SYMBOLIC COMPUTATIONS IN DIFFERENTIAL GEOMETRY

DIEGO CONTI

Abstract. We introduce the C++ library Wedge, based on GiNaC, for symbolic computations in differential geometry. We show how Wedge makes it possible to use the language C++ to perform such computations, and illustrate some advantages of this approach with explicit examples. In particular, we describe a short program to determine whether a given linear exterior differential system is involutive.

Keywords: Curvature, differential forms, computer algebra, exterior differential systems.

Introduction

There are many computationally intensive problems in differential and Riemannian geometry that are best solved by the use of a computer. Due to the nature of these problems, any system meant to perform this type of calculations must be able to manipulate algebraic and differential expressions. For this reason, such systems are generally implemented as extensions, or packages, for a general purpose computer algebra system: the latter takes care of handling expressions, and the extension introduces the differential-geometry specific features — differential forms, tensors, connections and so on. Examples are given by the packages difforms and GRTensor (see [9]) for Maple, or the Ricci package for Mathematica.

A remarkable consequence of this approach is that one is essentially limited, when implementing one’s own algorithms, to using the programming language built in the computer algebra system. Some drawbacks and limitations of these programming languages are described in [1], where an alternative is also introduced, namely the C++ library GiNaC. As suggested by the name (an acronym for GiNaC Is Not A CAS), GiNaC differs from the above mentioned computer algebra systems in that it is based on a general purpose, well-established programming language such as C++, rather than introducing a new one. In particular, development, debugging and documentation of a program based on GiNaC can take advantage of the many tools commonly available to a C++ programmer. This makes GiNaC a natural choice when implementing new, complex algorithms, like the one introduced in [5], which provided the original motivation for the present work.

In this paper we introduce Wedge, an extension to GiNaC that can be used to write a C++ program that performs computations in differential and Riemannian geometry. Wedge is able to perform algebraic or differential computations with differential forms and spinors, as well as curvature computations in an adapted frame, and contains some support for vector spaces, represented in terms of a basis. Bases of vector fields, or frames, play a central rôle in Wedge: in particular, the tangent space of a manifold is represented by a frame, and a Riemannian metric

2000 Mathematics Subject Classification. Primary 53-04; Secondary 53C05, 68W30, 58A15.
is represented by a (possibly different) orthonormal frame. Notice that the above-
mentioned package GRTensor also supports working with adapted frames, but our
approach differs in that the frame need not be defined in terms of coordinates; this
can be useful when working on a Lie group, where the geometry is defined natu-
really by the structure constants, or on the generic manifold with a fixed geometric
structure (see Section 2).

Another unique feature of Wedge among packages for differential and Riemann-
ian geometry, beside the choice of C++, is the fact that it is completely based on
free, open-source software. Like GiNaC, Wedge is licensed under the GNU general
public license; its source code is available at http://libwedge.sourceforge.net

This paper is written without assuming the reader is familiar with C++, and its
purpose is twofold: to introduce the main functionality of Wedge, and to illustrate
with examples certain features of C++ which can prove very helpful in the practice
of writing a program to perform some specific computation.

In the first section we introduce some basic functionality, concerning differential
forms and connections; at the same time, we illustrate classes and inheritance.

In the second section we introduce spinors and “generic” manifolds; at the same
time, we illustrate object-oriented programming.

In the third section we explain briefly how GiNaC handles expressions, then
introduce bases and frames, namely the linear algebra features of Wedge.

In the final section we give an application from Cartan-Kähler theory, with a
short program that reproduces the computations of [3] that prove the local existence
of metrics with holonomy $G_2$.

1. Working with differential forms and connections

A fundamental feature of C++ is the possibility of introducing user-defined types,
i.e. classes (or structs). The definition of a class specifies not only the type of data
contained in a variable which has that class as its type, but also some operations
that can be performed on this data. This is generally better than having global
functions which either take a long list of arguments or use global variables, and this
can be seen, for instance, in situations where more sets of data appear in the same
program. In this section we shall illustrate this point with an example, considering
a problem concerning multiple connections on a fixed manifold. In the course of
the section, we shall introduce some of the essential functionality of Wedge.

The basic geometric entity in Wedge is the class Manifold. A variable of type
Manifold represents a manifold in the mathematical sense, which is assumed to be
parallelizable, and represented by a global basis of one-forms $e^1,\ldots,e^n$, with dual
basis of vector fields $e_1,\ldots,e_n$. This assumption is tailored on Lie groups, but one
can also think of a Manifold object as representing a coordinate patch; also, every
manifold can be written as a quotient $M/G$ of a parallelizable manifold, so one
can always reduce to the parallelizable case. For instance, one can compute the
curvature of a Riemannian metric on $M/G$ applying the O’Neill formula (see [10]).

