, Pamplona, Spain
* Michal Kozubek, Centre for Biomedical Image Analysis, Masaryk University, Brno, Czech Republic
* Keith Lidke, Department of Physics & Astronomy, University of New Mexico, New Mexico
* Rainer Heintzmann, King’s College, London, UK; Institute of Physical Chemistry, Friedrich-Schiller-Universität Jena, Germany; and Leibniz Institute of Photonic Technology, Jena, Germany
* Etc.

Would Anne Carpenter of *CellProfiler* write a letter of support? Would that be useful?

# Implementation notes and thoughts

## Support

Repository: G*it* (on GitHub?). This will make it easier to collaborate and accept patches and additions from outside people (merging repositories rather than patching our local copy and committing). I suggest using GitHub (or similar) hosting: it provides good tools and services, it uses up-to-date software, there’s less downtime, and it doesn’t cost a dime. Why do it ourselves?

Web site: GitHub also hosts web sites, we probably should move diplib.org to point to the GitHub website (unless Google allows uploading HTML files, the web page editing system they have is crap). Alternatively, we could redirect www.diplib.org to diplib.github.org or whatever.

Documentation: D*oxygen*. This will make sure the function signatures in the documentation match those in the header files. Documentation written in the source code files also makes it a little easier to remember to update documentation when updating code.

Build system: *Waf*? I don’t like *CMake*! *Autotools* is no good either. If we keep using Mike’s build system, we will have to do major changes to support C++, Python, and whatever else we want to do. It’ll be more effort to include 3rd party libraries in our build tree. Let’s use a system that can automatically compile dependencies.

Other: *Gerrit* (integrates with *Git*), it allows any authorised user to submit changes to the master repository, and other users to comment and/or approve commits (otherwise, *Git* requires one master person to merge repositories of users who propose fixes/updates/additions). *KWStyle* for checking source code style (also integrates with *Git*), it can give errors during compilation or during commit if the source code is not formatted according to a style.

## The Image object

Images can have any number of dimensions and data types, as current. We will add the option of tensor images by adding dimensions to the data block. Tensor dimensions are administered separately from spatial dimensions, functions should not ignore this separation. Colour images have three or four tensor components in one dimension, and have a colour space info structure. Image methods should allow moving dimensions from being tensorial to spatial and vice versa (as a cheap operation). See below for thoughts on implementation of tensor dimensions.

A version of the constructor should take a data pointer and a deallocator function pointer, and create an object from it. The deallocator function pointer can be NULL to avoid deallocating the data. We can make cast operators from/to *OpenCV* and other libraries in independent include files that are not used by the library itself (so as to avoid dependencies on *OpenCV*). Library users can take these include files into their programs if they need them.

There should also be a facility for allowing an external interface to allocate the data array. The image object has a ForgeHandler() function that will be called to allocate the data. For the MATLAB interface, this function will be set to a function that creates an appropriate mxArray, and returns its data pointer. This function also returns a deallocator functor (object that can be called like a function, the object can hold additional data, for example the mxArray pointer from which the data pointer was taken), as well as the stride arrays (which the interface might want to control). The data pointer is stored in the std::shared\_ptr member, which takes care of deallocating the data when needed. The std::shared\_ptr stores the deallocator function also.

By using a std::shared\_ptr, we can have multiple image objects point to the same data block. When the last of these images is destroyed, the memory block is freed. We need a method to query whether the data pointer is being shared or not.

The image object should have a flag that, when set, avoids its data segment to be re-allocated. When the output image of a function has such a flag set, functions will simply convert their output to match that of the image. If such conversion is not possible, maybe because the image size is incorrect, an exception is thrown. This will allow a function such as dip::Gauss(in,out,sigma), which would normally produce a floating-point output, to produce an integer output instead. This beats adding “output data type” parameters to functions, as the functionality is out of the way unless one is interested in it.

Something to consider: Can we implement image arrays as std::vector<dip::Image&>? It would avoid copying image headers (data is never copied anyway), but it would probably be impossible to create one of these in a function and pass it as output argument. Maybe it’s better to consider the vector as the container for the images.

## Tensor dimensions

It makes intuitive sense for tensor dimensions to be indexed as (row number, column number), and to not allow trailing singleton dimensions (these can automatically be removed). Furthermore, it would be beneficial to support symmetric tensors without the overhead of duplicating data and computations (these are used extensively in things such as the structure tensor, the Hessian, Harris corner detector, etc.).

There are two possible ways of implementing tensor images:

#1: add two dip::IntegerArrays to the image object: tensordimensions and tensorstrides.

#2: add an int ntensorelements, an int tensorrows, an int tensorstride, and an enum class tensorshape to the image object. tensorshape would contain options such as scalar, columnvector, rowvector, matrix\_rowmajor, matrix\_columnmajor, symmetricmatrix, uppertriangularmatrix, etc.

