Advanced Object Oriented Programming Mandatory Assignment 2

Magnus Erik Hvass Pedersen
University of Aarhus, Student #971055
December 2005

1 Introduction

The purpose of this document is to verify attendance of the author to the *Advanced Object Oriented Programming* course, at the Department of Computer Science, University of Aarhus.

1.1 Aim

This work was originally intended as a study of how to convert certain types of audio processing filters, to make them able to process video also. The overall idea of doing this, is to treat each pixel in the video stream, as if it was an audio signal, and therefore have one processing filter, for each pixel in the image. To do this simply and efficiently, a style of programming known as *meta-programming* is being used, and as this is a fairly new addition to the C++ programming language, much research was needed, mostly from unorthodox sources such as webpages, newsgroups, etc. Much of this research is documented here.

1.2 Overview

As the project developed, new methods of cleverly applying meta-programming were discovered, and emphasis is therefore also put on these discoveries. The document is structured as follows:

- Section 2 is a brief introduction to *Digital Signal Processing*. The introduction to section 2 as well as section 2.2, are both recommended reading; in particular the recurrence relation for a socalled lowpass filter (Eq.(2), page 3).
- Section 3 describes an implementation of the filters from the previous section. This implementation is in C++ and uses meta-programming. Unfortunately though, it does not work in all cases, and a much simpler, yet specialized version of the filter, implements the lowpass filter in a manner that works for images also. This is found in section 3.3 (page 13); which is recommended reading along with section 3.4 on page 14.
- Section 4 describes various ways of implementing classes in C++ for storing and manipulating numeric vectors. The major contribution is found

in section 4.10 (page 37), where a new and much simplified way of using meta-programming for manipulating such vector-classes is discovered. That section is naturally recommended reading.

Testing is conducted in section 5 on page 47, demonstrating the correctness and capabilities of the various implemenations. And section 6 concludes the report and briefly summarizes the object-oriented techniques, that can be combined to implement a very simple and effective numeric vector processing library.

2 Digital Signal Processing

The objective of *Digital Signal Processing* (DSP), is often to create an algorithm that takes some input x, and transforms it into some output y, where both input and output change over time. To model this in a discrete context, we shall denote by $x[n]$ the n'th element of the sequence x.[1] Typically, the input and output are real-valued and limited to the range $[-1, 1]$, but other domains and ranges are also possible; indeed, we shall work with N-dimensional domains, where N is the total number of pixels in an image.

2.1 IIR Filter

By DSP filter, is often meant an algorithm that either enhances or attenuates some of the frequency components in an input sequence x. For example, in sound-processing, we may wish to attenuate the treble and boost the bass of a certain sound, or vice versa.

A certain type of filter, is known as an *Infinite Impulse Response* (IIR) filter, because each input $x[n]$ will affect the output of the filter indefinitely.[2] An IIR filter is based on a weighted sum of the current and previous l input samples, as well as the previous m output samples. The basic formula is therefore:

$$y[n] = a_0 \cdot x[n] + \sum_{i=1}^{l} a_i \cdot x[n-i] + \sum_{i=1}^{m} b_i \cdot y[n-i] \tag{1}$$

Where a_i and b_i are called the filter coefficients, and depend on what kind of filter we desire (e.g. low- or high-pass, etc.), as well as the characteristics of that filter (activation frequency, slope, etc.). Also note that we may generally assume $l \geq 0$ and $m \geq 1$ in any IIR filter, so we always use at least one input sample (the current one), and one output sample – otherwise the filter would simply not be classified as an IIR filter.

2.2 Lowpass Filter

The filter coefficients for a socalled *one-pole* lowpass filter – a filter which only lets low-frequency content pass through the filter, and that has a soft attenuation

[1] Some texts denote the sequence x as $x[\cdot]$ or even $x[n]$, but we shall refrain from this, and denote the entire sequence simply by x.

[2] That is, for the usual and non-extreme settings of filter parameters.

of higher frequencies – are as follows (see e.g. [1, Chapter 19]):[3]

$$\begin{aligned} a_0 &= 1 - c \\ b_1 &= c \end{aligned}$$

Where $c \in [0, 1]$ is the cutoff-constant to be set by the user, with $c = 0$ meaning the filter is entirely open, and all content passes of the input signal passes through, so that $y = x$. At the other extreme $c = 1$ means the filter is fully closed, and nothing passes through, so that $y = 0$.[4]

Inserting the cutoff-constant c back into Eq.(1) instead of the coefficients a_0 and b_1, and simplifying the general IIR filter formula, we find that a one-pole lowpass IIR filter, at each step merely performs linear interpolation between the current input and the previous output:

$$y_l[n] = (1 - c) \cdot x[n] + c \cdot y_l[n - 1] \tag{2}$$

This explicit equation may therefore be used for a simplified implementation of the lowpass filter (see section 3.3).

2.3 Highpass Filter

To make a high-pass IIR filter, we only have to change the coefficients to the following:

$$\begin{aligned} a_0 &= (1 + c)/2 \\ a_1 &= -(1 + c)/2 \\ b_1 &= c \end{aligned}$$

Again with $c \in [0, 1]$ being the cutoff-constant set by the user.

2.4 Bandpass Filter

In [1, Chapter 19] we also find the coefficients for socalled narrow-band filters. For example, to make a socalled bandpass filter, that only lets through the frequency components around the center frequency f, we have the following formulae for finding the IIR filter coefficients:

$$\begin{aligned} a_0 &= 1 - K \\ a_1 &= 2(K - R)\cos(2\pi f) \\ a_2 &= R^2 - K \\ b_1 &= 2R\cos(2\pi f) \\ b_2 &= -R^2 \end{aligned}$$

[3] All other coeffiecients are zero.

[4] More specifically, when $c \equiv 1$ is fixed, then $y[n] = y[0]$ for all n. So for $y[n]$ to be zero, would require $y[0]$ to be zero.

Where R and K are defined as follows:

$$\begin{aligned} R &= 1 - 3b \\ K &= \frac{1 - 2R\cos(2\pi f) + R^2}{2 - 2\cos(2\pi f)} \end{aligned}$$

With b being the bandwidth or slope of the filter response. Both the center frequency f and the bandwidth b are expressed as fractions of the sampling rate, meaning $f, b \in [0, 1/2]$.

2.5 Bandreject Filter

Using the constants R and K from above, we can make an IIR filter that instead suppresses components of the input signal near the center frequency f, by having the following filter coefficients:

$$\begin{aligned} a_0 &= K \\ a_1 &= -2K\cos(2\pi f) \\ a_2 &= K \\ b_1 &= 2R\cos(2\pi f) \\ b_2 &= -R^2 \end{aligned}$$

Deriving the coefficients for the various filter types, is beyond the scope of this document, and the reader is once more referred to a more general text on DSP, such as [1].

3 IIR Filters In C++

The basic implementation of a general IIR filter, as a class in the C++ programming language, would use hardcoded datatypes for its input, output, and filter coefficients, as well as using `for`-loops in the summations of Eq.(1). Since we may use the IIR filters for processing different kinds of data, for example sound as well as images, we would like the ability to change the datatypes in a class, without having to rewrite the class for each new datatype. Furthermore, since we usually only require a few input and output samples in the summations in Eq.(1), we would like to avoid the costly overhead of having `for`-loops, and instead *flatten* those loops in some way.

3.1 Template Datatypes

Providing arbitrary datatypes in a C++ class or function, is done by way of socalled template arguments. Then, upon instantiation of such a class (or function), one provides the actual datatype to be used.

An important aspect of template arguments, is that they must be known at compile-time, so the execution code specialized for a given datatype, can already be generated during compilation of the source-code.

3.1.1 Template Meta-Programming

Using this aspect of template programming, was in recent years found to facilitate something called *meta*-programming, in which we write template-classes and -functions, that specify how to make specialized data-structures and/or code, that are then created automatically by the compiler at compile-time, hence avoiding any potentially costly run-time overhead.

In the following, we will see many examples of meta-programming, all of which aim to provide simple notation, or reuse base-classes, without the usual degrading of execution speed, associated with common object-oriented hierarchies using `virtual`-functions for specialization.

3.2 IIR Filter Base-Class

The template class for a general IIR filter, must define the template datatypes for input and output, as well as the number of such samples to be used. Presently ignoring the functions of the class, we have:

```
template <class T, typename U, int kNumY, int kNumX>
class LFilterIIR
{
public:
    LFilterIIR (T const& init) : mX(init), mY(init) {}

    // ...

protected:
    LCircularBuffer<T, kNumX-1>  mX;   // Previous input.
    LCircularBuffer<T, kNumY>    mY;   // Previous output.

    U mA[kNumX];               // Coefficients for input.
    U mB[kNumY];               // Coefficients for output.
}
```

Where `T` is the datatype for input and output, `U` is the datatype for the coefficients, and `kNumX` and `kNumY` are the number of input and output samples, respectively.

Note that the previous input and output are initialized with an object passed (by reference) to the constructor of the `LFilterIIR`-class. The reason for this, is that we will eventually wind up using a datatype for holding arrays, which requires such initialization. For scalar-values (e.g. when the filter is to be used for audio signals), we would just supply a value of zero.

3.2.1 IIR Equation

In the `LFilterIIR`-class, we override the `operator()` function to take a single input sample, and return a reference to a single output sample:

```
template <class Exp>
inline T const& operator() (Exp const& x)
{
    return mY.push_peek( GetA0() * mX.push_peek(x)
                    + DotProductFlat<kNumX-1, T>(mA, mX)
                    + DotProductFlat<kNumY,   T>(mB, mY) );
}
```

There are many things to note here. First is the use of the function `push_peek()` from the `LCircularBuffer`-class. This function essentially stores a value in the buffer, and returns a reference to it. Next is the use of the `DotProductFlat()` function, which flattens the computation of the dot-product of its arguments, instead of using a loop. This function will be discussed in more detail below.

Another most important thing, is that instead of passing the current input sample `x` as a reference to an object of type `T`, it is sometimes convenient to pass a value of unknown type. For example, when we use arrays for datatype `T`, it may be appropriate to pre-process an array before passing it to the filter, but at the same time, we would probably want this pre-processing to be *merged* with the filter's own processing, to increase performance.