As an example, we shall consider the nilpotent Lie group $X$, characterized by
the existence of a global basis of one-forms $e^1,\ldots,e^4$ such that

$$
de^1 = 0, \quad de^2 = 0, \quad de^3 = e^{12}, \quad de^4 = e^{13}.
$$
In the above equation, $e^{23}$ stands for the wedge product $e^2 \wedge e^3$, and so on; this shorthand notation is also used by Wedge, and so it will appear throughout the paper in both formulae and program output. Equations (1) translate to the following code:

```cpp
struct X : public ConcreteManifold, public Has_dTable {
    X() : ConcreteManifold(4) {
        Declare_d(e(1),0);
        Declare_d(e(2),0);
        Declare_d(e(3),e(1)*e(2));
        Declare_d(e(4),e(1)*e(3));
    }
};
```

The first line means that $X$ inherits from the two classes `ConcreteManifold` and `Has_dTable`. Roughly speaking, this means that it inherits the functionality implemented by these two classes; inheritance defines a partial ordering relation, and is best represented by a graph (see Figure 1). Specifically, inheriting from `ConcreteManifold` ensures that the forms $e^1, \ldots, e^4$ are defined (where the dimension 4 appears as a parameter on the second line). The fact that $X$ inherits from `Has_dTable` means that the operator $d$ on the manifold $X$ is known in terms of its action on the basis $e^1, \ldots, e^4$; the relations (1) are given with calls to the member function `Declare_d`. These calls appear in the body of the constructor of $X$, which is invoked automatically when a variable of type $X$ is constructed. We can now instantiate a variable of type $X$ and perform computations with it, e.g.

```cpp
X M;
cout<<M.d(M.e(4))<<endl;
```

has the effect of printing $e^{13}$. Notice that both the forms $e^i$ and the operator $d$ are implemented as members of $X$; this means that every variable of type $X$ has its own set of forms $e^1, \ldots, e^n$ and $d$ operator, and so the code must tell the compiler which $X$ should be used. This is achieved here by the "M." appearing in the function calls; however, inheritance provides an equivalent, more appealing alternative, as in the following:
struct SomeCalculations : X {
    SomeCalculations() {cout<<d(e(4))<<endl;}
} M;

The class Manifold also implements the Lie bracket and Lie derivative as member functions, whilst the exterior and interior product of forms are implemented by global functions. Covariant differentiation and curvature computations are accounted for by the class Connection, which we now introduce with an example.

Every almost-complex manifold \((M, J)\) admits an almost complex connection \(\tilde{\nabla}\) such that its torsion form \(\Theta\) and the Nihenjuis tensor are related by

\[16\Theta(X, Y) = N(X, Y);\]

by \([8]\) \(\tilde{\nabla}\) can be obtained from an arbitrary torsion-free connection \(\nabla\) by

\[
\tilde{\nabla}XY = \nabla XY - Q(X, Y), \quad 4Q(X, Y) = (\nabla JY)X + J((\nabla Y)X) + 2J((\nabla X)Y).
\]

Suppose one wants to compute such a connection in the case of our nilpotent Lie group \(X\), with the almost-complex structure determined by

\[J(e_1) = e_2, \quad J(e_3) = e_4.\]

We can do so with the following code:

```cpp
struct AlmostComplex : public X {
    TorsionFreeConnection<true> h;
    ex J(ex Y) {return Hook(Y,e(1)*e(2)+e(3)*e(4));}
    ex A(ex X,ex Y) {
        return h.Nabla<VectorField>(X,J(Y))-J(h.Nabla<VectorField>(X,Y));
    }
    ex Q(ex X, ex Y) {
        return (A(J(Y),X)+J(A(Y,X))+2*J(A(X,Y)))/4;
    }
    AlmostComplex() : omega(this,e()) {
        Connection k(this,e());
        for (int i=1;i<=4;++i)
            for (int j=1;j<=4;++j)
                k.DeclareNabla<VectorField>(e(i),e(j),
                h.Nabla<VectorField>(e(i),e(j))-Q(e(i),e(j)));
        cout<<k<<k.Torsion();
    }
} M;
```

This code defines and instantiates a struct named AlmostComplex inheriting from X; AlmostComplex has a data member \(h\) of type TorsionFreeConnection<true>, which represents a generic torsion-free connection \(\nabla\) on \(X\). The constructor uses \(h\) to compute another connection \(k\), corresponding to \(\tilde{\nabla}\) in \([8]\), then prints the connection forms of \(\tilde{\nabla}\) and its torsion form \(\Theta\). The latter is a vector

\[
(0, \quad 0, \quad -\frac{1}{4}e^{32} - \frac{1}{4}e^{41}, \quad \frac{1}{4}e^{42} - \frac{1}{4}e^{31})
\]

consistently with the fact that the almost complex structure \(J\) is not integrable.