Under #1, any tensor dimensionality is supported. Individual tensor dimensions can be “converted” to spatial dimensions. This provides a lot of flexibility.

Under #2, only tensors with up to 2 dimensions are allowed. However, it will be possible to implement symmetric tensors in an efficient way. Furthermore, it is less likely than under #1 for a scalar plane extracted from a tensor image to have strides that do not allow to transverse the whole image as a single line, and it is easier to transverse the tensor elements of a pixel (only need one stride in a single loop, rather than a vector of strides in a multi-dimensional loop).

I have never seen any applications for tensors with more than 3 dimensions in image analysis, so the flexibility of #1 seems less important right now than the efficiency of #2.

More details for #2:

* scalar: other values can be ignored
* vector: ntensorelements = the length of the vector
* matrix (N×M): ntensorelements=N\*M, tensorrows=N
* symmetricmatrix (N×N): ntensorelements=N\*(N-1)/2, tensorrows=N

Transposing the tensor, reshaping it, etc. are all trivial operations.

## Indexing syntax

There is two types of indexing: into pixel dimensions and into tensor dimensions. We will overload the [] operator to index into tensor dimensions. Picking one of the “planes” is much more common than picking a pixel. No data copying is necessary, the output will point to the same data block.

img[i] // Get tensor element i, using linear indexing  
img[IntegerArray{i,j}] // Get tensor element (i,j)

Both these notations return a dip::Image object.

Tensors are stored column-wise, to be consistent with expectations. That is, the first index is down (row number), the second one is to the right (column number). But we do start at 0, because starting at 1 makes no sense. :)

For example:

dip::Image img = dip::Read( "filename.tif" );  
img = img[2]; // use only the blue channel.

To index into spatial dimensions we use the at() method (as in the standard library):

img.at(i) // Get pixel at linear index i  
img.at(x,y) // Get pixel at coordinates (x,y), only for 2D images  
img.at(x,y,z) // Get pixel at coordinates (x,y,z), only for 3D images  
img.at(IntegerArray{...}) // Get pixel at given coordinates, general syntax

All these notations return a dip::Pixel object. This object simply contains a pointer to the pixel data in the image, has a reference to the image’s tensor size and tensor stride arrays, and knows the data type also. This allows the user to read or write the pixel’s value:

img.at(x,y) += 5;

The dip::Pixel object can be casted to a double or an int to get the first tensor component. It can be indexed using the [] notation, just like the image, and also has similar arithmetic and comparison functions. The following two notations do exactly the same thing:

img.at(x,y)[i] // Get the tensor value pixel at (x,y), then get the tensor component i  
img[i].at(x,y) // Extract an image for the tensor component i, then index pixel (x,y)

A dip::pixel object always points to an image. When the image is destroyed, deallocated or reallocated, the pixel object becomes invalid.

img.at(mask); img.at(coord\_array);

Yet another option would be to overload the () operator, but again, this might cause too much confusion, as in C++ that operator is used for function calls. *OpenCV* uses references, iterators and more to allow this type of functionality. I’m not sure if we want to copy this. I’m open for suggestions.

Casting an image to e.g. double or int will extract the very first pixel from the image. Thus

double v = (double)img.At(x,y);

should be the simple way of extracting a pixel value.

## More complex indexing syntax

Sometimes people want to select multiple pixels at once. There are two cases: regular subsampling, and irregular mask indexing.

First case: The regular case yields a new image with the same dimensionality as the source image, but is smaller in size. The new image points to the same data block as the original image, so it can be used to apply an operation to a single channel or a single plane, for example. Per dimension, you need to choose a start index, a stop index and a step size. We can support two ways of accomplishing this. One uses a dip::Range struct:

dip::Range rx {xstart, xstop, xstep};  
dip::Range ry {ystart, ystop, ystep};  
img.at(rx,ry); // Returns an image that points to the same data as img

img.at(range) // For 1D images  
img.at(xrange, yrange) // For 2D images  
img.at(xrange, yrange, zrange) // For 3D images  
img.at(RangeArray {...}) // General case for ND images  
img[range] // Into tensor dimensions

The alternative method is a function as we have in the current *DIPlib*:

img = GetROI( img, start, stop, step );  
GetROI( img, img, start, stop, step );

This function takes start, stop and step as IntegerArrays. We should want to redefine this to allow sampling the tensor dimensions also. Otherwise the user can turn tensor dimensions into spatial dimensions.

Second case: Indexing with a mask image produces a collection of pixels that is not regular, and cannot be represented in a normal image. The easiest way to support this type of indexing is to copy the selected pixels into a 1D image. Being a copy of the data, the resulting image cannot be used to modify the original image. A second indexing operation needs to be applied to copy the modified pixels back into the original image:

roi = img.applymask( mask ); // Pixels where mask==true are copied to roi  
img.applymask( mask, roi ); // Pixels where mask==true are overwritten with the values in roi

This is quite ugly. We could think also of allowing img.at(mask) or img[mask].