In such cases we shall pass an object holding an abstract representation of the pre-processing, and this is actually passed all the way to `push_peek()`, at which point the expression is evaluated upon assignment to the circular buffer's internal storage. Furthermore, when using arrays implemented such as the ones in section 4 (e.g. section 4.15 in particular), the expression:

```
GetA0() * mX.push_peek(x)
    + DotProductFlat<kNumX-1, T>(mA, mX)
    + DotProductFlat<kNumY,   T>(mB, mY)
```

will itself result in another such abstract representation, which will get passed to `mY.push_peek()`, and also first evaluated upon the call of the assignment-operator inside that function. This all probably sounds very strange, but should become clearer in section 4.

3.2.2 Retrieving Filter Coefficients

The `LFilterIIR` base-class provides a number of functions for setting and getting the coefficients. Actually, we only have a single function for getting the filter coefficient a_0 for the current input sample x, as the other coefficients a_i are used in a dot-product computation. The function returning a_0 is as follows:

```
inline U const& GetA0 () const { return mA[kNumX-1]; }
```

Note the return of a `const`-reference, which means a reference to the actual storage in `mA[kNumX-1]` is returned, but that we are not allowed to modify the data.

But why return a reference, are these coefficients not just scalar values? Well, usually they are, but we can easily imagine scenarios in which we filter

samples that are themselves arrays of data (such as images), and where it would be useful to have different coefficients for each element of those arrays. In such a case, the coefficients will also have to be arrays, and returning a copy of an array is rather expensive – particularly so when it is not needed.

3.2.3 Setting Filter Coefficients For Output Samples

Setting the filter-coefficients is done in the sub-classes of `LFilterIIR`. In some contexts, it is necessary to set the coefficients for every input sample. This is for example the case in an audio synthesizer, where the filter parameters are to be changed over time, and in order to do this smoothly, one usually does it for every sample. So the functions for setting the filter coefficients must therefore not incur any additional overhead, such as `if`-statements, additional index-arithmetics, and so forth.

Now, we would like to use mathematical indexing of the filter coefficients b_i, meaning that i goes from 1 to m, instead of the C and C++ style of array-indexing, which goes from 0 to $m-1$. This means we must map the index i in the $\{1, \cdots, m\}$ range, to $i-1$. To make it easier for the compiler to do this mapping at compile-time, and hence save a subtraction operation at run-time, we provide the index as a template-argument as follows:

```
template <unsigned int i>
inline void SetB (U const& b)
{
    assert(i<=kNumY);
    mB[i-1] = b;
}

template <>
inline void SetB<0> (U const& b) { throw; } // Error!
```

Note that we have specialized the function for the case where the template-argument is zero; in which case an exception is thrown, as it would be an error, because there is no coefficient b_0. Also note, that when the index is non-zero, an assertion will ensure the index is less than or equal to `kNumY`, which is the number of previous output samples, used by that particular instance of the IIR filter.

3.2.4 Setting Filter Coefficients For Input Samples

We have similar functions for setting the filter coefficients a_i for the input samples. At first, the functions may look awkward, but there are very good reasons why they are made that way.

Since the current input sample is not yet stored in the circular buffer `mX` of `LFilterIIR` (that would be a waste of space), we have to separate the computation of $a_0 \cdot x[n]$ from the rest of the dot-product, as illustrated in Eq.(1). Now, if we were to only use the current input sample $x[n]$ in the IIR filter (this is the

case for the lowpass filter), then we only need one coefficient a_0. However, if we also separated the actual storage of the coefficient a_0 from the rest of the a_i's, then with `kNumX` being one and `mA` then being an array of size tt kNumX-1, we would have to specialize the `LFilterIIR`-class, as we are not allowed to have C/C++ arrays of size zero.

This would be rather tedious, and the solution is of course to store a_0 with the rest of the a_i's, so the `mA`-array always has at least one element. But at the same time, we would like to address a_1, a_2, and so on, in the same manner as we did b_1, b_2, etc., then we must store a_0 at the end of the `mA`-array, instead of at the beginning, as would have been most natural. Hence, the functions for setting the a_i coefficients are as follows:

```
template <unsigned int i>
inline void SetA (U const& a)
{
    assert(i<kNumX);
    mA[i-1] = a;
}

template <>
inline void SetA<0> (U const& a)
{
    mA[kNumX-1] = a;
}
```

3.2.5 Circular Buffer

The input and output samples in an IIR filter are stored in circular buffers. These buffers allow us to always index the lastly inserted elements, as if they were the first elements. In other words, a circular buffer is just an array, with an index-offset and some arithmetic code to adjust this offset as new elements are inserted. The class is defined as follows:

```
template <class T, int kSize>
class LCircularBuffer
{
public:
    LCircularBuffer (T const& init) : mIndex(0)
    {
        for (int i=0; i<kSize; i++)
        {
            mBuffer[i] = init;
        }
    }

    // ...
```

```
protected:
    T               mBuffer[kSize];  // Storage.

    unsigned int    mIndex; // Index into mBuffer for
                            // lastly inserted element.
};
```

Where the buffer holds objects of type `T`, and the buffer is of size `kSize`; which must be known at compile-time. The integer `mIndex` is the index for the lastly inserted element, so the function in `LCircularBuffer` that returns this lastly inserted element, is simply:

```
inline T const& top () const
{
    return mBuffer[mIndex];
}
```

Inserting an element into the circular buffer, means we must first adjust `mIndex` (remembering it can not be less than zero and no higher than `kSize-1`), before assigning the element to the buffer's internal storage. The function is as follows, where you should not that the element is passed as an arbitrary expression (see the discussion in section 3.2.1):

```
template <class Exp>
inline void push_back (Exp const& elm)
{
    // Update index (move back).
    --mIndex;
    mIndex += kSize;
    mIndex %= kSize;

    // Update element at that index.
    mBuffer[mIndex] = elm;
}
```

Also note the index is decremented, which means it is rather simple to lookup elements offset by `mIndex`:

```
inline T const& operator[] (unsigned int i) const
{
    assert(i<kSize);

    return mBuffer[(mIndex + i) % kSize];
}
```

It is sometimes convenient to combine the `push_back()` and `top()` functions into a single function `push_peek()`:

```
template <class Exp>
inline T const& push_peek (Exp const& elm)
{
    push_back(elm);
    return top();
}
```

3.2.6 Empty Circular Buffer

We will sometimes need to instantiate circular buffers that are empty; for example in case of the input samples for the lowpass filter. However in `LCircularBuffer` above, we made an allocation of the storage as `mBuffer[kSize]`, which is of course illegal when `kSize` is zero. Since `kSize` is a template argument, the solution is to make a template specialization when this occurs, and have the accessing functions throw an exception:

```
template <class T>
class LCircularBuffer<T, 0>
{
public:
    LCircularBuffer<T, 0> (T const& init)
    { /* OK, we may init an empty buffer */ }

    template <class Exp>
    inline void push_back (Exp const& elm)
    { /* OK, no access is made */ }

    template <class Exp>
    inline Exp const& push_peek (Exp const& elm) { return elm; }

    // Throw an exception on all access to buffer.
    inline T const& top () const { throw; }
    inline T const& operator[] (unsigned int i) const { throw; }
};
```

You may argue that `push_peek()` should have thrown an exception also, but the way it is being used in `LFilterIIR`, requires for it to merely be the identity-function, whenever the circular buffer is actually empty.

3.2.7 Circular Buffer Initialization

Note the intialization of the buffer's objects in section 3.2.5. In a more general context, one should instead pass a function object to the constructor of `LCircularBuffer`,[5] and implement a default class for this, which simply initializes its argument to zero. When having datatypes that are really arrays or

[5] See section 4.9 for a description of function objects.

vectors (e.g. for storing images), one could then make another such initializer-class, and bind one of its parameters to the vector-size, and then have the vector-objects allocate memory accordingly. Then `LCircularBuffer` would be completely transparent in its initialization, and could be used for scalar datatypes as well as vector datatypes, without the need to pass an actual instance of a vector.[6]

3.2.8 Dot-Product

As described in section 3.2.1, the generic implementation of an IIR filter, makes use of the function `DotProductFlat()`, which is supposed to flatten the computation of the dot-product of the filter's previous input (or output) and the filter's coefficients.

Now, the obvious way to do this, would be to implement a template function, something like this:

```
template <class T, class U, class V, unsigned int i> inline
T DotProductFlat(U const& a, V const& b)
{
    return a[i-1]*b[i-1] + DotProductFlat<T, U, V, i-1>(a, b);
}
```

And then have a termination case:

```
template <class T, class U, class V> inline
T DotProductFlat<0>(U const& a, V const& b)
{
    return 0;
}
```

But this is not legal in C++. What we need to do, is to implement the functional parts inside a class, and have template arguments for both the functions as well as the class:

```
template <class T, class U, class V>
class LDotProductFlat
{
public:
    template <unsigned int i>
    static inline T Do (U const& a, V const& b)
    {
        return a[i-1] * b[i-1] + Do<i-1>(a, b);
    }
```

[6] This may sound meaningless now, but in a high-performance array- or vector-library, the assignment operator should not first allocate storage; therefore we would need a function object, to ensure a proper allocation-function in the array-class, was called during initialization of the objects in the circular buffer. However, the blitz++ library described in section 4.15, does allocate storage in the operator assignment; whenever needed.

```
    template <>
    static inline T Do<0> (U const& a, V const& b)
    {
        return 0;
    }
};
```

Note that we have to declare the functions `static`, so we can call them without instantiating an actual object from the class; nothing is needed from the surrounding class-definition, apart from the template-arguments. Also, C++ is not able to automatically deduce the template arguments for a class, but only for a function; so we make a wrapper-function for the `LDotProductFlat`-class, which is the function that the user should call (as done in section 3.2.1):

```
template <unsigned int i, class T, class U, class V> inline
T DotProductFlat(U const& a, V const& b)
{
    return LDotProductFlat<T, U, V>::Do<i>(a, b);
}
```

Here, the template arguments `U` and `V` can be automatically deduced, which leaves us to just specify the number of elements `i` in the input `a` and `b`, and the return-type `T`.

However, this is not exactly what we want, and in fact, it does not even work when the type `T` is an array-class from the blitz++ library. Oddly enough though, it does work when `T` is `LVectorMini` from section 4.5 below. It really should have been vice versa, because when the return-type `T` is some array-class, a temporary instance of that class would have to be created for each call to the `Do()`-function (more on temporary objects later). Since the implementation of `LVectorMini` does not allow for the creation of temporaries (not documented in section 4.5 though), but blitz++ does, the discrepancy remains a mystery.

