Compile time sparse vectors in C++

Jaakko Järvi
University of Turku, Department of Computer Science,
Lemminkäisenkatu 14, FIN-20520 Turku, Finland

T U C S

Turku Centre for Computer Science
TUCS Technical Report No 107
March 1997
ISBN 951-650-988-6
ISSN 1239-1891

Abstract

Templates are a powerful feature of C++. In this article a template library for a special class of sparse vectors is outlined. For these vectors, the sparseness structure of the vectors can be arbitrary but must be known at compile time. In this case it suffices to store only the nonzero elements of the vectors, no indexing information about the sparseness pattern is required. The indexing information is contained in the type of the sparse vector as non-type template parameters. It is shown how common vector operators can be overloaded for the given vectors. With these operators it is possible to write abstract vector expressions between arbitrary sparse vectors. When compiled the operators yield code which performs only the necessary elementary operations between nonzero elements without any runtime penalty for the indexing of zero and nonzero elements. All indexing is performed at compile time resulting in very fast execution speed. The vector classes are best suited for short vectors up to few dozens of elements.

Automatic differentiation of expressions is given as an example application for the sparse vector library. It is shown how classes for automatically differentiable numbers can be defined with the sparse vectors of the library. A comparison against other vector representations gave superior results in execution speed of differentiating few common expressions and came very close to the calculation speed of symbolically differentiated expressions.

Keywords: sparse vectors, C++ templates, automatic differentiation

TUCS Research Group

Algorithmics Group

1. Introduction

Templates are a powerful feature of C++. Templates are usually taken as a means to write generic classes and functions, but we can go beyond that. With clever usage of templates, it is possible to make the compiler to perform as an interpreter at compile time. For instance compile time bounded loops and branching statements (if, switch) can be written using recursive template definitions and instance specialisation. This kind of usage of templates is referred to as *template metaprograms* [1].

In this report template metaprograms are utilised in definition of *compile time sparse vector* (CTSV) classes. The term compile time here means that the structure of the vectors (the positions containing zero and non-zero elements) must be known at compile time. This is of course a major restriction, and the definition is not meant to serve as a general purpose sparse vector representation. Instead the definition yields very efficient code for special applications where the above requirement can be fulfilled. It is first shown how common operations can be overloaded to produce very effective code in expressions of CTSVs and in the end of the report it is shown how CTSVs can be used with automatic differentiation [2].

The techniques presented take advantage of the new template features present in the current draft standard of C++ [3], and are not necessarily implemented in all compilers yet.

1.1. Template metaprograms

In C++ the formal parameter of a template class can either be a type parameter or a constant expression of some integral type. The use of constant expressions as template parameters is a key factor behind template metaprograms. As an example of a simple template metaprogram, consider the code in Listing 1.

```
template < unsigned int N>
class F {
public:
  enum { val = N * F<N-1>::val };
};

class F<0> {
public:
  enum { val = 1 };
};
```

Listing 1. A template metaprogram for calculating the factorial at compile time.

The template class `F` is parameterised by a single integral template parameter `N`. It contains only an enumeration definition for the constant `val`, which is intended to contain the factorial of `N`. In the definition of the enumeration constant `val` of the class `F<N>`, the enumeration constant val of the class `F<N-1>` is used. Hence, the compiler must instantiate the class `F<N-1>` before the class `F<N>` can be instantiated. When encountering the instantiation of `F<N>` the compiler therefore triggers the instantiation of `F<N-1>`, which will trigger the instantiation of `F<N-2>` etc. The spesialisation for `F<0>` has been given, which ends the recursion. This all occurs at compile time and the enumeration constant `val` will indeed be given the value `N!` at compile time.

The factorial definition serves as an example of compile time loop. Other control flow structures can also be defined. Listing 2 shows an example of a compile time if statement. Depending on `condition`, which must be a compile time constant expression, either `statement1` or `statement2` is generated. Note, that the statement function is defined as static, which enables the above call syntax where a member function is called without an instantiated object of the class.

```
// Class declarations
template< bool c>
class ifelse {};

class ifelse<true> {
public:
  static inline void statement() { statement1; // true case}
};

class ifelse<false> {
public:
  static inline void statement() { statement2; // false case }
};

// Usage
ifelse<condition>::statement();
```

Listing 2. Compile time if-statement

Compile time switch statement can be implemented analogously, using some integral type as the template parameter and giving specialisations for each case of the value. The integral type can be an enumeration, so case labels can be used. The default case is given as the general template definition. For an excellent introduction to template metaprograms see [1], which is also the source of the previous examples.