There are two connection objects appearing here, both represented internally by their connection forms in terms of the frame \(e_1, \ldots, e_n\). On construction, they are
initialized in terms of symbols $\Gamma_{ijk}$, representing the functions

$$\Gamma_{ijk} = \langle \nabla_{e_i} e_j, e_k \rangle, \quad \tilde{\Gamma}_{ijk} = \langle \tilde{\nabla}_{e_i} e_j, e_k \rangle,$$

where $\langle , \rangle$ represents the pairing on $TM \otimes T^*M$. Since the type of $h$ is $\text{TorsionFreeConnection<true>}$, the constructor of $h$ ensures that the torsion is zero by solving the linear equations in the $\Gamma_{ijk}$ given by

$$(3) \sum_i e^i \wedge \nabla_{e_i} e^j - de^j = 0.$$

Of course this does not determine $\nabla$ uniquely, and some of the $\Gamma_{ijk}$ remain as parameters. The conditions (2) on the connection $k$ are imposed directly by calling the member function $\text{DeclareNabla}$. The tensor $Q$ is computed by the function $Q$, which in turn calls $\text{Nabla}$; since the latter is a member function, the connection $h$ is used to compute the covariant derivative. Thus, the connection forms of $k$ depends on the $\Gamma_{ijk}$, although the torsion $\Theta$ is independent of the $\Gamma_{ijk}$, as expected from the theory.

Another important point is that $\text{DeclareNabla}$ does not assign values directly to the connection forms. Rather, it uses GiNaC’s function $\text{lsolve}$ to solve a system of equations, linear in the $\Gamma_{ijk}$, and substitutes the solution into the connection forms. This procedure is more flexible than giving the list of the $\Gamma_{ijk}$ for two reasons: the arguments passed to $\text{DeclareNabla}$ need not be elements of the frame (they can even be forms of degree higher than one, or, for Riemannian connections only, spinors), and part of the connection forms may be left unspecified. However, said procedure only works because the $\Gamma_{ijk}$ are stored in the $\text{Connection}$ object, which is how $\text{DeclareNabla}$ knows it should leave the $\Gamma_{ijk}$ as parameters, and solve with respect to the $\tilde{\Gamma}_{ijk}$.

Summing up, we have illustrated how the fact that a connection is represented as an instance of a class makes working with two different connections on a fixed manifold as natural as defining two variables of the same type.

2. **Torsion free connections and generic manifolds**

Class $X$ from Section 1 represents a fixed manifold, where the action of the $d$ operator can be recovered from its action on a basis of one-forms; we have seen that Wedge enables one to perform torsion computations, or impose torsion conditions. One can also go the other way, and compute the action of $d$ in terms of the covariant derivative, with respect to a torsion-free connection. To illustrate this, suppose one has a four-dimensional Riemannian manifold $X$, with a global orthonormal frame $e^1, \ldots, e^4$. The parallelism induces a spin structure; in general, on a parallelizable $n$-dimensional manifold a spinor can be viewed as a map $X \to \Sigma$, where $\Sigma$ is the $n$-dimensional spinor representation. Let $m = [n/2]$ be the integer part of $n/2$; $\Sigma$ is a complex vector space of dimension $2^m$, represented in Wedge in terms of the basis $u_0, \ldots, u_{2^m-1}$, where $u_k$ corresponds to

$$u((-1)^{r_{m-1}} \ldots, (-1)^{r_0}), \quad k = \sum_{r=0}^{m-1} (1 - e_r)2^r$$

in the notation of [2] (see also [6] for more explicit formulae). In our case, we can declare that the constant spinor $\psi: X \to \Sigma$, $\psi \equiv u_0$ is parallel, i.e.

$$\nabla_{e_i} \psi = 0 \quad i = 1, \ldots, 4$$
with the following code:

```cpp
struct X : public ManifoldWith<RiemannianStructure> {
    X() : ManifoldWith<RiemannianStructure>(4) {
        for (int i=1;i<=4;++i)
            DeclareNabla<Spinor>(e(1),u(0),0);
    }
};
```

Hence, $X$ is a generic parallelizable Riemannian 4-manifold with holonomy contained in $SU(2)$; this condition does not determine the connection form uniquely, but it does impose certain conditions on them. Internally, the template class `ManifoldWith<RiemannianStructure>` contains a member of type `LeviCivitaConnection<false>`, which behaves in a similar way to its counterpart `TorsionFreeConnection<true>` from Section 1, with two differences: first, it represents the Levi-Civita connection, which is both torsion-free and Riemannian, and secondly it does not use the operator $d$ of the manifold to impose the condition $d^2 = 0$. Instead, `ManifoldWith` uses the Levi-Civita connection to compute the action of $d$ via (3).

**Convention.** All further code fragments appearing in this section will be assumed to appear in the body of the constructor of a class such as `SomeCalculations` of Section 1.