The way to implement the set of `LDotProductFlat::Do()` functions, would be to have them return an `Expression`-object, as the ones used in the arithmetic meta-programming from section 4.10. This would make the flattening work as intended, without the creation of any temporaries, and without any great mystery as to why it works.

3.2.9 Lowpass Filter

Implementing the lowpass filter is then a fairly trivial matter, where we merely have to specialize the `LFilterIIR`-class. Again the datatype of the input and output samples is `T`, and the datatype for the filter coefficients is `U`. The lowpass filter is then implemented as follows:

```
template <class T, typename U>
class LFilterIIR_LP1 : public LFilterIIR<T, U, 1, 1>
{
```

```
public:
    LFilterIIR_LP1 (T& init) :
        LFilterIIR<T, U, 1, 1>(init) {}

    template <class Exp>
    void SetParameters (Exp const& c)
    {
        SetA<0>(1-c);
        SetB<1>(c);
    }
}
```

Which is really just an instance of `LFilterIIR<T, U, 1, 1>`, with the additional `SetParameters()`-function, for setting the filter coefficients given a cutoff value $c \in [0, 1]$.[7]

3.3 Hardcoded Lowpass Filter

For the datatype `T` we will ultimately be using arrays from the blitz++ library (see section 4.15), and because this does library not work with the computation of the dot-product from section 3.2.8, we shall instead make a hard-coded version of the lowpass filter. This will also serve as a gauge, whether the effort of making `LFilterIIR` such a general implementation of an IIR filter, was reall worth it. The hard-coded lowpass filter for arbitrary datatypes `T` and `U`, is simply:

```
template <class T, typename U>
class LFilterIIR_LP1_HC
{
public:
    LFilterIIR_LP1_HC (T& init) : mY(init) {}

    template <class Exp>
    inline T const& operator() (Exp const& x)
    {
        return mY = mA * x + mB * mY;
    }

    // Set the cutoff of the filter, c is [0,1].
    template <class Exp>
    inline void SetParameters (Exp const& c)
    {
        mA = 1-c;
        mB = c;
    }
```

[7] The `Exp`-typed parameter for the `SetParameters()`-function, should probably have been `U`-typed instead, to ensure no temporary objects are created. Otherwise `SetA()` and `SetB()` should have had their arguments as arbitrary temporary types also.

```
protected:
    T mY;       // Previous output.
    U mA, mB;   // Coefficients.
};
```

Clearly, we will have to reuse `LFilterIIR` quite a few times, to make up for its significantly more complex implementation.

3.4 Image Storage

We are interested in processing sequences of images using the above filters. To do this, we must implement a data-structure for storing an image, that supports the arithmetic operations used by the filters; namely addition and multiplication.

Each image is akin to a matrix whose entries are pixels, each pixel consisting of one or more colors, and possibly a transparancy- or opacity-measure. This is equivalent to having several matrices for a single image, one matrix for each color and one matrix for the image's opacity (which is sometimes called the alpha-layer).

However, a matrix may just as well be stored in a single array, and if necessary, functions can be provided that map row and column indices to and from the equivalent array index. Now we just need to find a way to implement a class for storing vectors of numerically typed elements, that efficiently supports the required arithmetic operations. This is the subject of section 4 next.

4 Numeric Vector

This section investigates different ways of providing numeric vectors in C++, and discusses the pro's and con's of the different ways. We end up with a fairly graceful solution in section 4.10, but as it still requires a somewhat substantial development effort, for the project at hand, we will ultimately settle for a commonly available library known as blitz++. But let us start at the beginning, and how C++ originally intended to supply numeric vectors.

4.1 STL's `valarray` Class

The *Standard Template Library* (STL) provides a template class called *valarray* for representing and manipulating numeric vectors [2, Section 22.4]. It is defined within the namespace `std` as follows:

```
template<class T> class valarray { /* ... */ }
```

where the template argument `T` designates the datatype of its elements. The class was originally intended to provide not only *syntactic sugar*, but also high-performance computing, whose implementation was tailored for the individual platforms on which C++ was implemented. Personally, I have yet to see anyone use it, and there is a good reason for this, as we will see shortly.

4.1.1 Arithmetic Operators

The `valarray`-implementation allows us to perform arithmetic operations such as the following:

```
const int n = 10;                          // Vector-size.
std::valarray<double> A(n), B(n), C(n);    // The vectors.

// Initialize values of vectors ...

A = B + C;                                 // A[i] = B[i] + C[i].
```

Although this kind of notation is exactly what we are looking for in the project at hand, `valarray` achieves support for it by overloading the arithmetic operators in the usual manner facilitated by C++, which has some unfortunate semantic implications.

4.1.2 Temporary Object Construction

First consider the following function interfaces, for addition of `valarray`-objects, as well as scalar values of type `T`:

```
template<class T>
valarray<T> operator+(valarray<T> const&, valarray<T> const&);

template<class T>
valarray<T> operator+(valarray<T> const&, T const&);

template<class T>
valarray<T> operator+(T const&, valarray<T> const&);
```

Notice two things in these operator overloadings. First that the functions' arguments are all references, and as discussed above, this means we avoid copying the parameters to the functions, which is good in case of the `valarray<T>` parameters, which take time copying; or if instances of the datatype `T` itself, are also large and hence take time copying.

Second however, notice that the return value is `valarray<T>`, but *not* a reference, thus implying a `valarray<T>` object would have to be created by each of the three functions, and then returned to wherever the function was called from. This is only natural, as the resulting vector must of course be stored somewhere, but it is clearly undesirable, because even more involved expressions, such as:

```
A = q * B + r * C + s * D;
```

would require the creation of a substantial number of such temporary objects, even when `q`, `r`, and `s` are scalar-values of type `T`. A very clever compiler could remove this overhead during optimization of the code, by `merging` the various loops into one, but that is quite unlikely, and we should certainly not count on it.

4.1.3 Inplace Arithmetic Operators

n easy way of avoiding the implicit and temporary creation of objects in general, is simply to use inplace arithmetic operators, which accumulate to the object on which the operator is invoked, instead of creating a new object. Consider the following functions of the `valarray` class:

```
template<class T>
class valarray
{
public:
    // ...

    valarray& operator+=(valarray<T> const&);
    valarray& operator*=(valarray<T> const&);
}
```

The implementation for both of these functions, should just accumulate to the object on which the function is invoked, and afterwards return a reference to that object. Inplace arithmetics have no need for temporary objects to be created, because the intermediate results are not held back before being assigned to their ultimate destination. But does this really work for general expressions? The answer is no; for anything but the simplest expressions, we will now need to manually employ a temporary vector, as demonstrated below.

Moreover, writing mathematical expressions using only inplace arithmetics is tedious and low-level, and rapidly becomes difficult to maintain. For a simple example, take the expression `A = B + C;` which would be:

```
A = B;
A += C;
```

You can imagine what this becomes like when the expression grows. Just take the expression from section 4.1.2:

```
A = q * B + r * C + s * D;
```

Which may be rewritten for inplace arithmetics, and then also requires the use of an auxiliary vector `V`, serving as temporary storage:

```
A  = B;
A *= q;       // A  = q * B;

V  = C;
V *= r;
A += V;       // A += r * C;

V  = D;
V *= s;
A += V;       // A += s * D;
```

So we have clearly lost our syntactic advantage, and have not even alleviated the need for temporary vector-storage, but have merely shifted its handling from being implicitly handled by the compiler, to now being explicitly controlled by the programmer.

4.1.4 Cache Locality

Another problem with the `valarray` approach, be it inplace or not, has to do with socalled *memory cache locality*. If the vectors (including the temporary objects) are very large and can not all be held in the memory cache at the same time, then some of their data will have to be *spilled* to memory, and in the worst case, we will never reuse data from cache, which is a very costly affair known as *thrashing*.

This problem arises from the fact that `valarray` – as well as similar implementations that work by direct operator-overloading – performs arithmetic operations on one pair of vectors at a time, and thus has to go through these two vectors in their entirety, before proceeding to the next operation with another pair of vectors (for example with one of them being the temporary vector resulting from a former operation).

What we need is a specialized function for each mathematical expression that we use, that computes the entire expression for each vector-element, and then moves on to the next element, computing the entire expression for that, and so on. For example:

```
for (int i=0; i<n; i++)
{
    A[i] = q * B[i] + r * C[i] + s * D[i];
}
```

for the expression from section 4.1.2, and again assuming `q`, `r`, and `s` are scalar-values of type `T`, and `A`, `B`, and `C` are one kind of array-implementation or another, with elements of type `T`, and supporting indexed lookup of their elements.

4.1.5 Composition Closure Objects

In [2, Section 22.4.7] a technique is suggested, which partially alleviates the need to create temporary `valarray` objects. The technique works by creating specialized temporary objects, that are dubbed *Composition Closure Objects* or *Compositors* for short, and then store references to the operands in these compositors, until larger expressions can be computed in one chunk.

Let us consider a slightly rewritten version of the example from [2, Section 22.4.7], where we wish to compute the following, for instances of `valarray`:

```
A = B * C + D;
```

Let us pretend this expression occurs often in our source-code, and we wish to apply a specialized function to avoid the creation of temporaries:[8]

[8] The source-code in this section is untested.

```
template <class T>
void mul_add_and_assign(valarray<T>& A,
                        valarray<T> const& B,
                        valarray<T> const& C,
                        valarray<T> const& D)
{
    // Ensure sizes of valarrays are consistent, etc.

    for (int i=0; i<A.size(); i++)
    {
        A[i] = B[i] * C[i] + D[i];
    }
}
```

For this kind of mathematical expression, we would then first create a compositor for the multiplication operator, and then another compositor for addition of an instance of that multiplication-compositor with another `valarray`-object. Let us start with the multiplication-compositor:

```
template <class T>
class Mul
{
public:
    Mul (valarray<T> const& l,
         valarray<T> const& r) : mL(l), mR(r) {}

    valarray<T> const& mL;
    valarray<T> const& mR;
}
```

And then we override the multiplication operator, to create such an object:

```
template <class T>
inline Mul operator*(valarray<T> const& l,
                     valarray<T> const& r)
{
    return Mul<T>(l, r);
}
```

Now we may create a compositor called `MulAdd`, used for wrapping an instance of the `Mul`-compositor with a `valarray`-object:

```
template <class T>
class MulAdd
{
public:
    MulAdd (Mul<T> const& m,
            valarray<T> const& v) : mM(m), mR(v) {}
```

```
    Mul<T> const&          mM;
    valarray<T> const&     mV;
}
```