The alternative is to make the dip::Image object more flexible, such that it can represent a masked set of pixels also. All functions will have to be aware of this, test whether the input image is a masked image or a regular image. This is difficult. Yet another alternative is to make a dip::IrregularImage object, which will have its own overload arithmetic operators and some other things, but not much else (as most functions could not work on such irregular data anyway). I don’t like the duplication of code that this represents. *DIPlib* currently allows a mask image input as a second argument to some functions. We could overload these functions to accept such an irregular image:

void dip::Filter( dip::Image in, dip::Image mask, ... );  
void dip::Filter( dip::IrregularImage in, ... ) { dip::Filter( in.Image, in.Mask, ... )}

Not sure if this would be useful. Probably more work than it’s worth.

## Initializing values of small images/pixel objects

Sure it is possible to use img.at()[] to write values into a small image, but must be a better way. Small images are used, e.g., when doing arithmetic on tensor images. 0D scalar and vector images are trivial:

dip::Image img( 1 ); // This is a 0D sfloat image with pixel value 1  
dip::Image img { 1 }; // Idem  
dip::Image img { 1, 2, 3 }; // This is a 0D sfloat value with a 3-vector as pixel, value [1,2,3]T

How about 0D matrix images (e.g. a rotation matrix)? Maybe something like this could work, with an initializer\_list object in there (not sure this would compile?):

dip::Image img( size, tensorsize, dip::DT::SFLOAT, { 0, 1, 2, 3 } );  
 // 0, 1, 2 and 3 will be the first four values in the data block.

## Arithmetic and comparisons

Of course all operators will be overloaded. These will be implemented as calls to

dip::Arithmetic( in1, in2, out, op )  
dip::Comparison( in1, in2, out, op )

where op is an enum specifying which operator to apply. out can be equal to in1 or in2 for inplace operation. Thus, we can do:

img1 += img2;  
dip::Arithmetic( img1, img2, img1, "+" );

dip::Arithmetic will do tensor arithmetic also. Comparisons can do per-element comparison for tensors, or we can implement other alternatives if necessary. Both functions do automatic singleton dimension expansion (set stride to 0, then you can increase the size from 1 to any other value).

When one of the arguments to the operator is a constant, there are two ways of handling it: 1) cast it to a 0D image and use the function above, or 2) write specific functions that do arithmetic and comparison with a constant. If option 1 is not much less efficient than option 2, it is to be desired (less code to maintain, smaller binaries, etc.)

The reason we propose one function for all arithmetic operations, rather than separate ones, is because that’s the way it’s done in the current *DIPlib* and it seems to work well... Maybe we should implement independent functions?

Several more complex arithmetic operations are rather common, and could be implemented as special functions. For example, instead of writing

out = a\*img1 + b\*img2 + c\*img3 + d;

one would write

dip::WeightedAddition( dip::ImageArray{img1,img2,img3}, out, dip::IntegerArray{a,b,c}, d );

For efficiency. There is template meta-programming that can convert an expression like the first one into a call like the second one, but that is too much templates for my taste. Is that really worth it? It doesn’t translate to other languages (Python, MATLAB, etc.).

We could try to complete the function that Mike started writing eons ago(?), in which a string expression is evaluated and applied pixel by pixel to a set of images:

dip::Evaluate( dip::ImageArray{img1,img2,img3}, out, "3\*a + 2\*(b+c) + 6 > 30\*(a+c)" );

The expression is converted to a representation can be used to efficiently apply the requested computation on each pixel. Which operators and functions to allow in there is open for discussion. In this example, letter a would refer to the first image in the input array, letter b to the second, etc.

## Class method vs function

Some libraries put all image processing/analysis functionality into the image object as methods. The idea is to filter an image by img.Gauss(sigma). This is a terrible idea for many reasons: it’s ugly, one never knows if the image object is modified by the method, and the core include file for the library changes when adding any type of functionality, forcing recompilation of the whole library. Filters should be functions, not methods.

Image object methods should be those that query image properties, set image properties, tweak dimensions, and so on. For example, we can have a IntegerRotation() method, which changes origin pointer, and strides and dimensions arrays, but doesn’t change the data, to rotate by a multiple of 90 degrees. Similar methods would be Mirror(), PermuteDimensions(), SwapDimensions(), TensorToScalar(), etc. (a better name for that last one? a method that makes tensor dimensions be image dimensions, for example converting a 2D RGB image into a 3D scalar image). All these methods affect the image object itself, they do not make a modified copy.

## Library initialisation and global variables

We should avoid having to call dip\_Initialise(), and we should avoid all globals. These make the library non-reentrant, which can be important for multi-threaded applications.