In fact, whilst the new $X$ does not derive from `has_dTable`, the definition of `SomeCalculations` of Section 1 is still legitimate. Of course, the output will depend on the $\Gamma_{ijk}$ which in part are left unspecified. In particular, it is not possible to guarantee that $d^2 = 0$, as this leads to an underdetermined system of differential equations, quadratic in the $\Gamma_{ijk}$ and linear in their derivatives. However, there are many interesting relations not involving the derivatives of the $\Gamma_{ijk}$ that can be proved using `ManifoldWith`. As a first example, we can verify that the existence of a parallel spinor is equivalent to the existence of a local frame $e^1, \ldots, e^4$ such that the forms

$$
(4) \quad e^{12} + e^{34}, \quad e^{13} + e^{42}, \quad e^{14} + e^{23},
$$

are closed (see [7]). In our case, since we have chosen $u_0$ as the parallel spinor, the standard frame $e^1, \ldots, e^4$ satisfies this condition, and the code

```cpp
cout<<d(e(1)*e(2)+e(3)*e(4))<<endl;
cout<<d(e(1)*e(3)+e(4)*e(2))<<endl;
cout<<d(e(1)*e(4)+e(2)*e(3))<<endl;
```

prints three times zero. One can obtain the opposite implication by invoking `Declare_d` in the constructor of $X$ to impose that the forms (4) are closed.

Compared to the examples of Section 1, this code shows an important feature of C++ which is common to many programming languages, but not so common among computer algebra systems: one can define different functions with the same name, and the compiler is responsible for selecting the correct one based on the context. In this case the functions $d$ and `Declare_d` appear as members of different classes, i.e. `Has_dTable` and `ManifoldWith`, and so the meaning of a call to $d$ in the body of a member of $X$ depends on whether $X$ inherits from `Has_dTable`, as was the case in Section 1 or `ManifoldWith`, as in the example above. In fact, C++ allows even greater flexibility, as we now illustrate with a second example.
In order to give a slightly more complicated application, we observe that our
manifold \( X \) is in particular an almost-Kähler manifold with respect to any one of
the closed two-forms \((4)\); we shall fix the first one. Thus, it makes sense to consider
the bilagrangian splitting \( TX = F \oplus G \), where \( F = \langle e_1, e_3 \rangle \) and \( G = \langle e_2, e_4 \rangle \); this
splitting determines a canonical connection \( \omega \). By \cite{11}, the torsion of \( \omega \) is zero if and only if the distributions \( F \) and \( G \) are integrable. We can actually prove this
equivalence with Wedge; we shall illustrate the “only if” implication here.

\begin{verbatim}
Connection omega(this,e(),"Gamma");
for (int k=1;k<=4;++k) {
    omega.DeclareNabla<DifferentialForm>(e(k),e(1)*e(2)+e(3)*e(4),0);
    for (int i=1;i<=4;++i)
        for (int j=i%2+1;j<=4;j+=2)
            omega.DeclareZero(Hook(e(j),
                omega.Nabla<DifferentialForm>(e(k),e(i))));
}
for (int k=1;k<=4;++k)
    for (int i=k%2+1;i<=4;i+=2)
        for (int j=k%2+1;j<=4;j+=2)
            omega.DeclareZero(Hook(e(j),
                omega.Nabla<VectorField>(e(k),e(i))-LieBracket(e(k),e(i))));
exvector T=omega.Torsion();
DeclareZero(T.begin(),T.end());
cout<<LieBracket(e(1),e(3))<<","<<LieBracket(e(2),e(4))<<endl;
\end{verbatim}

This code defines a generic connection \( \omega \), whose connection parameters are de-
noted by \( \Gamma'_{ijk} \) to distinguish them from the parameters \( \Gamma_{ijk} \) of the Levi-Civita
connection, and imposes the two sets of conditions that determine the symplectic
connection. First, the holonomy is reduced to \( GL(2,\mathbb{R}) \) by requiring that the sym-
plectic form be parallel and the two Lagrangian distributions \( F \) and \( G \) preserved
by the covariant derivative. Then one imposes
\[
\nabla_X Y = [X_F, Y_G]_F, \quad \nabla_X [Y_F] = [X_G, Y_F]_F, \quad X, Y \in \Gamma(TX),
\]
where the subscripts denote projection (reflected in the code by the use of the in-
terior product function \( \text{Hook} \)). Notice that \( \text{LieBracket} \) is a member function of
\text{Manifold}, which \text{ManifoldWith} reimplements in terms of the Levi-Civita connection;
thus, its result depends on the \( \Gamma_{ijk} \). By the general theory, the conditions
determine \( \omega \) completely, i.e. \( \omega \) does not depend on the \( \Gamma_{ijk} \) but only on the \( \Gamma_{ijk} \).
Since this dependence is linear, the condition that the torsion is zero gives equa-
tions in the \( \Gamma_{ijk} \) that can be solved by the function \text{DeclareZero}, a member of
\text{ManifoldWith}. Having imposed this torsion conditions, the program concludes
that \( [e_1, e_3] \) and \( [e_2, e_4] \) equal
\[
e_1\Gamma_{142} + e_3\Gamma_{331} + e_4\Gamma_{431} + e_2\Gamma_{231}
\]
respectively, proving that the distributions \( F \) and \( G \) are involutive.