And this time we need to override the addition operator for those specific types of arguments:

```
template <class T>
inline MulAdd<T> operator+(Mul<T> const& m,
                           valarray<T> const& v)
{
    return MulAdd<T>(m, v);
}
```

Of course, we should overload the addition operator for when the operands are interchanged, as we may reuse the `MulAdd`-compositor:

```
template <class T>
inline MulAdd<T> operator+(valarray<T> const& v,
                           Mul<T> const& m)
{
    return MulAdd<T>(m, v);
}
```

Finally, we have to actually carry out the operation of `B * C + D`, whenever needed; which means we have to make special cases of the constructor as well as assignment-operator for the `valarray`-class, meaning the actual mathematical evaluation of a multiplication and addition expression, is deferred until this point:

```
template <class T>
class valarray
{
    // ...

public:
    // Initialize by results.
    valarray (MulAdd<T> const& m)
    {
        // Allocate storage etc. ...

        mul_add_assign(this, m.mM.mL, m.mM.mR, m.mM.mV);
    }

    valarray& operator= (MulAdd<T> const& m)
    {
        mul_add_assign(this, m.mM.mL, m.mM.mR, m.mM.mV);
```

```
        return *this;
    }
}
```

So what essentially happens, when expressions such as the following occur, for `valarray`-objects:

```
A = B * C + D;
```

First, due to operator precedence, the `operator*` function is called with arguments `B` and `C`, which results in a `Mul`-object. Then the `operator+` function is called with its first argument being the `Mul`-object just created, and `D`. This results in a `MulAdd`-object. So far, no part of the mathematical expression has been computed, we have merely determined what kind of expression it is, and built a small tree that mirroring the mathematical expression. Then the `operator=` function is called with `A` and the `MulAdd`-object as arguments, and finally, this overloaded assignment operator retrieves the references for the sub-expressions `B`, `C`, and `D`, and calls the `mul_add_assign()`-function on these, which then performs the actual mathematical operations.

A short-coming of this compositor-technique, is that we still do not avoid the need for creating temporary instances of `valarray` in general, we have only avoided it for the special cases of expressions and sub-expressions, for which we will go through the trouble of making compositor-classes, override the appropriate operators, etc.

4.1.6 Expression Super-Class & Virtual Function For Element Lookup

Another solution to the problem of temporary objects and cache locality, is to make a super-class, that merely dictates the overriding of the element-lookup operator for its sub-classes:[9]

```
template <class T>
class Exp
{
public:
    Exp () {}

    virtual T&       operator[] (unsigned int i) = 0;

    virtual T const& operator[] (unsigned int i) const = 0;
}
```

Note that both `const` and non-`const` operators should be provided in these kinds of classes. Also note that we return references instead of values, and as usual because the datatype `T` may be expensive to copy.

The idea is then to create sub-classes of the `Exp`-class, that implement the various operators; multiplication, addition, division, etc. For example, we might have the following for multiplication:

[9] The source-code in this section is also untested.

```
template <class T>
class Mul : public Exp<T>
{
public:
    Mul (Exp<T> const& l,
         Exp<T> const& r) : mL(l), mR(r) {}

    virtual T&       operator[] (unsigned int i)
    {
        return mL[i] * mR[i];
    }

    virtual T const& operator[] (unsigned int i) const
    {
        return mL[i] * mR[i];
    }

    Exp<T> const& mL;
    Exp<T> const& mR;
}
```

Now, this class only stores references to `Exp`-objects, so either we need to make `valarray` a sub-class of `Exp`, or we must wrap `valarray`-objects before passing them to the `Mul`-constructor. Such a wrapper could be:

```
template <class T>
class Wrap : public Exp<T>
{
public:
    Wrap (valarray<T> const& v) : mV(v) {}

    virtual T&       operator[] (unsigned int i)
    {
        return mV[i];
    }

    virtual T const& operator[] (unsigned int i) const
    {
        return mV[i];
    }

    valarray<T> const& mV;
}
```

Then we need to override the multiplication operator with various combinations of parameters. First we have the case where both arguments are `valarray`-objects:

```
template <class T>
inline Mul operator*(valarray<T> const& l,
                     valarray<T> const& r)
{
    return Mul<T>(Wrap<T>(l), Wrap<T>(r));
}
```

Note that the source-code for this function is very similar to that of the previous section, with the exception that we wrap the arguments `l` and `r` in `Exp`-objects, before passing them to the constructor of the `Mul`-class. We also need the other combinations of arguments, where one or both of the operands are already `Exp`-objects:

```
template <class T>
inline Mul operator*(valarray<T> const& l,
                     Exp<T> const& r)
{
    return Mul<T>(Wrap<T>(l), r);
}
```

And of course:

```
template <class T>
inline Mul operator*(Exp<T> const& l,
                     valarray<T> const& r)
{
    return Mul<T>(l, Wrap<T>(r));
}
```

And the case when both arguments are `Exp`-objects already:

```
template <class T>
inline Mul operator*(Exp<T> const& l,
                     Exp<T> const& r)
{
    return Mul<T>(l, r);
}
```

Finally, we must provide the appropriate functions in `valarray` for construction and assignment, which now traverse the given source-expression one element at a time:

```
template <class T>
class valarray
{
    // ...

public:
    // Initialize by results.
```

```
    valarray (Exp<T> const& e)
    {
        // Allocate storage etc. ...

        for (int i=0; i<size(); i++)
        {
            (*this)[i] = e[i];
        }
    }

    valarray& operator= (Exp<T> const& m)
    {
        for (int i=0; i<size(); i++)
        {
            (*this)[i] = e[i];
        }

        return *this;
    }
}
```

So we must create a class for each kind of operator we wish to support, as well as override the corresponding operators. But once this work is all done, arbitrary expressions involving `valarray`-objects, will not require the implicit creation of any temporary objects.

There is a flipside however, and one which is a major drag indeed: For each call to an `operator[]` function, there is the overhead incurred by those functions being declared virtual. This means we no longer have high performance anyway, even though we have avoided the need for temporary objects altogether.

It is important to note, that the above technique builds an expression-tree at run-time, and also traverses that expression at run-time. Next we shall see a variant of this technique, that instead uses meta-programming, to build and `flatten` the traversal of the expression-tree at compile-time, thus overcoming the need for virtual functions, and hence the gross overhead involved.

4.2 Meta-Programming

Once again, we find template meta-programming useful, as it allows us to instruct the compiler how to generate specialized code for evaluating mathematical expressions whose operands are vectors. Provided the compiler has proper support for meta-programming, this means we not only get support for our much desired syntactic sugar, but altogether avoid the need for temporary objects holding intermediate results, and also have friendly cache-usage.

The overall idea is to have the compiler build an expression-tree out of template-classes as it parses the source-code, and then in a later phase, the compiler will *melt* these classes together, to form a single piece of efficient ex-

ecutable code for that particular expression, after which the expression-tree is discarded.[10]

4.3 Hardcoded Element-Datatype

There are different ways to implement meta-programming for a vector-datatype in C++. The simplest is when the type of the vector's elements is known and can be hardcoded into the various constructs needed for the meta-programming. To illustrate how this is done, a basic implementation supporting only binary expressions and addition is given in the following.

The following exposition is inspired by [3, Section 10.5], which in turn is inspired by the *blitz++* numeric computation library [4], described in section 4.15 below.

4.3.1 Expression Struct

The template class[11] representing a binary expression that evaluates to some `double`-typed value, is as follows:

```
template <class L, class Op, class R>
struct Expression
{
    Expression(L& l, R& r) : mL(l), mR(r) {}

    inline double operator[] (unsigned index)
    {
        return Op::apply(mL[index], mR[index]);
    }

    L& mL;
    R& mR;
};
```

Where the template arguments `L`, `Op`, and `R` are the left-hand, operator, and right-hand expressions, respectively. The operator for array-lookup is also overrided, and returns the value resulting from applying the given operator `Op` to the appropriate elements of the left- and right-hand expressions. When this is applied recursively, it means that the whole mathematical expression is evaluated for each element of the vectors, instead of evaluating parts of the expression on the whole vectors in turn.

4.3.2 Addition Struct

The `struct` representing the addition-operator, merely implements the `apply()` function, as follows:

[10] In the debug-version of the executable code generated by the C++ compiler used here, the actual expression-tree is actually output as code, and one can debug it step-by-step.

[11] Recall that the C++ `struct` is just a class with all its fields having `public` scope.

```
struct OpPlus
{
    static inline
    double apply(double a, double b) { return a + b; }
};
```

Note that the function is declared `static`, which allows us to use the function without having an actual instance of the `plus` struct.

Also note the function is declared `inline`, which is a further hint to the compiler, that it should insert the executable code for the function wherever it is being used, instead of inserting a call to that function. Doing this for all the arithmetic operations, means the entire mathematical expression is *flattened* into a single piece of code, and thus executes faster.

4.3.3 Overloaded Addition Operator

We need to override the addition operator so that we may write `B + C` for instances of our vector-class (which we have not defined yet). Since the return-type of the expression is hardcoded, C++ lets us get away with the following:

```
template <class L, class R>
Expression<L, plus, R> operator+(L& l, R& r)
{
    return Expression<L, OpPlus, R>(l, r);
};
```

Which works for all combinations of data-types for the left- and right-hand expressions, so we do not have to make special cases for all the different combinations of data-types, as long as they eventually resolve into something supporting the `operator[]` lookup that we need for the assignment-operator below.

4.3.4 Overloaded Assignment Operator

Finally we need to supply our vector-class with an assignment operator, which traverses the vectors and assigns to each element the appropriate value of its right-hand `x`:

```
template <class Expr> inline
LVectorDouble& LVectorDouble::operator=(Expr& x)
{
    for (int i=0; i<kDim; i++)
    {
        (*this)[i] = x[i];
    }

    return *this;
}
```

The class `LVectorDouble` is not detailed here. All we need to know, is that it must support the `operator[]` way of addressing its elements, that must be `double`-typed to match the above meta-programming.

Also note that this function is declared `inline`, which means one could have several nested assignments, such as:

```
A = B + (C = D);
```

and the compiler would then try and inline the two assignments and the addition, into a single piece of code, which could be optimized even further by the compiler, to result in just a single `for`-loop.

4.4 Resizable Vectors

We now turn to vectors whose elements are of some arbitrary type `T`. The constructs used for meta-programming are of course still implemented using templates, but introducing the additional template argument `T` turns out to be a bit problematic.