Global settings can be replaced by standard default values in functions. If you need to change the default, you just enter the right value every time you call the function. It’s not elegant, but it’s much more robust and it’s easier to see what will happen. Using defaults that can be changed by another thread is awkward.

The current *DIPlib* does a lot with registries because it is a closed-source library. The registries allow extension of functionality in a way that would otherwise not be possible. In an open-source library this is much less important, since users can modify and re-compile the library. For some things, such as the measurement function and the colour conversion function, registering new functions makes sense: it is reasonable to expect a user to want to extend existing functionality, and it might be beneficial to not modify the “standard” library, or submit modifications to the maintainers. For these cases, an object should be created (see below under measurement and colour conversion). Other things that we currently register are not necessary: e.g. the image type will never be extended, and if someone does it, it is a sufficiently significant change to warrant recompiling the library (as none of the existing functions would work with the new image type anyway). Do we want file reading and writing functions to be registered? If one writes a function to read files of a specific type, one can simply call that function directly.

## Parallelisation

dip\_FWClassical() and dip\_FWDoubleStripe() need to be parallelised anew, dip\_FWClassicalOMP() and dip\_FWDoubleStripeOMP() are overly complicated (and slow!) because they are written for pthreads. In short, we need to make sure that each thread allocates its own buffers (malloc inside the parallel section rather than before, as is now).

dip\_PixelTableArrayFrameWork() and dip\_PixelTableFrameWork() are currently not yet parallelised, but should. We should also be able to parallelise sorting functions, reductions (max, sum, etc.), measurements, and noise generation.

## Measurement

dip::MeasuringTool measuringTool;  
measuringTool.Register("myMsrName",myMsrFunc\_Info,myMsrFunc\_Prepare,myMsrFunc\_Measure,...);  
dip::Measurement msr = measuringTool.Measure(labimg,greyimg,{"myMsrName","size"});  
std::cout << msr["myMsrName"][1]; // How to do indexing?  
std::cout << msr.at("myMsrName",1); // How to do indexing?

Allocate instance of measurement tool class, call its Measure() method in much the same way we now call dip\_Measure(). The Register() method allows to register new measurement features, in just the same way we do now. The constructor calls Register() for each of the standard types. This should be fast.

Currently we use a function to get an ID for each of the measurement features. Instead, we always use strings to refer to measurement features. These are mapped to an index into a table that stores the function pointers. We can use std::unordered\_map, a hash table, to map the names to the indices.

The measurement functionality will be different than it is now. The current measurement structure is too flexible; this is not necessary, and causes undue difficulties in usage. Each measurement feature must register a set of functions, currently this is more than should be needed. This is how I envision the measurement functionality:

* The measurement data structure will only contain floats (currently: int, int array, float, float array), with a fixed number of them per object. These will be stored as a single array for each measurement. The objectID list that comes with it identifies which object each row is for. We can keep the hash table we’re using, but it just contains a row number for each object ID (for quick searching).
* Each measurement feature will register an Info function and a Measure function. Currently they also register a Value and a Convert function, these are useless. LINE\_BASED features have a Prepare, a Measure, and a Finish function. Prepare and Finish are called for each object. Measure is called for each line. IMAGE\_BASED, CHAINCODE\_BASED and CONVHULL\_BASED features have only a Measure function, called once (image-based feature) or once for each object (the other two). COMPOSITE features have a Measure function that has access to other measurement data (called after the other functions are done) and is called once for each object.
* The Info function returns the number of output values the measurement will generate (based on dimensionality, physical dimensions, etc.), as well as the number of intermediate values it will need for each object (only line-based functions get to store intermediate values). It also returns a description and unit strings to be used for each measurement (also dependent on image properties), which are stored in the measurement structure. At the same time, it checks image properties, and generates an error if the measurement cannot be made. Finally, it returns FeatureID values for composite measures.
* Measure functions receive the array they’re supposed to write in, as well as the row number and the object ID. They can use the physical dimensions directly in their computations; line-based functions can add physical dimensions at the end, in the Finish function.
* Access functions can still retrieve data for a single measurement and object, but there should be other functions that return the full array for a measurement. This will make the conversion to MATLAB easier.
* Convenience functions could be made to help with the units. For example:
  + Intensity(physdims,power) ADU^2
  + Spatial(dimension,physdims,power) px^2
  + SpatialCross(dimensionarray,physdims,power) m^2 s^2

## Colour space conversion

dip::ColourConverter colourConverter;  
colourConverter.Define("newspace",3); // "newspace" has 3 channels.  
colourConverter.Register(rgb2new,"rgb","newspace",cost); // cost is optional, default = 1  
colourConverter.Register(new2rgb,"newspace","rgb");   
colourConverter.Register(new2grey,"newspace","grey"); // cost is optional, default = 1e9  
colourConverter.Register(grey2new,"grey","newspace");   
colourConverter.Convert(img,img,"newspace"); // convert img to ‘newspace’, write output in img

The object constructor registers the standard colour conversion functions, the user can allocate new functions. These functions compute the conversion for a single pixel, and have the form

rgb2new( vector<double> in, vector<double> out, double whitepoint[9] )

The whitepoint matrix is given in column-major order. Instead of std::vector we might use a copy of the 1D array defined in the Boost library, which cannot change in size after it is allocated, and thus is a lot more efficient (allocated on stack, not in heap).