1.2. Sparse vectors

A vector is called sparse if only a relative small number of its elements are nonzero. The sparseness of vectors can be taken advantage of when storing or processing them. Instead of storing all the elements of a vector, only the nonzero elements can be stored along with some auxiliary information to determine the logical positions of the elements in the vector. In addition to space, also time can be saved with sparse vectors, since certain elementary operations only have to be performed to nonzero elements.

Basically a sparse vector $\mathbf{x}$ can be written as a set of value-position pairs:

$$\mathbf{x} = ((x_{i_1}, i_1), (x_{i_2}, i_2), \ldots, (x_{i_n}, i_n)) .$$

With this notation we can e.g. write the sum of two sparse vectors (1,0,1,0,0) and (0,0,2,0,2) as

((1,1), (1,3)) + ((2,3),(2,5)) = ((1,1),(3,3),(2,5)).

As can be seen, instead of five additions between the elements, only one addition and two value copy operations are needed to calculate the result. To perform only the necessary elementary operations, some kind of indexing scheme must be used to find the right operands. This bookkeeping causes some extra overhead and it is desirable to minimise it. For an introduction to various indexing schemes, see [4].

2. Compile time sparse vectors

If the positions of the zero and non-zero elements of sparse vectors are known at compile time, it is possible to write very efficient code for performing operations with them. E.g. for the example of section 1.2. the code would contain one elementary add operation and two move operations. It is of course too tedious to write the code like that directly and in this section it is shown how the code can be generated from abstract vector expressions using template metaprograms.

The template definitions tend to become rather lengthy. Therefore the code in this article is presented for floating point vectors instead of generic vectors. It is however straightforward to generalise the class definitions by making the element type also a template parameter.

2.1. Representing CTSVs as C++ classes

An optimal representation for a sparse vector would contain only the nonzero elements and nothing else. This means that no memory is allocated for the nonzero elements or for any kind of indexing information! This optimal representation can in fact be achieved with template classes where the indexing information will be contained in the template parameters, i.e., in the type of the vector. Hence each vector with a different sparseness pattern is a type of its own.

A special representation for a vector element is needed. Listing 3 shows the definition of the class Elem, giving the definition for the element type. It has a single template parameter N and the general class definition for any N is a class having no data members, just two constructors. For N = 1 a specialisation is provided, containing a single data member for storing a value. Elem<0> is therefore an empty class, whereas Elem<1> is a class containing a single floating point value. Other values than 0 or 1 for N are not meant to be used. The default constructor is defined as not to do anything and the constructor taking a single floating point is meant for initialisation the element with a value. To give a uniform interface the class Elem<0> also has a constructor taking a floating point value, though it performs no action.

```
template <unsigned int N>
class Elem {
public:
  Elem<N> (float vv) {}
  Elem<N> () {}
};

class Elem<1> {
public:
  float value;

  Elem<1>(float vv) : value(vv) {}
  Elem<1>() {};
};
```

Listing 3. Elem-class.

The actual CTSV is a collection of Elem<N> objects: Elem<1> objects in nonzero positions and Elem<0> in zero positions. With this kind of element type definitions, we are able to store empty classes, i.e., store nothing as the zero elements. It must be noted that depending on the implementation, an empty class may not be totally empty. Most commonly a ‘dummy’ byte is allocated for an empty object.

The next step is to include the indexing information to specify the zero and nonzero positions. This is also achieved with template parameters. Note first, that a sparse vector $\mathbf{x}$ can be characterised with a bit sequence $\mathbf{b}_{\mathbf{x}}$, where ‘1’ corresponds to a nonzero element and ‘0’ to a zero element respectively. As integral types can be manipulated as bit patterns in C++, an unsigned integer template parameter can be used to represent the bit sequence in the generic CTSV class definition. The bit pattern of this template parameter determines the nonzero positions of the CTSV. More precisely, Elem<1> type objects are stored at nonzero positions and Elem<0> type objects at zero positions. The class definition is shown in Listing 4.

```
template <unsigned int N>
class CTSV {
public:
  enum { bits = N };

  Elem< N & 1 > head;
  CTSV< N >> 1 > tail;

  CTSV<N>(float v) : head(v), tail(v) {}
  CTSV<N>(){};

  CTSV< N >> 1 >& GetTail() { return tail; }
};

class CTSV<0> {
public:
  enum {bits = 0 };

  Elem<0> head;

  CTSV<0>(float v) : head() {};
  CTSV<0>(){};

  inline CTSV<0>& GetTail() { return *this; }
};
```

Listing 4. CTSV-class.