The code at work here uses a typical object-oriented programming technique.
Specifically, \text{Connection} needs to compute the action of \( d \) in order to compute the
torsion; to this end, every \text{Connection} object contains a pointer to the \text{Manifold}
object it refers to. In this case, the manifold to which \( \omega \) refers is represented
by a \text{ManifoldWith} object, which implements \text{d} using the Levi-Civita connection;
since $d$ is a virtual function, the calls to $d$ performed by Torsion execute the implementation of ManifoldWith. In the examples of Section 1 those same calls execute the implementation of Has_dTable. This ensures that the torsion of $\omega$ is computed correctly in terms of the $\Gamma_{ijk}$ here, whereas the code of Section 1 computes the torsion using the action of $d$ on the forms $e^1, \ldots, e^4$, although the function Torsion is exactly the same.

Remark. There is nothing essential about the assumption that $X$ has holonomy SU(2). In fact, one can easily modify the code to obtain the same result for a generic symplectic 4-manifold $X$.

3. Forms and frames

In this section we introduce the main linear algebra functionality of Wedge; before doing that, we need to explain how GiNaC handles expressions.

We have seen in Section 1 that manifolds are represented in Wedge by a global basis of one-forms $e^1, \ldots, e^n$, which can be accessed via a member function of the class Manifold. The C++ type of a form, say $X.e(1)$, is the class ex defined in the library GiNaC, which handles expressions such as $2*e(1)+e(2)$, the result of which is still an ex. The class ex acts as a proxy to the GiNaC class basic. In practice, this means that an ex contains the address of an object whose type is a class that inherits from basic (e.g. add representing a sum, or numeric representing a number), and it is this object that performs the actual computations. The use of virtual functions ensures that the correct code is used, depending on the C++ type of the object the ex points to (for details, see [1]).

Thus, while $X.e(1)$ has type ex, under the hood it “refers” to an object of type DifferentialOneForm, and it is this type that implements skew-commutativity, making use of a canonical ordering that exists among basic objects. More significantly, objects of type DifferentialOneForm and linear combinations of them can be used interchangeably; nonetheless, they can be distinguished by their internal representation, since the former have type DifferentialOneForm and the latter have type add (e.g. $e(1)+e(2)$) or mul (e.g. $2*e(1)$). We refer to objects in the former category as simple elements. Mathematically, we are working with sparse representations of elements in the vector space generated by all the variables of type DifferentialOneForm appearing in a program. All this works equally well with other types than DifferentialOneForm.

In general, given a small set of vectors, it is possible to switch from sparse to dense representation by extracting a basis, and writing the other terms in components. This is accomplished by the class template Basis. As a simple example, suppose $X$ defines the Iwasawa manifold, characterized by the existence of an invariant basis $e^1, \ldots, e^6$ satisfying

$$de^1 = 0 = de^2 = de^3 = de^4, \quad de^5 = e^{13} + e^{42}, \quad de^6 = e^{14} + e^{23}.$$ 

We can compute a basis of the space of invariant exact three-forms with the following code (with the usual convention of p. 6):

```cpp
Basis<DifferentialForm> b;
for (int i=1; i<=6; ++i)
    for (int j=i+1; j<=6; ++j)
        b.push_back(d(e(i)*e(j)));
cout<<b;
```
resulting in the output
\[ e^{412}, -e^{312}, -e^{342}, e^{341}, e^{642} - e^{631} - e^{352} + e^{415}. \]

One can then recover the components of, say, \( de^{45} \) by
\[
cout<<b.Components(d(e(4)*e(5))); \]

obtaining the vector \((0 \ 0 \ 0 \ -1 \ 0)\).

Another use of \texttt{Basis} is related to the identification of each vector space with its dual. Indeed, the existence of simple elements enables us to establish a canonical pairing, which is bilinear and satisfies
\[
\langle \alpha, \beta \rangle = \begin{cases} 1 & \alpha = \beta \\ 0 & \alpha \neq \beta \end{cases}, \quad \alpha, \beta \text{ simple.}
\]

In particular, we can identify vector fields with one-forms, by interpreting the pairing \( \langle, \rangle \) as the interior product. For instance, the standard frame \( e^1, \ldots, e^n \) associated to a \texttt{ConcreteManifold} always consists of simple elements, and so in this case the form \( e^i \) and the vector field \( e_i \) are represented by the same element \( e(i) \). However, this fact only holds for \( \langle, \rangle \)-orthonormal bases.