4.4.1 Storage

The actual storage of a resizable vector, is just a wrapper for STL's `vector`-class, and merely overrides the assignment-operator, with a looped iteration of the vector's elements:

```
template <class T>
class LVector : public std::vector<T>
{
public:
    LVector (unsigned int size=0) : std::vector<T>(size) {}

    // Assignment from any kind of data-structure or
    // expression supporting operator[] lookup.
    template <class Expr> inline
    LVector& operator= (Expr& x)
    {
        int i = 0;
        iterator itor = begin();

        for (; itor != end(); itor++, i++)
        {
            *itor = x[i];
        }

        return *this;
    }
};
```

4.4.2 Expression & Addition Structs

The template struct used for storing an abstract representation of a binary expression, is as follows:

```
template <typename T, class L, class Op, class R>
struct Expression
{
    Expression(L& l, R& r) : mL(l), mR(r) {}

    inline T operator[] (unsigned int index)
    {
        return Op::apply(mL[index], mR[index]);
    };

    L& mL;
    R& mR;
};
```

Where `T` merely substitutes the return-type `double` from the `Expression`-struct in section 4.3.1. The abstract addition operator has a similar and straightforward change made to it:

```
template <typename T>
struct OpPlus
{
    static inline T apply(T const& l,
                          T const& r) { return l+r; }
};
```

Note that parameters are passed as `const`-references. We would of course have to make one of these structs for each of the arithmetical operations that we wish to support.

4.4.3 Overloaded Addition Operator, Iterative Case

Unfortunately, we can no longer just use the very general operator-overloading for the binary `operator+`, as we did in section 4.3.3. Because of the template argument `T`, we now need to be more specific as to the types of the left- and right-hand expressions of an addition, to assist the C++ compiler in deducing the proper type `T`.

When both the left- and right-hand expressions of an addition are themselves instances of the `Expression`-class above, then the operator-overloading is as follows:

```
template <typename T,                       // Template arguments.
          class L1, class Op1, class R1,
          class L2, class Op2, class R2>
```

```
Expression<T, Expression<T, L1, Op1, R1>,  // Return-type.
              OpPlus<T>,
              Expression<T, L2, Op2, R2> >
operator+(Expression<T, L1, Op1, R1>& l,   // Function parameters.
          Expression<T, L2, Op2, R2>& r)
{
    return Expression<T, Expression<T, L1, Op1, R1>,
                         OpPlus<T>,
                         Expression<T, L2, Op2, R2> >(l, r);
};
```

Which rears the ever so ugly head of C++ syntax.

It would require a substantial amount of (largely redundant) work, to implement even the basic set of artihmetic operators required for vector computations, in this meta-programming framework. So what we would like, is either to use an existing library, or have better syntax and support for meta-programming in C++, or have an easy way of automatically generating meta-programs – that is, the support of *meta-meta*-programming. The latter two are out of the question, and for this project, we will eventually turn to the first existing libraries – although section 4.10 describes a meta-programming framework, which is very simple and powerful; even more so than the existing library that we will use.

4.4.4 Overloaded Addition Operator, Termination Cases

The remaining cases of operator overloading take care of the termination cases, when either one or both of the left- and right-hand expressions are `LVector` objects:

```
template <typename T>
Expression<T, LVector<T>, OpPlus<T>, LVector<T> >
operator+(LVector<T>& l, LVector<T>& r)
{
    return Expression<T, LVector<T>,
                         OpPlus<T>,
                         LVector<T> >(l, r);
};

template <typename T, class Expr>
Expression<T, LVector<T>, OpPlus<T>, Expr>
operator+(LVector<T>& l, Expr& r)
{
    return Expression<T, LVector<T>,
                         OpPlus<T>,
                         Expr>(l, r);
};

template <typename T, class Expr>
```

```
Expression<T, Expr, OpPlus<T>, LVector<T> >
operator+(Expr& l, LVector<T>& r)
{
    return Expression<T, Expr,
                      OpPlus<T>,
                      LVector<T> >(l, r);
};
```

One would have to implement similar operator overloadings for each arithmetic operation that must be supported. Once again, when the type `T` was hardcoded to be e.g. `double`, we did not need all of these special-cases.

4.5 Small Fixedsize Vectors

When the size of a vector is very small (say 4 or 8, or maybe 16 elements), and known at compile-time, it is desirable to *flatten* the assignment-loop, to avoid the overhead otherwise associated with performing the actual `for`-loop. The class storing such a vector, can be implemented as follows, where its size is now specified as a template argument:

```
template <class T, unsigned int kSize>
class LVectorMini
{
public:
    LVectorMini () {}

    // ...

protected:
    T mStorage[kSize];
};
```

This kind of storage is not required for this project, but is useful in many other contexts. For example when we have an outer-loop iterating many times over some arithmetic vector-expression involving fairly small vectors, in which case the cost of the `for`-loop in the assignment-operator of the `LVector`-class alone, would be responsible for a substantial portion of the total computational time used.

4.5.1 Overloaded Lookup Operator

Inside the `LVectorMini`-class, we would have the following overloaded lookup operator, taking an integer as its argument, and asserting it is the correct size, before returning the requested element:[12]

[12] Recall that assertions are not compiled into the socalled release-build of the code.

```
inline T& operator[](unsigned int i)
{
    assert(i<kSize);
    return mStorage[i];
}
```

And we also have a similar `const`-version.

4.5.2 Overloaded Assignment Operator

The overloaded assignment operator now calls a function from another class, `MAssign`, which is specified below:

```
template <class Expr> inline
LVectorMini& operator=(Expr& x)
{
    MAssign<Expr,kSize>::Do(*this, x);
    return *this;
}
```

The idea is that the `Do()` template-function unfolds through the use of meta-programming, to one long inlined sequence of evaluation- and assignment-operations of the elements of `x`. The reason for wrapping the `Do()` function in the `MAssign`-class, is another one of many C++ quirks.

The `MAssign`-class is also made to be a member of the `LVectorMini`-class, so that it may reuse the enclosing template-arguments of `LVectorMini`. The reason for having the `MAssign`-class, is that C++ does not allow us to merely make the `Do()`-function with several template-arguments. This problem was also encountered in section 3.2.8 on page 11, for the dot-product used in the IIR-filter class. The `MAssign`-class is as follows:

```
template <class Expr, unsigned int i>
class MAssign
{
public:
    static inline
    void Do (LVectorMini<T,kSize>& v, Expr& x)
    {
        v[i-1] = x[i-1];
        MAssign<Expr, i-1>::Do(v, x);
    }
};
```

Where the `Do()`-function makes use of another instantiation of that class, with the second template argument being decreased. This template argument is of course the counter for how many elements remain to be evaluated, and as it is decreased, we iterate through the vectors in a backwards manner. Since the vectors are assumed to be small, this should not give any (serious) cache

locality problems, but one could make a slightly more involved forward-traversal version instead, also by use of meta-programming. The termination-case for this backwards version, is as follows:

```
template <class Expr>
class MAssign<Expr, 0>
{
public:
    static inline void
    Do (LVectorMini<T,kSize>& v, Expr const& x)
    { /* Do nothing. */ }
};
```

4.5.3 Overloaded Addition Operator, Termination Cases

We also need to implement the overloaded addition operators for the termination cases of expressions containing `LVectorMini`-objects. However, these are completely analogous to the ones from section 4.4.4, and are hence omitted. Again, these would are not needed if the element-type `T` was hardcoded.

4.6 Using Pre-Allocated Storage

Imagine a situation, in which we want to perform computations using our existing meta-programming constructs, but on vectors that have been allocated elsewhere in the program.

The basic idea is to create a storage-class, which does not actually hold the data for the vector (as that would require costly copying of its elements), but merely holds a reference to the data that is supplied. The following allows arbitrary types `T` for the vector's elements, and the actual storage can also be an arbitrary data-structure, as long as it supports lookup of its elements using the `operator[]` function. For example, `T` could be `double` and the storage-type `U`, could be `double*` for a regular C/C++ array, or `std::vector<double>` for a C++ `vector`-container:

```
template <class T, class U>
class LVectorPre
{
public:
    LVectorPre (U& storage) : mStorage(storage) {}

    inline T& operator[](unsigned int i)
    {
        return mStorage[i];
    }

    // operator= same as for LVector ...
```

```
protected:
    U& mStorage;
};
```

To use this in meta-programming, it of course also requires overloading of the addition operator, similarly to how it was done in section 4.4.4 for `LVector`.

Again, if we only wish to use such a wrapper-class for a particular hardcoded data-type, our lives are a whole lot easier, as we would have something like the following for `T = double` and `U = double*`:

```
class LVectorPreDouble
{
public:
    LVectorPreDouble (double* storage) : mStorage(storage) {}

    inline double& operator[](unsigned int i)
    {
        return mStorage[i];
    }

    // operator= same as for LVector ...

protected:
    double* mStorage;
};
```

And then we would only need the simpler hardcoded meta-programming constructs from section 4.3.

4.7 Combining Vector-Types

By now, it should come as no surprise, that if we used hardcoded datatypes, such as in section 4.3 or above for `LVectorPreDouble`, then we could write expressions such as:

```
A = q * B + r * C + s * D;
```

with vectors `A`, `B`, `C`, and `D` being instances of different kinds of vector-objects (that is, objects overloading the `operator[]` function), and we would only need the meta-programming construct from section 4.3.3 to generate the expression-tree.

For arbitrary element datatypes `T`, we can still write expressions that combine instances of different vector-storage classes, but we must overload the addition operator for each termination case of such a combination. For example, to combine `LVector` and `LVectorMini` instances, we would need the following two termination cases:

```
template <typename T, unsigned int kSize>
```

```
Expression<T, LVector<T>,
              OpPlus<T>,
              LVectorMini<T,kSize> >
operator+(LVector<T>& l, LVectorMini<T,kSize>& r)
{
    return Expression<T, LVector<T>,
                         OpPlus<T>,
                         LVectorMini<T,kSize> >(l, r);
};

template <typename T, unsigned int kSize>
Expression<T, LVectorMini<T,kSize>,
              OpPlus<T>,
              LVector<T> >
operator+(LVectorMini<T,kSize>& l, LVector<T>& r)
{
    return Expression<T, LVectorMini<T,kSize>,
                         OpPlus<T>,
                         LVector<T> >(l, r);
};
```

And this would have to be done for each possible combination, for each arithmetical operator that we wish to support, all because we have the element type `T` as a template argument.