Colour space names as strings might be OK if we use a hash table to store colour space information. std::unordered\_map

The object stores a list of known colour spaces and known conversion routines in a graph structure. Colour spaces are nodes, conversion routines are edges. The cost associated to each edge is 1 by default, but can be increased for certain nodes if necessary. The cost can indicate computational cost, but also the loss of information. This is why conversion to grey scale has a very high cost by default.

A standard Dijkstra algorithm will find an optimal conversion path to convert an image from its current colour space to the destination colour space. This optimal path can then be cached in the object, if we see that this would save a significant amount of time. Because conversion to grey is so expensive, the optimal path will never be “rgb->grey->Lab” or something like that. **This has been implemented in *DIPimage*, to prove it works well.**

With the standard path as a list of function pointers, we call the point-scanning framework and for each pixel we call each of the functions in the path. For efficiency, these functions could iterate over a line, like all the other FrameWork callback functions do.

An image will carry its colour space name, and optionally a white point matrix. If no white point is present, the default white point is assumed.

## Function overloading

We want to use templates for function overloading (instead of generated code through multiple inclusion of the same C file). But we don’t want to expose any templates to the user: no knowledge of data types is assumed at compile time (see more on this below). The decision about which overloaded version of a function to call has to be made at run time. The easiest way we have come up with so far to accomplish this is by defining macros such as:

#define DIP\_OVL\_CALL\_REAL( fname, paramlist, dtype ) \  
 switch( dtype ) { \  
 case dip::DT::UINT8: fname <dip::uint8> paramlist; break; \  
 case dip::DT::UINT16: fname <dip::uint16> paramlist; break; \  
 case dip::DT::UINT32: fname <dip::uint32> paramlist; break; \  
 case dip::DT::SINT8: fname <dip::sint8> paramlist; break; \  
 case dip::DT::SINT16: fname <dip::sint16> paramlist; break; \  
 case dip::DT::SINT32: fname <dip::sint32> paramlist; break; \  
 case dip::DT::SFLOAT: fname <dip::sfloat> paramlist; break; \  
 case dip::DT::DFLOAT: fname <dip::dfloat> paramlist; break; \  
 default: throw("error message"); }

Similar macros would be defined for INTEGER, UNSIGNED, SIGNED, FLOAT, NON\_COMPLEX, NON\_BINARY, etc. The template would be used in this way:

template<typename TPI>  
static dip\_\_MyFunction( void\* vin ) {  
 TPI\* in = static\_cast<TPI\*> ( vin );  
 ...  
}  
void dip::dip\_MyFunction( dip::Image in ) {  
 dip::DataType dt = in.GetDataType();  
 DIP\_OVL\_CALL\_REAL( dip\_\_MyFunction, (in), dt );  
}

## Frameworks

DIPlib uses Frameworks to handle most of the filtering. These need some refactoring for consistency. Current code is written for POSIX threads, and adapted to OpenMP, but this lead to suboptimal code.

* dip::FrameworkFilterFull()
  + A framework that scans the image line by line, using the pixel table concept to do *n*D filter neighbourhoods.
  + Boundary extension accomplished by copying whole image. Data type of the image copy matches buffer type (or should we do this with a multi-dimensional buffer? depends on filter size?).
  + dip::FrameworkFilterFullSingle(): 1 input, 1 output
  + currently: dip\_PixelTableArrayFrameWork() and dip\_PixelTableFrameWork()
* dip::FrameworkFilter1D()
  + A framework that scans the image line by line. Image lines are copied to match buffer types and boundary extensions. Can work in-place.
  + Filtering function can be overloaded and takes a 1D vector of pixels, optionally with strides.
  + dip::FrameworkFilter1DSingle(): 1 input, 1 output
  + dip::FrameworkSeparableFilter(): repeated calling of dip::FrameworkFilter1D()
  + dip::FrameworkSeparableFilterSingle(): 1 input, 1 output
  + currently: dip\_SeparableFrameWork(), dip\_MonadicFrameWork() (as called by interpolation funcs)
  + If boundary extension is 0, and buffer types are same as image types, do dip::FrameworkScan()!
* dip::FrameworkScan()
  + A framework that scans the image, doing a point operation (line-by-line, or whole image as one long line). Does not use buffer types (maybe an output buffer?), can work in-place.
  + Filtering function must be overloaded and takes a 1D vector of pixels, with strides. dip::FrameworkFilter1D() can be used if buffers are needed (e.g. no overloaded function).
  + dip::FrameworkScanSingle(): 1 input, 1 output
  + dip::FrameworkScanSingleOutput(): 1 output
  + currently: dip\_MonadicFrameWork(), dip\_ScanFrameWork(), dip\_SingleOutputFrameWork()
  + option: ignore coordinates (DIP\_FRAMEWORK\_AS\_LINEAR\_ARRAY)