The definition is recursive. The `head` member holds the value of an element. Whether it will be of type `Elem<0>` or `Elem<1>` is determined by the least significant bit of the template parameter N. This can be examined by taking the bitwise AND operation with 1. Hence, if the least significant bit of the CTSVs characteristic bit sequence is 1, a value is stored, otherwise an empty object is stored. The `tail` member holds the remaining part of the CTSV. The new template parameter for the next step of the recursion is given by shifting `N` one bit to right, i.e., dropping the least significant bit. This will eventually result to the template parameter being zero and a specialisation for `N = 0` is provided to end the recursion. Note, that the enumeration constant `bits` holds the value of `N`, which will be needed to define operations for CTSVs.

The default constructor is defined as not to do anything. In addition, a constructor taking a single float argument is provided for initialisation of a vector with a given value. It will merely pass the same argument to the head and tail members. In the case of `Elem<1>` type head, the value is stored, otherwise nothing is done, since the corresponding `Elem<0>` constructor is empty. The recursion is ended with the constructor of `CTSV<0>` which also is empty.

The `CTSV<0>` has no tail-member, but instead a function `GetTail`. The general version of the `GetTail` function returns just the tail of the CTSV, but the specialisation for `CTSV<0>` returns the vector itself. This makes the definition of operations somewhat easier, as we can also get the tail of a `CTSV<0>` object, and a tail of a tail of a `CTSV<0>` object, etc.

The above class definition leads to quite a number of function calls and it is crucial that all the functions are inlined. Due to inlining, empty functions are discarded from the compiled code.

2.2. Operations between CTSVs

The most common mathematical operations defined for vectors are addition, subtraction, unary negation, multiplication with a scalar and the dot product. Also some arbitrary function can be applied to each element of the vector. For sparse vectors multiplication with a scalar and negation are straightforward, since they do not alter the sparseness pattern of the vector. On the other hand, addition and subtraction may result in a vector having different characteristic bit sequence from either of the operand vectors. Consider two sparse vectors $\mathbf{x}$ and $\mathbf{y}$ with bit sequences $\mathbf{b}_{\mathbf{x}}$

and $\mathbf{b}_y$. Vector sum $\mathbf{x}+\mathbf{y}$ will have a characteristic bit sequence $\mathbf{b}_x$ OR $\mathbf{b}_y$. The dot product results a scalar. Applying an arbitrary function to a vector will usually yield a dense vector.

At this stage the function prototypes for the operators can be given. To begin with the easy ones, negation is just

```
template<unsigned int N>
CTSV<N> inline operator-(const CTSV<N>& a);
```

Multiplication with a scalar would be

```
template<unsigned int N>
CTSV<N> inline operator*(float a, const CTSV<N>& b);
```

The function prototype for the addition operation can be defined as

```
template <unsigned int N, unsigned int M>
CTSV<N|M>
inline operator+(const CTSV<N>& a, const CTSV<M>& b);
```

This template can be instantiated with two CTSVs of arbitrary bit sequences. The resulting type is a CTSV having a characteristic bit sequence formed by bitwise OR. The implementation of the `operator+` is developed further, since it covers most of the principles behind all the operator definitions. The implementation is recursive, the addition is performed on the heads of the vectors, and the tails of the operands are added recursively.

The `operator+` serves an interface to the actual addition operation, which is implemented as a static member function of a templatised class named `plus`. The template parameters of the plus class are the characteristic bit sequences of the operands of the addition. The code for `operator+` and the `plus` class is given in Listing 5.

```
template <unsigned int N, unsigned int M>
CTSV<N|M>
inline operator+(const CTSV<N>& a, const CTSV<M>& b) {
  CTSV<N|M> c;
  plus<N,M>::add(a,b,c);
  return c;
}

template<unsigned int N, unsigned int M>
class plus {
public:
  static inline void add(const CTSV<N>& a, const CTSV<M>& b, CTSV<N|M>& c)
  {
    add(a.value,b.value,c.value);
    plus< N>>1 , M>>1 >::add(a.GetTail(),b.GetTail(), c.GetTail());
  }
};

class plus<0,0> {
public:
  static inline void add(const CTSV<0>& a, const CTSV<0>& b, CTSV<0>& c){}
};
```

Listing 5. Operator+ and the plus class.