\texttt{Basis} is based on the standard container \texttt{vector}, part of the Standard Template Library. So, a \texttt{Basis} object can be thought of as a sequence \( x_1, \ldots, x_n \) of \texttt{ex}, where the \( x_k \) are linear combinations of simple elements of a given type \( T \), which is a template parameter of \texttt{Basis}. However, \texttt{Basis} differs from \texttt{vector} in the following respects:

- When elements are added to a \texttt{Basis} object, resulting in a sequence of possibly dependent generators \( x_1, \ldots, x_n \), an elimination scheme is used to remove redundant (dependent) elements. This is done in such a way that the flag \( V_1 \subseteq \ldots \subseteq V_n \),
  \[
  V_k = \text{Span}\{x_1, \ldots, x_k\}
  \]
  is preserved.

- The first time the member functions \texttt{Components} or \texttt{dual} are called, \texttt{Basis} sets up some internal data according to the following procedure. First, the set
  \[
  S = S(x_1, \ldots, x_m) = \{ \alpha \text{ simple} \mid \langle \alpha, x_k \rangle \neq 0 \text{ for some } 1 \leq k \leq m \}
  \]
  is computed; then, the sequence \( \{x_1, \ldots, x_m\} \) that represents the basis internally is enlarged with elements \( x_k \in S, \ m < k \leq n \) so that
  \[
  \{x_1, \ldots, x_n\} \text{ is a basis of } \text{Span} \ S.
  \]

Then, for each \( \alpha \in S \) the components \( b_{\alpha j} \) satisfying
\[
\alpha = \sum_{j=1}^{n} b_{\alpha j} x_j
\]
are computed; as a matrix, \( (b_{\alpha j}) \) is the inverse of the matrix with entries \( a_{j\alpha} = \langle x_j, \alpha \rangle \). The dual basis \( x^1, \ldots, x^n \) is then given by

\[
x^k = \sum_{\alpha \in S} b_{\alpha k} \alpha.
\]

The actual code also take advantage of the fact that with a suitable ordering of the columns, one obtains the block form

\[
(a_{j\alpha}) = \begin{pmatrix}
* & * \\
0 & I_{n-k}
\end{pmatrix}.
\]

The dual basis and the \( b_{\alpha j} \) are cached internally, until the basis is modified.

- The member function \texttt{Components} uses the pairing \( \langle \cdot, \cdot \rangle \) to determine the components with respect to \( S \), then multiplies by the matrix \( (b_{\alpha j}) \) to obtain the components with respect to the basis \( x_1, \ldots x_m \).

From the point of view of performance, the operations of \texttt{Basis} have the following complexity, measured in term of the number of multiplications:

- Given a list \( x_1, \ldots, x_n \) of elements which are known to be linearly independent, one can construct a \texttt{Basis} object with no overhead over \texttt{vector} and no algebraic operation involved.

- Inserting elements \( y_1, \ldots, y_k \) in a \texttt{Basis} object consisting of elements \( x_1, \ldots, x_n \) has a complexity of \( O(m(n + k)^2) \), where

\[
m = |S(x_1, \ldots, x_n, y_1, \ldots, y_k)|.
\]

- Setting up a \texttt{Basis} object consisting of elements \( x_1, \ldots, x_n \) in order that the dual basis or components of a vector may be computed has a complexity of \( O((n + m)n^2) \), where

\[
m = |S(x_1, \ldots, x_n)|.
\]

- Once the \texttt{Basis} object is so set up, computing the components of a vector has complexity \( O(nm) \).

Since we work in the symbolic setting, the number of multiplications only gives a rough estimate of execution time. In any case, the above figures show the importance of being able to define a class that only performs certain operations when it is needed. Again, this is something which can be achieved naturally in C++, by storing a flag in each object \texttt{Basis}. Since a \texttt{Basis} object is only modified by invoking certain member functions, the flag is cleared automatically when the object is modified, effectively invalidating the cached values of \( b_{\alpha j} \).

Remark. The frames associated to a \texttt{Manifold}, \texttt{RiemannianStructure} or \texttt{Connection} object are represented by \texttt{Basis} objects; thanks to the mechanisms described in this section, these frames need not consist of simple elements. In fact, this was the main motivation for introducing \texttt{Basis}.

4. An application: Cartan-Kähler theory

In this section we consider, as an application, the problem of determining whether a linear exterior differential system (EDS) is involutive. This problem eventually boils down to computing the rank of certain matrices, but actually writing down
these matrices requires a system that supports differential forms and interior products, as well as linear algebra; this makes Wedge particularly appropriate to the task.