Note also, that the style by which the elements are traversed, is independent from this, and is either looped or flattened, depending solely on the class of the left-hand of the assignment operator, and how that class implements the assignment operator. For `LVector` it would be looped, for `LVectorMini` it would be flattened.

4.8 Vector Super-Class

It might be possible to create a common super-class for the different kinds of vector-types above, and then only provide the various meta-programming constructs for that one class, making the meta-programming framework available through normal inheritance from this super-class. We could call the class `LVectorBase`, and all we essentially need to provide the sub-classes with, are functions for assignment and element lookup.

We could declare the functions for assignment and element lookup to be pure virtual in `LVectorBase`, and then specialize them in `LVector`, `LVectorMini`, and so on. But for the `operator[]` function, this would incur a severe penalty, and would bring us back to square one, in terms of performance. So we need a way to specialize the `operator[]` function in the sub-classes of `LVectorBase`, without making them `virtual`. An impossibility one should think, but once more, template programming comes to our rescue.

4.8.1 Reverse Inheritance

A small trick that enables us to specialize the behaviour of a class, without the use of virtual functions, is described in [6, Section 3.3] and dubbed *Reverse Inheritance*. The idea behind reverse inheritance, is to provide specialization of a class, through inheritance from a super-class, that is supplied as a template argument to the former. In our case, we would have something like:[13]

```
template <class S>
class LVectorBase : public S
{
public:
    LVectorBase (unsigned int size) : S(size) {}

    // Omit operator[] and operator= functions.
}
```

Where `S` is the super-class providing actual storage of the vector's elements, as well as `inline` functions for element lookup and assignment. Such a storage-class could be `LVector` from section 4.4.1 that inherited from `std::vector`, or we could make a version that allocates and deletes the array directly:

```
template <class T>
class LVectorArr
{
public:
    LVectorArr (unsigned int size=0) : kSize(size)
    {
        assert(size>0);

        mArray = new T[size];
    }

    virtual ~LVectorArr()
    {
        delete mArray[];
    }

    inline T& operator[] (unsigned int i)
    {
        assert(i<kSize);

        return mArray[i];
```

[13] Please note this section merely depicts an idea, and further research is probably needed to make it work, particularly in regards to the exact C++ syntax. Section 4.10 presents an even better solution to a more general problem, that has been implemented and tested, and works. But this section has been left as is, to document the train of thought, as well as the idea of reverse inheritance.

```
    }

    inline T const& operator[] (unsigned int i) const
    {
        assert(i<kSize);

        return mArray[i];
    }

    // Assignment from any kind of data-structure or
    // expression supporting operator[] lookup.
    template <class Expr> inline
    LVectorArr& operator= (Expr& x)
    {
        for (int i=0; i<kSize; i++)
        {
            mArray[i] = x[i];
        }

        return *this;
    }

    protected:
        T*                      mArray;
        const unsigned int      kSize;
};
```

For convenience, we could then define the vector-types that we need, such as:

```
typedef LVectorBase<LVectorArr<double> > TVector;
```

Note the mandatory spacing between `<double>` and the right-most `>`, otherwise `>>` would be interpreted by the C++ compiler as an operator. Anyhow, we may then instantiate and work on our vector-objects as follows:

```
const int n = 10;                           // Vector-size.
TVector A(n), B(n), C(n);                   // The vectors.

// Initialize values of vectors ...

A = B + C;                                  // A[i] = B[i] + C[i].
```

Where the addition operator is provided solely through the meta-programming framework for `LVectorBase`.

4.9 Function Objects

We may use socalled *Function Objects* [2, Section 18.4] instead of `OpPlus` and similar structs. The classes used to instantiate the standard function objects,

derive from `unary_function` and `binary_function` provided in the header-file named `functional`. For example:

```
template <class Arg1, class Arg2, class Res>
struct binary_function
{
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
};
```

And the class used to instantiate addition-objects from, is then given by:

```
template <class T>
struct plus : public binary_function<T, T, T>
{
    T operator()(T const& l, const T& r) const
    {
        return (l + r);
    }
};
```

Since we override the `operator()` function instead of having a static function named `apply()` as in `OpPlus`, we first need to instantiate an object of type `Op`, and then invoke the `operator()` function on that object. The `Expression`-struct is therefore rewritten to be:

```
template <typename T, class L, class Op, class R>
struct Expression
{
    Expression(L& l, R& r) : mL(l), mR(r) {}

    inline T operator[] (unsigned int index)
    {
        Op op;
        return op(mL[index], mR[index]);
    };

    L& mL;
    R& mR;
};
```

All we have to change in the operator-overloadings, is to use e.g. `std::plus<T>` instead of `OpPlus<T>`, as follows:

```
template <typename T>
Expression<T, LVector<T>, std::plus<T>, LVector<T> >
operator+(LVector<T>& l, LVector<T>& r)
```

```
{
    return Expression<T, LVector<T>,
                      std::plus<T>,
                      LVector<T> >(l, r);
};
```

In a similar manner, we could have the constructor of `Expression` accept a parameter `op` of type `Op&`, which is already a function object instance, and hence allows for the user to supply her own function objects.

4.10 Expression Super-Class

It is actually possible to create a common super-class for the meta-programming expression-classes (or structs), as well as the set of `LVector`-classes; instead of just for the `LVector`-classes as suggested in section 4.8.

We can not make the `operator[]` virtual, as that would most likely spoil the meta-programming aspects, which is what we are after.[14] Once again, we therefore use reverse inheritance to specialize the `operator[]` function, while still having it declared `inline`. Let us first define the class representing any expression as follows:

```
template <typename T, class S>
class Expression : public S
{
public:
    Expression() : S() {}
};
```

Where `T` is the datatype of the vector-elements (e.g. `double`), and `S` is the super-class that `Expression` must inherit from (see examples below). Note that we are unable to pass arguments to the `Expression`-constructor, and then through to the constructor of the `S`-class, because some take no arguments, others take one or two.

4.10.1 Binary Expression

For a binary expression, we would have the `Expression`-class' base-class `S` being as follows, which simply stores its left- and right-hand expressions as pointers instead of object-references as usual, thus circumventing the inability to pass the references to the constructor:

```
template <typename T, class L, class Op, class R>
class BinExp
{
public:
```

[14] A clever compiler could realize that the virtual functions are being used as if they were inlined, but this should not be assumed.

```
    BinExp() : mL(0), mR(0) {}

    inline T operator[] (unsigned int index)
    {
        assert (mL && mR);

        // Create operator function-object.
        Op op;

        // Apply operator and return result.
        return op((*mL)[index], (*mR)[index]);
    };

    void Set(L* l, R* r)
    {
        // Set the left- and right-hand expressions.
        mL = l;
        mR = r;
    }

  L*  mL;    // Left expression.
  R*  mR;    // Right expression.
};
```

Also note that we make use of function objects instead of the `OpPlus` style of structs.

Then we would have the `Expression`-subclass for binary expressions, which essentially just calls the constructor of the `Expression`-class, and then calls the `Set()`-function from `BinExp`, to actually set the left- and right-hand expressions:

```
template <typename T,
          class L,
          class Op,
          class R>
class BinaryExpression :
      public Expression<T, BinExp<T, L, Op, R> >
{
public:
    BinaryExpression(L& l, R& r) :
        Expression<T, BinExp<T, L, Op, R > >()
    {
        BinExp<T, L, Op, R>::Set(&l, &r);
    }
};
```

And this class is what should be used in building the expression-tree by way of meta-programming.

4.10.2 Operator Overloading

The benefit of all this, is that we only need one operator-overloading for each operator we wish to support, and this works for all sub-classes of `Expression`; including the various kinds of vector-classes, and nested expressions. Take for example the addition-operator:

```
template <typename T, class S1, class S2>
BinaryExpression<T, Expression<T, S1>,
                    std::plus<T>,
                    Expression<T, S2> >
operator+(Expression<T, S1>& l, Expression<T, S2>& r)
{
    return BinaryExpression<T, Expression<T, S1>,
                               std::plus<T>,
                               Expression<T, S2> >(l, r);
};
```

4.10.3 Vector-Class

For the vector-class, let us use the example of `LVectorMini` from section 4.5. We split the class in two; the first for the actual storage of the vector's elements, and definition of its inlined `operator[]` functions for accessing those elements. This class will serve as the `S`-class of `Expression`, as can be seen below. We have:

```
template <class T, unsigned int kSize>
class LVectorMini_Imp
{
public:
    LVectorMini_Imp () {}

    // Element lookup.
    inline T& operator[] (unsigned int i)
    {
        assert(i<kSize);
        return mStorage[i];
    }

    // Element lookup, const.
    inline
    T const& operator[] (unsigned int i) const
    {
        assert(i<kSize);
        return mStorage[i];
    }
```

```
protected:
    T   mStorage[kSize];
};
```

This class is then used to define the actual class that one should instantiate vectors from, which we call `LVectorMini` once more:

```
template <class T, unsigned int kSize>
class LVectorMini :
    public Expression<T, LVectorMini_Imp<T, kSize> >
{
public:
    LVectorMini () :
        Expression<T, LVectorMini_Imp<T, kSize> >() {}

    // Assignment from any kind of data-structure
    // or expression supporting operator[] lookup.
    template <class Expr> inline
    LVectorMini& operator= (Expr& x)
    {
        for (int i=0; i<kSize; i++)
        {
            (*this)[i] = x[i];
        }

        return *this;
    }
};
```

For convenience, we have printed a version which uses a `for`-loop in the assignment-operator, whereas in section 4.5 this loop was flattened (also by use of meta-programming). The compiler might choose to flatten the `for`-loop by itself, as the integer `kSize` is known at compile-time. If one wants the choice and explicit control over whether to use a `for`-loop or a the flattened version, a boolean template-argument could be added, allowing switching of these two versions in the object instantiation, and thus generate the corresponding code at compile-time, without any additional run-time overhead.

4.10.4 Constant Values & Variables

This all seems fairly simple, but there are still many details that must be sorted out. Take for example expressions involving constants and variables, such as:

```
A = q * B + 10.0 * C;
```

Where `A`, `B`, and `C` are vectors, `q` is a variable, and `10.0` is of course a constant value. If `q` is expensive to copy, then we would like to just store a reference instead of a copy in the meta-programming classes. But C++ does not allow us

to store a reference to a constant value such as the `10.0` here; or more precisely, the `double`-object holding the value `10.0`, is destroyed before we get around to using it.

Fortunately enough though, we may overload the multiplication operator with one operand being an `Expression`-object, and the other being either typed as `T&` or `T const&`, where the former overloading is used for variables (such as `q`), and the latter is used for constants (such as `10.0`).