In the body of the `operator+`, an object of the resulting CTSV type is created with the empty constructor. The parameters to be added and the newly created object are passed to the function `plus<N,M>::add`. This function adds the heads of the vectors with the add function of the `Elem<N>` classes, to be defined below, and calls the `plus< N>>1 , M>>1 >::add` function recursively with the tails of the operands. The template parameters are shifted right along the recursion, leading eventually to a call to `plus<0,0>::add` function, which ends the recursion.

Here the necessity of the `GetTail` function can be seen. At some point of the recursion `plus<0,N>::add` or `plus<N,0>::add` may be called and we must be able to continue the recursion even if one of the input parameters is of type `CTSV<0>`. Since the `CTSV<0>::GetTail` returns the `CTSV<0>` object itself, we may continue to pass the 'tail' of this type of object too.

For a complete code for the addition of CTSVs, the add functions for the `Elem` classes are yet to be defined:

```
inline void
add(const Elem<0>& a, const Elem<0>& b, Elem<0>& c) {}

inline void
add(const Elem<0>& a, const Elem<1>& b, Elem<1>& c) { c.v = b.v; }

inline void
add(const Elem<1>& a, const Elem<0>& b, Elem<1>& c) { c.v = a.v; }

inline void
add(const Elem<1>& a, const Elem<1>& b, Elem<1>& c) { c.v = a.v + b.v; }
```

The resulting type is 'promoted' from `Elem<0>` to `Elem<1>` if an `Elem<1>` type object is involved in the operation. In this way the types are handled correctly. It is easy to see that the compilation of these definitions yields the optimal lines of code: for addition of two `Elem<0>` objects, no code is produced, the addition of `Elem<0>` and `Elem<1>` results in a single move and the addition of two `Elem<1>` objects in a single addition.

2.3. Generated code

To clarify how the above works, we examine the example of section 1.2. more closely. The definition of the vectors **x** = ((1,1), (1,3)) and **y** = ((2,3), (2,5)) is coded as

```
CTSV<5> x(1);
CTSV<20> y(2);
```

Note that $5 = (00101)_2$ and $20 = (10100)_2$. The addition of the vectors is just

```
x+y;
```

The resulting code compiled with Borland C++ 5.0 for Intel 80486 with no optimisations is shown in Listing 6.

The initialisations are just the two inevitable move commands for both **x** an **y**, both having two nonzero elements. In this example two moves (positions 1 and 5) and one addition (position 3) is needed to carry out **x** + **y**, which can be seen from the resulting code. The last six lines of the assembly language code originates from the return statement and the implicit invocation of the copy constructor. The resulted code is clearly very minimal.

From a logical point of view the object copy caused by the return statement seems unnecessary. The copy is performed, since the local object to be returned goes out of scope and must be copied to the return value. A clever compiler may however avoid this by optimising the local object out of existence, and create the object directly on the caller's stack. This is known as *return value optimisation.* (There is a syntactic extension to C++ supported by g++ that allows an explicit reference to the functions return value, which would avoid the extra copy).

Even without a compiler supporting this optimisation technique, it is however possible to circumvent the extra copy if we content ourselves with a slightly more awkward syntax. Instead of writing

```
CTSV<x.bits|y.bits> z = x + y;
```

we would have to write

```
CTSV<x.bits|y.bits> z;
plus<x.bits,y.bits>::add(x,y,z);
```

However, even with this latter syntax, the resulting type of the operation does not have to be calculated explicitly, the *bits* enumeration constant conveys this information.

```
// CTSV<5> x(1.0);
mov       dword ptr [ebp-12],1
mov       dword ptr [ebp-7],1
// CTSV<20> y(2.0);
mov       dword ptr [ebp-174],2
mov       dword ptr [ebp-169],2
// x + y;
mov       eax,dword ptr [ebp-12]   // 1. position
mov       dword ptr [ebp-192],eax
mov       edx,dword ptr [ebp-7]    // 3. position
add       edx,dword ptr [ebp-174]
mov       dword ptr [ebp-187],edx
mov       ecx,dword ptr [ebp-169]  // 5. position
mov       dword ptr [ebp-182],ecx

mov       eax,dword ptr [ebp-192]   // return
mov       dword ptr [ebp-208],eax
mov       edx,dword ptr [ebp-187]
mov       dword ptr [ebp-203],edx
mov       ecx,dword ptr [ebp-182]
mov       dword ptr [ebp-198],ecx
```

Listing 6. x + y compiled.