Suppose we have a real analytic EDS with independence condition \((\mathcal{I}, \theta_1, \ldots, \theta_k)\) on a manifold \(X\). As a special case, we shall take \(X\) to be the bundle of frames over a 7-manifold \(M\), the \(\theta_i\) the components of the tautological form and \(\mathcal{I}\) generated by the exterior derivative of the \(G_2\)-forms \(\phi\) and \(\star\phi\). We wish to determine whether the system is involutive, i.e. every point of \(X\) is contained in some submanifold \(S \subset X\) such that all the forms of \(\mathcal{I}\) vanish on \(S\) and the forms \(\theta_1|_S, \ldots, \theta_k|_S\) are a basis of the cotangent bundle \(T^*S\); we then say that \(S\) is an integral manifold for \(\mathcal{I}\). In the special case, the EDS turns out to be involutive, proving the existence of local metrics with holonomy \(G_2\) (see [3]).

The general procedure is the following. Complete \(\theta_1, \ldots, \theta_k\) to a basis \(\theta_1, \ldots, \theta_k, \omega_1, \ldots, \omega_n\) of 1-forms on \(X\). Suppose that \(E \subset T_xX\) is a \(k\)-dimensional subspace such that the forms \(\theta_1|_E, \ldots, \theta_k|_E\) are independent on \(E\); then one can write

\[
\omega_i|_E = p_{ij}\theta_j|_E.
\]

We say that \(E\) is an integral element if the forms of \(\mathcal{I}\) restrict to 0 on \(E\). The \((x, p_{ij})\) are coordinates on the Grassmannian of \(k\)-planes over \(I\), and by definition the tangent space of an integral manifold at each point will be an integral element \(E\) of the form above. The integral elements of dimension \(n\) form a subset \(V_n\) of the Grassmannian.

Suppose that there exists an integral element \(E \subset T_xX\), and \(V_n\) is smooth about \(E\). By the Cartan-Kähler theory, the (local) existence of an integral manifold \(S\) with \(T_xS = E\) is guaranteed if one can find a flag

\[
E_0 \subset \ldots \subset E_k = E, \quad \text{dim } E_k = k,
\]

such that

\[
\text{dim } V_n = c_0 + \ldots + c_k,
\]

where \(c_j\) is the dimension of the space of polar equations

\[
H(E_j) = \{\alpha(e_1, \ldots, e_j, \cdots) \mid \alpha \in \mathcal{I}, e_l \in E_j\} \subset T_x^*X.
\]

Since we are assuming that \(E_j\) is contained in an integral element \(E\), \(H(E_j)\) has the same dimension as its image \(H^*(E_j)\) under the quotient map

\[
T_x^*X \to \frac{T_x^*X}{\langle \theta_1, \ldots, \theta_k \rangle}.
\]

We shall say that a system is linear if

\[
\mathcal{I} \subset \Lambda^* \text{Span}\{e^1, \ldots, e^k\} \wedge \text{Span}\{\omega_1, \ldots, \omega_n\},
\]

where the span is over \(C^\infty(X)\). This condition implies that the equations defining \(V_n\) are affine in the \(p_{ij}\). In addition, one has \(H^*(E_j) = H^*(E'_j)\), where \(E'_j\) is the horizontal projection of \(E_j\) determined by the splitting

\[
\langle \theta_1, \ldots, \theta_k \rangle \oplus \langle \omega_1, \ldots, \omega_n \rangle.
\]

One can compute the dimension of \(V_n\) and the \(c_i\) with the surprisingly short program of Fig. 2. This program represents the frame bundle \(P\) as a parallelizable manifold with a frame \((\theta_i, \omega_{jk})\), where the \(\theta_i\) are the components of the tautological
form, and the \( \omega_{jk} \) are the components of a torsion-free connection form, related by the structure equation
\[
de^i = e^j \wedge \omega_{ij}.
\]
The member function `GetEquationsForVn` simply replaces every occurrence of \( \omega_{jk} \) with \( p_{jk} \) in terms of the coordinates \( p_{ij} \) of the Grassmannian, and imposes that the coefficients of the resulting forms are zero. The member function `GetReducedPolarEquations`, on the other hand, calls itself recursively to obtain all the forms \( e_{i_1} \cdots e_{i_l} \cdot \alpha \), where \( \alpha \) ranges in a basis of \( \mathcal{I} \) and \( 1 \leq i_1 < \cdots < i_l \leq j \). When the resulting degree is one, the function stops the recursion and stores the differential one-form, applying a substitution corresponding to the projection \( [5] \).