Input to the framework is:

* a pointer to a function that will process a 1D buffer
* an array of input images, each one with associated information such as buffer data type (the framework will use a buffer if the image’s data type does not match the requested buffer type)
* an array of output images, also with associated information
* flags stating whether the input can be overwritten (in-place operation), whether the function is multi-threading safe, the size of the boundary, etc.
* information about any additional temporary buffers that need to be allocated and passed to the function
* a pointer to a structure with parameters to pass directly to the callback function

An alternative would be to combine the function pointer and the structure with parameters to pass to this function into a functor object. This could add a lot of flexibility and of course avoids the use of pointers altogether.

FrameWorks in current DIPlib find the dimension with the largest size, when asked to use optimal scanning. The callback function loops over this dimension. This minimizes the number of callback calls. However, if the size difference is not too large, it might be better to use the dimension with the smallest stride. We’ll have to measure where the size ratio cutoffs are for this.

## Alias handler

Many of the current DIPlib functions (the ones that cannot work in-place) use a function ImagesSeparate() to create temporary images when output images are also input images. The resource handler takes care of moving the data blocks from the temporary images to the output images. We can do that this way in C++:

class AliasHandler {  
 private:  
 ImageRefArray output; // We save references to the original output images here  
 ImageArray temp; // We keep temporary images here  
 public:  
 AliasHandler( ImageRefArray in, ImageRefArray out ) {  
 for (int ii=0; ii<output.size(); ++ii) {  
 if ( out[ii] is in in ) { // How to do this check is up for discussion  
 output.push\_back( out[ii] );  
 temp.emplace\_back();  
 out[ii] = temp.back();  
 } } };  
 ~AliasHandler() { for (int ii=0; ii<output.size(); ++ii) { output[ii] = std::move(temp[ii]); } }  
};

void DoSomethingWithImages( Image in1, Image in2, Image out1, Image out2, Image out3 )  
{  
 ImageRefArray inar { in1, in2 };  
 ImageRefArray outar { out1, out2, out3 };  
 AliasHandler ah( inar, outar );  
 // outar[0], outar[1], and outar[2] are now safe to use.  
 outar.Strip(); outar.CopyDimensions( in1 ); outar.Forge(); // do processing ...  
} // ah goes out of scope, its destructor is called, and temporary images are copied to output

## Physical units

The image object should contain a physDims structure. This structure maybe should also contain the absolute position of the origin. Manipulation functions will update this (e.g. resampling changes physical dimensions of the pixel). For some functions this is difficult:

* Rotate: what if the two dimensions being intermingled have different physDims?
* The Fourier transform should set physDims[i] := size[i]/physDims[i] (or something like that).

Parameters to functions (filter sizes, etc.) should still be in pixels, but sometimes we want to give them in physical dimensions. We can solve this with a method of the physDims structure:

dip::FloatArray sigmas { 45, 45 }; // sigmas in physical units (say micron)  
dip::Gauss( in, out, in.physDims.topixels( sigmas ) ); // filter size parameter converted to pixels

The measurement function always returns measurements with physical units. If physical dimensions are not known, they default to 1 px, and the measurement function returns measurements in pixels. Operations between images with different physical dimensions leads to units to revert to 1 px. Scaling of an image with physical dimensions of 1 px does not affect its physical dimensions.

Tensor images have same intensity units for all tensor elements (or???). Intensity will most likely be “Arbitrary Units” (“ADU”), only in very special cases the intensity has physical units. Problem: ADU^2=ADU, ADU\*px=ADU, etc., but we don’t want that. Maybe we don’t even need to store the intensity units. This also saves us from having to modify them with every arithmetic operation.

To decide: How do we represent units? A class that knows how to convert inches to cm? Do we always use units in SI, and modify the value to account for kilo/mili/micro, etc.? In this case, we need to be able to automatically add prefixes to the units to make values readable.

## 3rd party libraries

We will use the C++ standard library for all maps, queues, vectors, sorting, etc., unless custom code is more efficient (measure if it really is the case!). One example could be creating our own version of std::vector for dip::IntegerArray and the like, that has a short vector optimization (these arrays we use mostly to do things per dimension, and usually images have only two or three dimensions; if the vector has e.g. four or fewer elements, this optimization will keep the data inside the object, avoiding a call to malloc; the standard library does this for string, but not for vector; copying these short arrays will then be trivial). It is a bit of work to write our own vector class, but I think it’s worth it. We will then also be able to add some methods that might be useful (product(), arithmetic operators, etc.)