2.4. Computational speed

The speed of CTSV operations was compared with ordinary dense vectors and sparse vectors with indexing performed at runtime. The implementation of the dense vector was just a standard C vector. The implementation of the sparse vector contained a vector of values and a vector of indices. The speed of addition operation was measured with vectors of 10 elements long, the number of nonzero elements ranging from 0 to 10. The code was compiled with Borland C++ 5.01 for Pentium processor and optimised for speed. Since the optimiser was not capable of performing the return value optimisation, we used the latter syntax to avoid the extra copy constructor invocations. The sparse vectors were allocated statically, consequently the runtime penalty originates only from the indexing performed, not from memory allocation. Since the Borland compiler could not inline the addition operations of the dense and sparse vectors the CTSV addition was also outlined. The results are shown in Figure 1. The exact comparison of the execution speed in modern processors is somewhat complicated since the cost of memory access may vary quite significantly, and in fact the results were different depending on where the accessed variables were situated in the memory. To eliminate this effect as much as possible all variables were allocated from the stack just before the call of the addition operator.

The addition functions were called in a loop repeatedly and the operand parameters were passed by reference. The CTSV addition for 0 nonzero elements yielded no code, just a call to an empty function. Hence the cost of the loop and function call should approximately equal the cost of the CTSV 0 case.

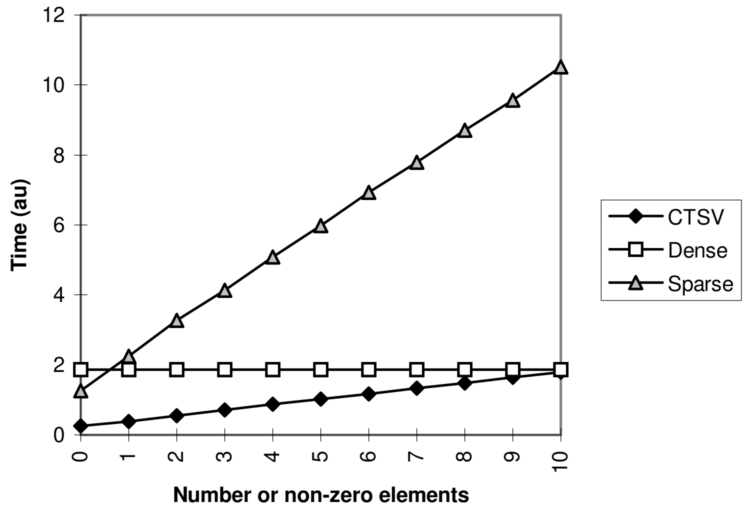

Figure 1. The CPU time of vector addition.

3. An application: automatic differentiation

The need for calculating partial derivatives arises in many applications, most notably in optimisation. Instead of symbolic calculation of the derivatives or using approximate difference values, *automatic differentiation* can be used to obtain derivatives. In automatic differentiation, the derivatives are computed by the well known chain rule, but instead of propagating symbolic functions, numerical values are propagated along the computation. The evaluation of the function and its derivatives are calculated simultaneously using the same expressions. There are several descriptions about automatic differentiation [2,5] and also software packages available [6]. Here, the concept is briefly described for understanding the benefits of CTSVs in this particular application.

Instead of computing with scalar values, we compute with multicomponent objects, consisting of a value of a function, and a vector of partial derivatives at the same point. For single derivatives, this object can be expressed as $< f, \frac{\partial f}{\partial x} >$. For several partial derivatives, say with respect to x, y and z, the object could be like $< f, (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}) >$. When building expressions with these multicomponent objects, at the leaf level of the expression tree the value of *f* is either a variable or a constant. For example when differentiating with respect to three variables x,y,z the constant 3.14 would be expressed as <3.14, (0, 0, 0)> and the variable y as <y, (0, 1, 0)>. Computation with these objects utilises the chain rule of derivatives:

$$g(< f, (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}) >) =< g(f), \frac{\partial g}{\partial f} \cdot (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}) >$$