Another problem is that we should like to use the above `BinaryExpression`-class for expressions that also involve such constants or variables. That is, we would like to make the corresponding argument to the `BinaryExpression`-constructor, an instance of, say, `ConstantExpression`. However, implementing it as follows, is not correct, because the `ConstantExpression`-object is destroyed before it must be used:

```
template <typename T, class S1, class S2>
BinaryExpression<T, Expression<T, S1>,
                    std::plus<T>,
                    ConstantExpression<T> >
operator+(Expression<T, S1>& l, T const& r)
{
    return BinaryExpression<T, Expression<T, S1>,
                               std::plus<T>,
                               ConstantExpression<T> >(l,
                                     ConstantExpression<T>(r));
};
```

To solve all these problems, we make use of something that could be called *Switched Storage*. We will not give the full details here, but merely describe the idea, which is to make a template-class, that takes as template-arguments the datatype `T`, and a boolean designating whether it should hold a copy or a reference to some variable. We then use this storage-class for the left- and right-hand expressions in the `BinaryExpression`-class, as well as the `ConstantExpression`-class. The only trouble is, that we now also have to pass the additional booleans around, which makes the source-code more difficult to understand and maintain.

4.10.5 Macro-Meta-Programming

Recall our early efforts of using meta-programming to implement an arithmetic vector class, and how we would like to have something we called meta-meta-programming available. Well, it does actually exist, in the form of macros – yes, the old-school C macros, which are strongly discouraged in C++ (and for good reasons).

Take the operator overloadings as an example. Here we will need to make almost identical overloads for each operator, where we only change the name of the operator (e.g. from `operator+` to `operator-`) and the function-class that should be applied (e.g. from `std::plus` to `std::minus`). Now, these are really just simple text-manipulations, something which macros are very good

at; in fact, that is the entire purpose of macros. So what we do, is to make a macro, say `MakeOperator(OP, FUNCTOR)`, that takes as arguments the name of the operator to overload, and the name of the function-class to use. Then we instantiate this macro for each operator that we need, and have thereby reduced the manual programming chore substantially. As long as we only use the macros in our internal programming, and do not expose them for the ordinary user of the vector library, then we should be able to avoid any wrongful use, which is an inherent risk of using macros.

This sort of programming can be called *macro-meta-programming* or meta-meta-programming. However, you should note that macros are not as powerful a tool as template meta-programming itself, in that macros may not be self-referential or recursive. However, if used decently, they can decrease the amount of largely redundant source-code, to an incredible extent.

4.11 User-Supplied Functions

It would also be desirable to extend the meta-programming framework, in such a way that a user may apply her own functions to sub-expressions. For example, imagine we have some function `f()` defined only for arguments of datatype `T`, and we now wish to compute the following expression, for entire vectors with elements of type `T` (say `LVectorMini<T, kSize>` vectors):

```
A = B + f(C + D);
```

Through the use of meta-programming, we would like this to effectively compile into the following loop:

```
for (int i=0; i<kSize; i++)
{
    A[i] = B[i] + f(C[i] + D[i]);
}
```

That is, the function `f()` is applied on the elements of its argument-vectors, on an element-by-element basis. All the usual mathematical functions such as `sin()`, `cos()`, and `log()`, should of course be provided this way.

The direct way of doing this, would be to provide two versions of `f()`, one taking an `Expression`-object as argument, and another taking an argument of type `T`. The one taking the `Expression`-argument, would then have to return a `UnaryExpression`-object, which is to overload the `operator[]` function, making it execute the `f()` function on the desired element. This would integrate seamlessly with the meta-programming framework, and facilitate flattening of expressions as in the example above. A concrete example of how to do this, is given in the next section on type-casting.

4.12 Type-Casting

It is sometimes convenient to have a single arithmetic expression, operating on vectors whose elements are of different types. For example, one could perform

addition of two vectors, one having `double`-typed elements, and the other having `float`-typed elements. This is not allowed in the meta-programming framework from section 4.10, because the `Expression`-class consistently demands the type of its elements. This is a good thing really, as it means we have stronger type-checking (the exception to this being the assignment-operator in `LVectorMini`, which makes no explicit type-checking by way of template-arguments).

4.12.1 Casting Of Each Vector-Element

What we want, is for casting to be done on an element-by-element basis, much like the application of a function `f()` from section 4.11. But first note that we should not override `static_cast`, `dynamic_cast`, and so on, for `Expression`-typed parameters. The reason is, that we may have a sub-class of `LVectorMini` – for example a class representing an image – and we may wish to use traditional re-casting on a `LVectorMini`-object, to see if it is actually some other class.[15]

4.12.2 Vector-Casting Functions

What we basically do, is to implement functions for each of the casting-types, for performing the corresponding casting of vectors on an element-by-element basis. For example, we might create a function `StaticCast()`, taking as parameter the `Expression`-object we wish to cast, and returns a `UnaryExpression`-object, for which the function to perform is `static_cast()`. First we define the class `statcast`, used for instantiating function objects, that actually perform the casting:[16]

```
template <typename T1, typename T2, class S>
struct statcast : public unary_function<T1, T2>
{
    T1 operator()(Expression<T2, S> const& exp) const
    {
        return static_cast<T1>(exp);
    }
};
```

Then we make the function `StaticCast()` which is to be called by the user, for casting vector-expressions:

```
template <typename T1, typename T2, class S>
UnaryExpression<T, Expression<T1, S>,
                   statcast<T> >
StaticCast(Expression<T2, S>& exp)
{
```

[15] Note that casting between different sub-classes of `Expression`, is not necessary in order to combine them in a single arithmetic expression. We may combine instances of `LVector` and `LVectorMini` in arithmetic expressions, as long as they both inherit from the `Expression`-class. This is one of the great things about the framwork from section 4.10.

[16] The source-code in this section is untested.

```
    return UnaryExpression<T, Expression<T1, S>,
                           statcast<T> >(exp);
};
```

4.12.3 Example

So we might have something like the following, where we wish to cast the result of adding two vectors, and add this result to a third vector, before finally assigning the result to a fourth vector:

```
const int n = 10;                          // Vector-size.
LVectorMini<double, n> A, B;               // Vectors (double).
LVectorMini<int, n> C, D;                  // Vectors (int).

// Initialize values of vectors ...

A = B + StaticCast<double>(C+D);
```

Where the last statement effectively compiles to:

```
for (int i=0; i<n; i++)
{
    A[i] = B[i] + static_cast<double>(C[i] + D[i]);
}
```

4.13 OpenMP Support

OpenMP is a set of compiler directives, that some C++ compilers support, and some do not. The directives are described in [7], and are used to instruct the compiler in how to compute portions of the program in parallel, by splitting and merging the computation at designated points in the code. OpenMP only supports *Shared Memory Processors* (SMP), for example machines with several CPUs, or where each CPU has several *cores* (such as Intel's HyperThreading technology, which is now available in their consumer-level CPUs). OpenMP does not support distributed computing with several machines connected over a network, however.

An added benefit of having a single `for`-loop in the computation of arithmetic expressions, is that we may add support for OpenMP incredibly easy, by adding just a single line of source-code.[17] If we take the `LVectorArr`-class from section 4.8.1 as an example, we would have:

```
template <class Expr> inline
LVectorArr& operator= (Expr& x)
{
    #pragma omp parallel for
```

[17] This incentive for using meta-programming, was also described in the paper [5, Section 4.4], that was supplied in the course-notes.

```
    for (int i=0; i<kSize; i++)
    {
        mArray[i] = x[i];
    }

    return *this;
}
```

This also works with compilers that do not support OpenMP, as the `pragma`-directive is simply ignored.

4.14 SIMD Support

Another way to perform parallel computation, is the socalled *Single Instruction Multiple Data* (SIMD) approach. In SIMD we perform the same operation, for example addition, on a small vector of numbers.

Generally speaking, there are several problems involved in SIMD programming. First off, it requires machine-specific low-level programming, and furthermore, as its basic datatype is a small vector of data, we can not pass the SIMD instructions just any data from memory. The data has to be aligned properly,[18] and we need a way to cope with arrays of data, that may not contain an exact number of such small vectors (e.g. the array may have 127 numbers, and if one SIMD vector consists of 4 numbers, then the last SIMD operation will only have 3 numbers in its SIMD vector).

With minimum effort, we can make the set of `LVector`-classes support SIMD in numerous ways, and still avoid many of the problems often encountered in SIMD programming.

4.14.1 SIMD Vector Data

The basic idea is to instantiate the `LVector`-class with a datatype supported by SIMD. For example, on Intel's Pentium III and IV processors, the SIMD co-processors are called SSE and SSE2, respectively, and they use particular datatypes for their small vectors. Intel has provided a number of header-files with intrinsics for their SIMD processors, that define the relevant datatypes. For single-precision floating point numbers, the header-file is called `xmmintrin.h`, which defines the datatype `__m128`, that packs a vector of 4 single-precision floating point numbers for use on the SSE and SSE2 co-processors. This datatype is also ensured to be properly aligned in memory.

We could instantiate vectors of type `LVector<__m128>`, and then specialize the arithmetic operators for arguments of type `__m128`, and have them call the specific SSE/SSE2 instructions. However, Intel has provided another header-file (`fvec.h`), which defines a class `F32vec4`, having all the usual operators

[18] That is, the address of the data must satisfy certain conditions, e.g. that the last few bits are zero.

overloaded. So we have instant support for Intel's SSE-processors, merely by instantiating our vectors as `LVector<F32vec4>`.[19]

4.14.2 Transparency & Cross-Platform Support

There are a few issues that must be resolved though, before we can use the SIMD extensions to `LVector` in general. Essentially, we need to make the use of SIMD transparent to the programmer, both in regards to whether or not the machine on which the program is to be run, actually supports e.g. SSE or SSE2, but also in regards to which platform (and hence the kind of SIMD co-processor) the program is to be run on.

The cross-platform issue can be solved by strategically placing `#ifdef`-statements in the source-code, and include different header-files depending on which platform the code is compiled for.

The has/has-not SIMD issue can be solved by allocating new `LVector`-objects through a special function at run-time, which first tests for SIMD support on the given machine, and depending on the result, allocates either e.g. `LVector<float>` or `LVector<F32vec4>`, and with the appropriate number of elements.