We should use *FFTW* for the Fourier transform. But it’s GNU licensed, meaning that we won’t be able to include it without making the library GNU also? We might be able to include it as an “optional” element, let the user download *FFTW* for him/herself. It is not clear whether that breaks the GNU license terms. And what does the poor soul do that wants to create non-GNU software with this? An alternative could be *djbfft* (http://cr.yp.to/djbfft.html), which looks like it hasn’t been updated in a while (since 1999!), but is claimed by the author to be faster than *FFTW2* and any other library at that time. I’m sure this is good enough for us. There are no indications of a license at all…

*Tina’s Random Number Generator Library* for RNG? (does parallel RNG also). Update: apparently the C++11 standard library includes the concepts and ideas from this library.

We will keep the TIFF and ICS readers/writers we currently have, using *libtiff* and *libics*. For other file types, we could use *Bio-Formats*. How to link to Java from C++?

*Eigen* is a pretty sweet linear algebra library. It’s stable, very efficient, and templated like the standard library. We will be able to wrap a pixel (with its strides) as an *Eigen* matrix and do computations.

## Compile-time vs run-time pixel type identification

Currently, *DIPlib* uses run-time identification of an image’s pixel type, and functions dispatch internally to the appropriate sub-function. These sub-functions are generated at compile time through templates (C-style: #include!). We do not want to go away from this system. The alternative, seen in most C++ image analysis libraries (*ITK*, *Vigra*, *CImg*, etc.), is to define the image class as a template, as well as most functions. The user declares an image having a specific data type, and the compiler then creates an image class with that data type for the pixels, as well as instances of all functions called with this image as input. This takes time. Compiling even a trivial program that uses *CImg* takes a minute, rather than a fraction of a second it takes to compile a trivial program that uses *DIPlib* (I do not have experience compiling *Vigra* or *ITK* programs, but presumably these also slow down compilation significantly). Writing most functionality as templates implies that most code is actually in the header files, rather than in the source files. This functionality then ends up in the application executable, rather than in an independent library file (shareable among many applications).

However, the largest disadvantage happens when creating an (even slightly) general image analysis program: you need to write code that allows the user of your program to select the image data type, and write code that does all the right dispatching depending on the data type. Alternatively, you have to restrict the data type to one choice. A library is meant to take away work from the programmer using the library, so it is logical that *DIPlib* should allow all data types and do the dispatching as necessary. After all, *DIPlib* is meant as a foundation for *DIPimage* and similar general-purpose image analysis tools, where you cannot determine in advance which data type the user will want to use. Think about how complicated each of the *DIPlib*-MEX-files would be if *DIPlib* had compile-time typing: each MEX-file would have to check the data type of the input array (or arrays), convert it to a *DIPlib* image class of the same type, then call one of 8 or 10 instances of a function. Furthermore, this MEX-file would need to know which image data types are meaningful for the function being called (integer only? binary only? does it work on complex values?). Instead, we simply convert the input array to a *DIPlib* image and call a function. The MEX-file is trivial, the *DIPlib* function itself takes care of everything.

*OpenCV*, as far as I have been able to tell, uses a template for the image class, but has another image class (virtual base class?) that the programmer typically uses. Thus, functions are compiled in the library, and do dispatching internally, as *DIPlib* does (not sure if they use RTTI or some other mechanism). However, I have seen functions explicitly defined for only a small subset of types (typically only uint8, uint16, sint16, or something like that). In *DIPlib* we aim at having as broad a type support as makes sense.

I’d like *DIPlib* to expose as few templates to the user (programmer) as possible.

## Placement of output image argument in function calls

Basically, there are two options:

#1: void dip::Gauss( dip::Image in, dip::image out, params );

#2: dip::Image dip::Gauss( dip::Image in, params );

The current *DIPlib* uses #1 (but returning an error code). Both have advantages and disadvantages. With #1 you can do in-place operations simply by giving the same image as input and output; you can also force the data type of the output by setting the “don’t adjust” flag as described somewhere near the top of the document. For external interfaces, which want to control the allocation of data blocks by assigning a forge handler to an image, #1 is the only choice. But with #2 you can chain function calls without having to define intermediate variables:

dip::Gauss( dip::Gauss( in, params1 ), params2 );

I very much like #2, as it makes programs easier to read (and more similar to corresponding *DIPimage* scripts), but we cannot do without the advantages of #1. One viable alternative is this:

void dip::Gauss( dip::Image in, dip::image out, params );  
inline dip::Image dip::Gauss( dip::Image in, params ) {  
 dip::Image out;  
 dip::Gauss( in, out, params );  
 return out;  
}

This allows both forms for the same function.