As an example, consider the two derivative case for function $y + \sin(x^2)$. Starting with

$$< y, (0,1) > + \sin(< x, (1,0) >^2),$$

by squaring *x* we get

$$< y,(0,1) > + \sin(< x^2,(2x,0) >) .$$

Taking the sine gives

$$< y,(0,1) > + < \sin(x^2),(2x\cos(x^2),0) >$$

and finally the addition with y gives

$$< y + \sin(x^2),(2x\cos(x^2),1) >) .$$

For numerical work, the computation is not done symbolically, rather the actual values of the function and its derivatives are calculated and propagated through the expression. Given
x = 2, y = 4, the same example would be

$$\begin{aligned}
&< 4,(0,1) > + \sin(< 2,(1,0) >^2)\\
&< 4,(0,1) > + \sin(< 4,(2,0) >)\\
&< 4,(0,1) > + < 0.06976,(1.9951,0) >\\
&< 4.06976,(1.9951,1) > .
\end{aligned}$$

The method can be applied to any machine computable function. All that is needed is to code the differentiation rules for simple functions and operations. Then any function composed of those elementary functions can be differentiated automatically. In C++ this means overloading of common functions and operators for multicomponent objects described above. In the following, the above described multicomponent objects will be denoted ADN (automatically differentiable numbers).

The derivative vectors of ADNs are typically sparse. This is because the arity of the elementary functions are low, usually from 0 to 2. The example above enlightens this feature: only the last step yields a dense vector. Typically there is only one nonzero derivative at the leaves of the expression tree, and the derivative vector becomes more dense when approaching the root. It is therefore beneficial to utilise sparse vector techniques in ADN computations to avoid extra calculations. However, the indexing in sparse vectors requires some work too; we have to locate the nonzero elements and perform correct operations within each subexpression. With the use of CTSVs, this indexing work can be avoided.

3.1. Automatic differentiation with CTSVs

We will first have to define the ADN class. To use CTSVs as the derivative vector, we define CTSV as a data member of the ADN class. The ADN class will therefore also be templatised with the characteristic bit sequence of the derivative vector. The common functions and operations must be overloaded to accept ADN objects. Listing 7 gives an outline of a possible class definition for ADNs. The constructor with one variable is for creating an ADN object with a given value and an initial derivative value 1. The addition operator and the sine function are given as an example in Listing 8.

```
template <int N>
class ADN
public:
  float value;
  CTSV<N> ders;
  ADN() {};
  ADN(float x) : value(x), ders(1) {}
}
```

Listing 7. Class definition for automatically differentiable numbers.

To declare ADNs, one has to assign a certain vector element position for each variable. This is done by using characteristic bit sequence containing single one bit. E.g. `ADN<1>, ADN<2>, ADN<4>` and so forth. Constants can be declared as `ADN<0>` type objects, having only the value member and an empty derivative vector.

```
template<int N, int M>
inline ADN<N|M> operator+(const ADN<N>& a, const ADN<M>& b)
{
  ADN<N|M> tmp;
  tmp.value = a.value + b.value;
  plus<N,M>::(a.ders, b.ders, tmp.ders);
  return tmp;
};

template <int N>
ADN<N> sin(const AND<N>& a)
{
  ADN<N> tmp;
  tmp.value = sin(a.value);
  tmp.ders = cos(a.value) * a.ders;
}
```

Listing 8. Addition operator and sin function for ADNs.

To demonstrate the use of ADNs the compilation of the code

```
ADN<1> f(2);
ADN<2> g(3);
f+g;
```

is shown in the Listing 9. Again the copy constructor invocation could be avoided with a clever optimiser or by using a similar less convenient syntax than with CTSVs.

```
// ADN<1> f(2);
mov       dword ptr [ebp-36],1073741824
mov       dword ptr [ebp-32],1
// ADN<2> g(3);
mov       dword ptr [ebp-48],1077936128
mov       dword ptr [ebp-43],1
// f + g;
mov       ecx,dword ptr [ebp-32]  // move the derivative of f
mov       dword ptr [ebp-60],ecx
mov       eax,dword ptr [ebp-43]  // move the derivative of g
mov       dword ptr [ebp-56],eax
fld       dword ptr [ebp-36]      // add the function values
fadd      dword ptr [ebp-48]
fstp      dword ptr [ebp-64]
mov       edx,dword ptr [ebp-64]    // return (copy constructor)
mov       dword ptr [ebp-280],edx
mov       ecx,dword ptr [ebp-60]
mov       dword ptr [ebp-276],ecx
mov       eax,dword ptr [ebp-56]
mov       dword ptr [ebp-272],eax
```

For a proper implementation of ADNs, some additional operations for CTSVs are needed. It is worthwhile to define a special function for the chain rule of multiplication performing the `f.value * g.ders + g.value * f.ders` in a single pass of the vectors. The same applies for the division of ADNs. Conversion functions from floating point numbers to `ADN<0>` type would also be desirable, as well as operations for accessing individual elements of the vector. These are not shown, instead we discuss a further refinement to CTSVs, which will avoid unnecessary computations in ADN expressions.