4.14.3 SIMD Element Lookup

This brings us to another most important issue: How do we address elements of the SIMD vectors, inside an `LVector`-object? Merely calling the `operator[]` function on an `LVector<F32vec4>`-object, naturally returns an `F32vec4`-object, and one would have to call the lookup-function `operator[]` on that object also, to get the individual elements of the SIMD vector. But then the interface is no longer transparent, in terms of whether or not SIMD is supported by the machine at hand.

One should try and find a graceful solution to this problem. A preliminary suggestion, is to provide another function in `LVector` for lookup of elements, say `operator()`, which is specialized for `LVector<T>` with `T` being SIMD datatypes, and should be used by the programmer, instead of `operator[]`, which is then only to be used by the meta-programming facilities. Again, this is a preliminary suggestion.

4.14.4 Automatic Compiler Support For SIMD

Effort is continuously put into improving C++ compilers, also in regards to how they optimize the code they output. For this reason, it may not be necessary to provide explicit support for SIMD in the set of `LVector`-classes, as it may be done automatically by the compiler. But even if this is so, one should still conduct tests to find out which of the methods results in the fastest code: The hand-coded, or the one automatically generated by the compiler. In case the

[19] Preliminary tests were conducted with addition of `LVectorMini<F32vec4, k>`-vectors, and `k` being some integer constant – and it worked.

hand-coded version is better, one is better off disabling the SIMD compiler-optimization for the `LVector`-classes.

4.15 The Blitz++ Library

Much of the pioneering of meta-programming in C++, seems to be made by the blitz++ library. The library still has known bugs in some of its more esoteric features, but for our requirements, its range of working features is plentiful. The blitz++ library appears highly complicated in its implementation, and is therefore difficult to maintain and extend. It is believed that the framework outlined in section 4.10, could prove to be greatly superior to the blitz++ library. However, for the time being, we shall use blitz++ for this image-processing project.

4.15.1 Vector-Class

The general blitz++ class for storing vectors, resides in the `blitz` namespace, and is called `Array`. It takes two template arguments, the first being the type of its elements, and the second being its dimensionality. The constructor for the `Array`-class, then takes as its argument, the desired size of the vector. So the example from section 4.1.1 would become:

```
const int n = 10;                          // Vector-size.
blitz::Array<double,1> A(n), B(n), C(n);  // The vectors.

// Initialize values of vectors ...

A = B + C;                                 // A[i] = B[i] + C[i].
```

5 Testing

For these tests, the implementation is carried out in *MS Visual C++ .NET 2003*. During development, multiple tests were of course performed (including checking the generated machine-code), but we will only present a small number of these tests here, with the purpose of demonstrating, that things are working as expected.

5.1 Vector Implementations

Let us begin with the vector implementations; both `LVectorMini` given in section 4.10.3, and blitz++ described in section 4.15. Starting with `LVectorMini`, we have the following test-code:

```
const kSize = 4;
ArrayOps::LVectorMini<double, kSize> A, B, C;
```

```
for (int i=0; i<kSize; i++)
{
    A[i] = i+1;
    B[i] = A[i]*10;
    C[i] = B[i]*10;
}

A = A + B + C;

for (int i=0; i<kSize; i++)
{
    std::cout << A[i] << std :: endl;
}
```

Which initializes the three vectors `A`, `B`, and `C`, so that summing their elements should results in the values 111, 222, 333, and 444, which is indeed what is output by the small program:

```
111
222
333
444
```

The same test-code using the blitz++ `Array`-class instead of `LVectorMini`, is as follows:

```
const kSize = 4;
blitz::Array<double, 1> A(kSize), B(kSize), C(kSize);

for (int i=0; i<kSize; i++)
{
    A(i) = i+1;
    B(i) = A(i)*10;
    C(i) = B(i)*10;
}

A = A + B + C;

std::cout << A << std::endl;
```

Note that elements are addressed using rounded parentheses instead of square brackets. This is not always desirable, since C and C++ addresses elements in an array with square brackets, and the blitz++ syntax is therefore somewhat non-standard. At any rate, the output is as expected (printed in blitz++ format):

```
4
 [       111       222       333       444  ]
```

5.2 Filter Testing

Before testing the two lowpass filter implementations, let us first consider what kind of output to expect. Looking at Eq.(2) on page 3, we see that if $x[n] = 0$ for all n, then the output must be:

$$y[n] = c^{n+1} \cdot y[-1]$$

Where $y[-1]$ is the initial internal value of the filter. Setting $y[-1] = 1$, we would have the following sequence of output values of the lowpass filter:

$$y[0] = c,\ y[1] = c^2,\ y[2] = c^3,\ \cdots$$

and setting c to some value, say $c = 0.85$, we would get the following sequence of (rounded) output values:

$$0.85,\ 0.723,\ 0.614,\ 0.522,\ 0.444,\ 0.377,\ 0.321,\ 0.272,\ 0.232,\ 0.197$$

If $y[-1]$ was instead some other value, we would just have to scale these numbers accordingly. So for example, if $y[-1] = 2$, then the sequence would have been 1.7, 1.445, 1.228, and so on.

5.2.1 Scalar-Valued Filters

Let us begin with a test of the general lowpass filter (the one that uses meta-programming), and with scalar datatypes. The test-program is as follows, which simply instantiates such a filter, and performs ten iterations:

```
const int kIterations = 10;
const double kCutoff = 0.85;

typedef double T;
T a, x;

a = 1;     // Filter's initial value.
x = 0;     // Filter's input.

// Instantiate filter and set its cutoff.
DSP::LFilterIIR_LP1<T, double> filter(a);
filter.SetParameters(kCutoff);

// Perform iterations of the filter, output result.
for (int i=0; i<kIterations; i++)
{
    std::cout << filter(x) << std::endl;
}
```

The output of this program is as follows, and matches the theoretical output:

```
0.85
0.7225
0.614125
0.522006
0.443705
0.37715
0.320577
0.272491
0.231617
0.196874
```

Running the same test, only with the hard-coded lowpass filter instead, merely requires us to change the line:

```
DSP::LFilterIIR_LP1<T, double> filter(a);
```

into the following:

```
DSP::LFilterIIR_LP1_HC<T, double> filter(a);
```

The output of the program is then:

```
0.85
0.7225
0.614125
0.522006
0.443705
0.37715
0.320577
0.272491
0.231617
0.196874
```

Which is also as expected.

5.2.2 Vector-Valued Filter

Since the `LVectorMini`-implementation is not complete (it lacks multiplication with scalar values and variables), we shall omit testing of the filters with that class here.[20] Also, we shall only consider the hardcoded lowpass filter, as the generalized filter implementation does not compile with blitz++ vectors as its datatype.[21] The test-program is as follows:

```
const int kIterations = 10;
const double kCutoff = 0.85;
```

[20] Note that testing during development has been performed, by setting the coefficient-datatype of the filter, to `LVectorMini` also.

[21] An obscure error-message is generated, that originates in blitz++ somewhere – the reason for this was discussed in section 3.2.8 above.

```
typedef blitz::Array<double,1> T;
const kSize = 4;

T a(kSize), x(kSize);
x = 0;                 // Filter's input.

// Generate the initial value of the filter.
for (int i=0; i<kSize; i++)
{
    a(i) = i+1;
}

// Instantiate filter and set its cutoff.
DSP::LFilterIIR_LP1_HC<T, double> filter(a);
filter.SetParameters(kCutoff);

// Perform iterations of the filter, output result.
for (int i=0; i<kIterations; i++)
{
    std::cout << filter(x) << std::endl;
}
```

Which produces the following output:

```
4
 [      0.85       1.7      2.55       3.4  ]
4
 [    0.7225     1.445    2.1675      2.89  ]
4
 [  0.614125   1.22825   1.84237    2.4565  ]
4
 [  0.522006   1.04401   1.56602   2.08802  ]
4
 [  0.443705  0.887411   1.33112   1.77482  ]
4
 [   0.37715  0.754299   1.13145    1.5086  ]
4
 [  0.320577  0.641154  0.961731   1.28231  ]
4
 [  0.272491  0.544981  0.817472   1.08996  ]
4
 [  0.231617  0.463234  0.694851  0.926468  ]
4
 [  0.196874  0.393749  0.590623  0.787498  ]
```

Again, this output is as expected, and we may conclude, that the hardcoded lowpass filter, works for vector-datatypes such as the one from the blitz++

library. This means that we would indeed be able to process video through such filters, and with a fairly small programming effort (once the vector library has been implemented).

6 Conclusion

Much ground was covered in this report. First a general implementation of a socalled IIR filter was given, that used template meta-programming in C++ for efficiency of the generated code. Then a specialized version of one of these filters was given, which was of course much easier to implement, and indeed showed that the effort needed for making the general IIR filter, was not justified by what was saved due to re-usability.

Of greater importance however, was perhaps section 4 which described various implementations of a numeric vector, supporting the usual arithmetic operators to different degrees of efficiency. By combining the following techniques, a very simple and efficient implementation was found:

- Meta-programming provided the overall framework for generating specialized code at compile-time.
- Reverse inheritance allowed much greater reuse of the meta-programming framework.
- Macro-meta-programming made specializations for this framework easy for different operators.

Although this resulted in fairly simple source-code, with only few lines of manual programming required, it can actually be simplified even further. It is believed that a full-fledged implementation of this framework, would be vastly superior to the libraries currently in existence (such as blitz++). Although all of the basics in such a library are presented here, there are still a number of details to be sorted out though.

Finally, testing showed that the original idea – that is, applying audio filters to video data – was possible with fairly simple means, when a decent vector processing library was available.

References

[1] Steven W. Smith, Digital Signal Processing – A Practical Guide for Engineers and Scientists, Newnes (Elsevier Science) 2003, ISBN 0-750674-44-X

[2] Bjarne Stroustrup, The C++ Programming Language 3rd ed., Addison-Wesley 1997, ISBN 0-201-88954-4

[3] David Abrahams and Aleksey Gurtovoy, C++ Template Metaprogramming, Addison-Wesley 2005, ISBN 0-321-22725-5

[4] Todd Veldhuizen, Blitz++, http://oonumerics.org/blitz/

[5] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. In Lecture Notes in Computer Science 3016, Dagstuhl seminar on Domain-Specific Program Generation. AOOP Note #18.

[6] Magnus Erik Hvass Pedersen, Mandatory Assignment 2, Scientific Computing Course, Danish Technical University, November 2005, http://www.daimi.au.dk/~u971055/scientific/sc_mandatory2.pdf

[7] OpenMP Specification version 2.5, May 2005, http://www.compunity.org/