The code of Fig. 2 is for a general linear EDS on the bundle of frames. In order to obtain a result for the special case of \( G_2 \), we can use the following:

```cpp
struct G2 : EDS {
    G2() : EDS (7) {
        lst I;
        I = d(ParseDifferentialForm(e(),"567-512-534-613-642-714-723")),
            d(ParseDifferentialForm(e(),"1234-6712-6734-7513-7542-5614-5623"));
        Basis<symbol> A;
        GetEquationsForVn(A,I);
        for (int j=0;j<7;++j)
            {
                Basis<DifferentialForm> V;
                for (lst::const_iterator i=I.begin();i!=I.end();++i)
                    GetReducedPolarEquations(V,*i,j);
                cout<<V.size()<<endl;
            }
        cout<<A.size()<<endl;
    }
};
```

The output shows that the \( c_i \) are
\[
0, 0, 0, 1, 5, 15, 28
\]
whereas the codimension of \( V_n \) is 49; thus, Cartan’s test applies and the system is involutive.

Notice that the equations on the Grassmannian are stored in a `Basis<symbol>` object, and so the codimension of \( V_7 \) is just the size of the basis. This makes sense because we know that the equations defining \( V_7 \) are linear in the \( p_{ij} \); however, the container \( A \) is passed as a template parameter to `GetEquationsForVn`. This means that if we were dealing with an EDS where the equation are affine in the \( p_{ij} \) rather than linear, e.g. the EDS associated to nearly-Kähler structures in six dimensions, it would suffice to modify the code by declaring \( A \) to be of type `AffineBasis<symbol>`.

The general philosophy, used extensively in Wedge, is that of writing “generic” template functions, without making specific assumptions on the nature of the arguments, so that the type of the arguments determines the precise behaviour of the code.

**Remark.** Concerning the non-linear case, Wedge also provides a class template `PolyBasis`, analogous to `Basis`, where polynomial equations can be stored, relying on CoCoA for the necessary Gröbner basis computations (see [4]). Notice however
struct EDS : public ConcreteManifold, public Has_dTable {
  lst moduloIC;
  int dim;

  EDS(int dimension) : ConcreteManifold(dimension*(dimension+1)) {
    dim=dimension;
    for (int i=1;i<=dim;++i) {
      ex x;
      for (int j=1; j<=dim;++j)
        x+=e(j)*e(i*dim+j);
      Declare_d(e(i),x);
      moduloIC.append(e(i)==0);
    }
  }

  template<typename Container>
  void GetEquationsForVn(Container& container, lst I) {
    lst substitutions;
    for (unsigned i=dim+1; i<=dim*(dim+1);++i) {
      ex ei;
      for (unsigned j=1;j<=dim;++j)
        ei+=symbol("p"+ToString(i,j))*e(j);
      substitutions.append(e(i)==ei);
    }
    for (lst::const_iterator i=I.begin(); i!=I.end();++i)
      GetCoefficients(container,i->subs(substitutions));
  }

  template<typename Container>
  void GetReducedPolarEquations(Container& container, ex form, int j) {
    if (degree<DifferentialForm>(form)==1)
      container.push_back(form.subs(moduloIC));
    else if (j>0) {
      GetReducedPolarEquations(container, form,j-1);
      GetReducedPolarEquations(container, Hook(e(j),form),j-1);
    }
  }
};

Figure 2. Using Wedge to determine if a linear EDS on the frame bundle is involutive

that the simple program of Fig. 2 assumes linearity when computing the polar equations.

References

[1] C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *J. Symbolic Comput.*, 33(1):1–12, 2002.
[2] H. Baum, T. Friedrich, R. Grunewald, and I. Kath. *Twistor and Killing spinors on Riemannian manifolds*. Teubner-Verlag Leipzig/Stuttgart, 1991.

[3] R. Bryant. Metrics with exceptional holonomy. *Annals of Mathematics*, 126:525–576, 1987.

[4] CoCoATeam. CoCoA: a system for doing Computations in Commutative Algebra. Available at http://cocoa.dima.unige.it.

[5] D. Conti. Invariant forms, associated bundles and Calabi-Yau metrics. *J. Geom. Phys.*, 57:2483–2508, 2007.

[6] D. Conti and A. Fino. Calabi-Yau cones from contact reduction, 2007. arXiv:0710.4441v1.

[7] N. J. Hitchin. The self-duality equations on a Riemann surface. *Proc. London Math. Soc. (3)*, 55(1):59–126, 1987.

[8] S. Kobayashi and K. Nomizu. *Foundations of differential geometry*. Interscience Publishers, 1963.

[9] K. Lake and P. Musgrave. GRTensor, a system for making the classical functions of general relativity elementary. In *Proceedings of the 5th Canadian Conference on General Relativity and Relativistic Astrophysics (Waterloo, ON, 1993)*, pages 317–320. World Sci. Publ., River Edge, NJ, 1994.

[10] Barrett O’Neill. The fundamental equations of a submersion. *Michigan Math. J.*, 13:459–469, 1966.

[11] Izu Vaisman. Symplectic curvature tensors. *Monatsh. Math.*, 100(4):299–327, 1985.

Dipartimento di Matematica e Applicazioni, Università di Milano Bicocca, via Cozzi 53, 20125 Milano, Italy.

E-mail address: diego.conti@unimib.it