3.2. Refined CTSVs with automatic differentiation

At the leaf level of the expression tree, the derivatives are either 0 or 1. There will probably be a considerable amount of multiplications with unity. Consider e.g. the ADN expression

$$< x,(1,0) > \cdot < y,(0,1) > .$$

When computed it will produce

$$< x \cdot y,(y \cdot 1 + x \cdot 0, y \cdot 0 + x \cdot 1) > .$$

With CTSVs the multiplications with 0 are not performed, but the expression will nevertheless yield three multiplications. In the case of symbolic differentiation only the multiplication for calculating the function value is needed since the multiplications with unity does not have to be performed. The same behaviour can be achieved with ADNs by keeping track of the positions of elements containing ones. Instead of having zero and nonzero elements, we would then have *zero, unity* and *arbitrary* elements. There is of course no point to keep track of the unity elements at runtime just to be able to replace a few multiplications with value moves. But since with CTSVs there is no runtime penalty for the indexing, this can be done and will in some cases produce significant savings in computation time. This is especially the case when the vector elements are complex numbers, the complex multiplication being rather expensive operation.

Since we now distinguish between three types of elements, there must be three specialisations of the `Elem` class, best `Elem<0>`, `Elem<1>`, `Elem<3>`. Hence two bits of the characteristic bit sequence of the CTSV are needed for each element. The elementary operations between different `Elem` classes must be redefined and the bit operations of the characteristic bit sequences of CTSVs will become somewhat trickier, but no other changes are required. The complete code for ADNs with refined CTSVs can be obtained along this report from [7].

3.3. Test runs

Some comparisons were performed with ADNs, objects having the derivative vector implemented as refined CTSVs, ordinary dense vectors and ordinary sparse vectors. Also manually optimised (all common subexpressions were stored to local variables and calculated only once) symbolically differentiated code was included in the tests. Again there was no runtime penalty for dynamic memory allocation with ordinary sparse vectors. To overcome the current compiler limitations the non expression syntax was used. Results of the test runs are shown in Figures 2 and 3.

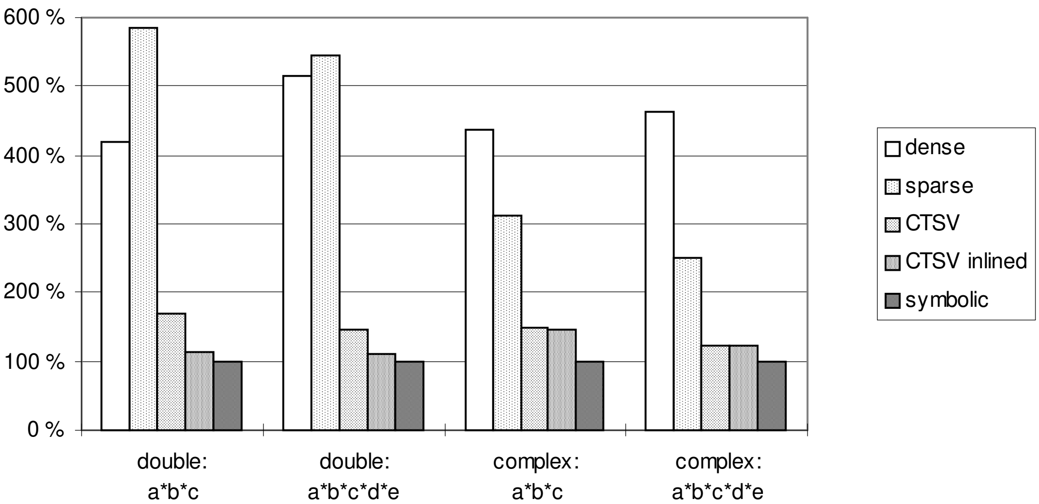

Figure 2. CPU time (in arbitrary units) of the calculation of expressions a*b*c and a*b*c*d*e and their derivatives with respect to all variables.