## Passing options to a function

The MMorph library uses strings instead of #defined constants to pass options to functions: dip::Fourier( in, out, "forward" ) instead of dip::Fourier( in, out, dip::ft\_dir::FORWARD ). This has two advantages: it is easier to generate interfaces to MATLAB and Python (less code in the interface), and there are fewer enum definitions in the header files, and thus fewer possibilities for name clashes. The disadvantage is that the function needs to do string comparisons rather than integer comparisons when parsing input arguments. I don’t really think this adds much of an overhead.

In short, as much of the parameter parsing that we currently do in *DIPimage* should be done in *DIPlib*, so that the C++ program looks more like the current MATLAB scripts.

Currently, there are quite a few functions that use a bitfield as an input argument. The bits are defined by an enum, but their combination (through the | operator) is not part of the enum, and consequently fires a compiler warning. We can do three things here: use an actual bitfield class (are & and | operators applicable?), use a compound string, or use a string array:

#1: struct param { int a:1, int b:1 };

#2: "a|b"

#3: dip::StringArray {"a", "b"}

I like #3, as it makes the MATLAB and Python interfaces as simple as possible, and the MATLAB/Python syntax as similar to the C++ syntax as possible. #2 does this too, but is a little bit more strange in use, and definitely will be more expensive to parse by the C++ function. For internal, low-level functions that are not exposed to the regular user nor to MATLAB and Python interfaces, we can keep using the more efficient enum method (e.g. for parameters to the framework functions).

## Functionality currently not in DIPlib that would be important to include

* An overlay function that adds a binary or labelled image on top of a grey-value or colour image.
* Stain unmixing for bright-field microscopy
* Some form of image display for development and debugging. We can have the users resort to third-party libraries or saving intermediate images to file, or we can try to copy *OpenCV*’s image display into *dipIO*.
* Some filters that are trivial to add:
  + Scharr (slightly better than Sobel)
  + h-minima & h-maxima
  + opening by reconstruction
  + alternating sequential open-close filter (3 versions: with structural opening, opening by reconstruction, and area opening)
* dilation/erosion by a rotated line is currently implemented by first skewing the image, applying filter along rows or columns, then skewing back. We can add a 2D-specific version that operates directly over rotated lines. The diamond structuring element can then be decomposed into two of these operations. We can also add approximations of the circle with such lines.
* Most of the functionality that is now implemented in *DIPimage* only:
  + automatic threshold determination (Otsu, triangle, background, etc.)
  + colour space conversion (discussed above)
  + 2D snakes
  + look-up tables (LUT, both for grey-scale and colour LUTs, using interpolation when input image is float)
  + general 2D affine transformation, 3D rotation
  + xx, yy, zz, rr, phiphi, ramp; extend this to coords(), which makes a tensor image.
* Radon transform for lines and circles, Hough transform for lines
* Level-set segmentation, graph-cut segmentation
* The Label() function should return the number of labels. It could optionally also return the sizes of the objects, since these are counted anyway.
* We need to figure out if it is worth it to use loop unrolling for some basic operations.

## Python interface

We will define Python class as a thin layer over the dip::Image C++ object. Allocating, indexing, etc. etc. through calls to C++. We’d create a method to convert to and from *NumPy* arrays, simply passing the pointer to the data. We’d have to make sure that deallocation occurs in the right place (data ownership). A *DIPlib* function to extract a 2D slice ready for display would be needed also, make display super-fast!

## MATLAB interface

The MATLAB toolbox will be significantly simplified:

* No need for the libdml.so / libdml.dll library, as there are no globals to store, and not much code to compile.
* Basically, all the code from that library will sit in a single header file to be included in each of the MEX-files:
  + Casting an mxArray input to: Image, IntegerArray, FloatArray, String, sint, uint, or double.
  + Casting those types to an mxArray output.
* We won’t need to convert strings to enums, as that will be done in the *DIPlib* library (see above).
* Some MEX-files will have more elaborate data conversion, as is the case now (e.g. dip\_measure).
* The few functions in the current *DIPlib* that use an ImageArray as input or output will use tensor images instead.

The M-file code will need to be adapted:

* The dip\_image object will not also double as dip\_image\_array (that was a mistake). Instead, tensor images will be a single data block, as they will be in *DIPlib*. Thus, some of the class methods will have to be rewritten, and some functions that use image arrays will have to be adapted. For an array of images, use a cell array.
* Much of the code for the dip\_measurement object will change also, as its representation will likely change with the changes in *DIPlib*.
* The colour conversion code will be replaced with a single call to *DIPlib*, as will some other functionality that will be translated from MATALB to C++. Much of the tensor arithmetic should be done through *DIPlib* also.
* dipshow will be simplified, as simple calls to a *DIPlib* display function will generate the 2D array for display.