To make an objective comparison, the expression was calculated in an outlined function, each parameter of the expression was passed to the function as a separate reference parameter and the function value with the derivatives was returned as a reference to an ADN object. In all cases the number of parameters of the test function was equal. The test function was then called in a loop repeatedly. The code was optimised for speed.

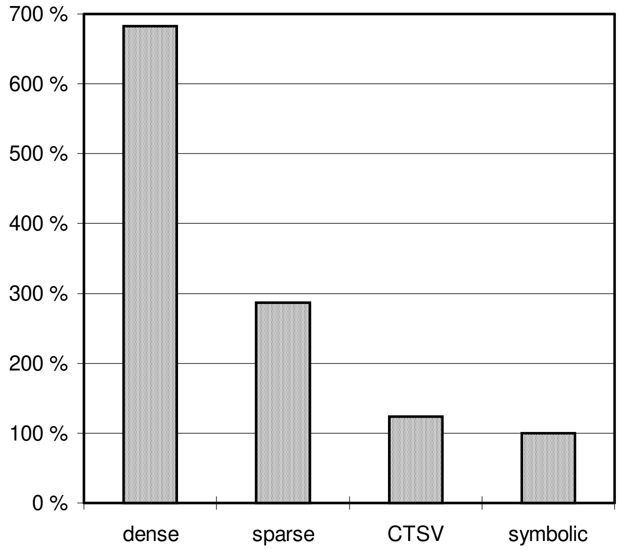

Figure 3. CPU time of the calculation of expression $\sum_{i=1}^{3} a_i e^{b_i t}$ and its derivatives with respect to a_i and b_i for $i = 1,2,3$.

4. Discussion

A collection of classes for representing sparse vectors in C++ was described. The precondition for the use of the presented classes is that the sparseness pattern of the vector is known at compile time. For special applications where the rather restrictive precondition is met, very efficient code can be written. It was shown how common vector operations can be overloaded for the presented CTSV classes to yield this minimal code for abstract vector expressions.

The classes rely heavily on C++ template usage, especially on recursive template definitions with non-type template parameters. The sparseness pattern of each vector is presented as a template parameter of the vector, i.e. in the type of the vector. The sparseness pattern is altered along operations with vectors, and each operation will potentially produce a new vector type. These new type instances are automatically generated from the vector templates by the compiler as encountered. Some of the template features used are quite new in C++ and are currently not available at all compilers. These features are however part of the current ANSI/ISO standard proposal for C++ and will therefore probably be implemented by major compiler vendors soon.

Execution of selected vector operations was compared with dense vectors and ordinary sparse vectors and the execution speed clearly outperforms both alternatives. The execution speed depends to some extent on the compilers optimisation capabilities; for best results the compiler ought to be able to perform return value optimisation.

Automatic differentiation was presented as an application of CTSVs. In automatic differentiation of expressions, derivatives are calculated along the function value with the chain rule of differentiation. The calculations are performed with multicomponent objects containing the function value and a vector of derivatives. The class definitions for such automatically differentiable numbers were given using CTSVs as the derivative vectors and it was shown how common mathematical operations and functions can be overloaded for these classes. Computational speed of the evaluation of some common expressions was compared with ADNs having alternative vector representations. With our examples ADNs with CTSV derivatives required CPU time ranging from 20 to 50 % less than ADNs with alternative vector representations. The execution speed was very near the speed of calculating the symbolic derivatives.

References

[1] T. Veldhuizen, Using C++ template metaprograms, C++ Report, Vol. 7, No. 4, May 1995, pp. 36-43.

[2] L. B. Rall, Automatic differentiation: Techniques and Applications, Lecture Notes in Computer Science 120, Springer-Verlag, Berlin 1981.

[3] 1997 C++ Public Review Document: Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++, ANSI X3J16/96-0225 ISO WG21/N1043.

[4] W.H.Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes in C: the Art of Scientific Computing, 2nd edn., Ch. 2 (Cambridge University Press, New York, 1992)

[5] J. J. Barton, L.R. Nackman, Automatic differentiation, C++ Report, Vol. 8, No. 2, February 1996, pp. 61-63.

[6] A. Griewank, D. Juedes, J. Utke, ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++, ACM Transactions on Mathematical Software, Vol. 22, No. 2, June 1996, pp- 131-167.

[7] http://www.tucs.abo.fi/publications

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi/

University of Turku
• Department of Mathematical Sciences

Åbo Akademi University
• Department of Computer Science
• Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
• Institute of Information Systems Science